

**UNIVERSITY OF INFORMATION TECHNOLOGY – VNUHCM
FACULTY OF COMPUTER NETWORKS AND COMMUNICATIONS**

**Nguyễn Đặng Quỳnh Như – 22521050
Nguyễn Phúc Nhi - 22521041**

CRYPTOGRAPHY PROJECT

**CRYSTALS-DILITHIUM: THE IMPLEMENTATION OF DIGITAL
SIGNATURE IN DIGITAL GOVERNMENT**

MAJORING IN INFORMATION SECURITY

**INSTRUCTOR
PhD. Ngoc-Tu Nguyen**

HO CHI MINH CITY, 2024

Acknowledgment

We are truly grateful for your guidance throughout our project. Your constructive feedback was instrumental in enhancing our work. We deeply appreciate the time you spent reviewing our project and providing us with insightful suggestions.

22521050 – Nguyễn Đăng Quỳnh Như – ATTT.2022

22521041 – Nguyễn Phúc Nhi – ATTT.2022

Table of Contents

I. Introduction	4
1.1 Overview	4
II. Background	4
2.1 Digital Signature.....	4
2.2 CRYSTALS-Dilithium.....	5
2.2.1 Overview.....	5
2.2.2 Performance.....	5
2.2.3 Algorithm	6
2.2.4 Scientific Background	6
III. Implementation.....	7
3.1 Context	7
3.2 Related parties	7
3.3 Assets	8
3.4 Risks	8
3.5 Security requirements.....	8
3.6 System design.....	9
3.6.1 Database	9
3.6.2 Modules	10
3.6.2.1 Create CA key and CA-self signed certificate.....	11
3.6.2.2 Create server key and server certificate	13
3.6.2.3 Publish government news in PDF form	16
3.6.2.3.a Generation QR code	20
3.6.2.3.b Sign PDF	21
3.6.2.3.c Upload to database.....	21
3.6.2.4 Download.....	22
3.6.2.5 Search	23
3.6.2.6 Verify	25
3.6.2.6.a Verify public key	30
3.6.2.6.b Verify Signature.....	31
IV. References	32

CRYSTALS-DILITHIUM: THE IMPLEMENTATION OF DIGITAL SIGNATURE IN DIGITAL GOVERNMENT

I. Introduction

1.1 Overview

In our contemporary society, the publishment of administrative documentation via internet has been rendered omnipresent due to their accessibility, especially during the lockdown and quarantine. Therefore, the authentication and integrity regarding governmental records is increasingly of paramount importance than ever due to the greater possibility of unauthorized alterations towards the messages' contents, which would hugely contribute to the distribution of misinformation.

As a result, an implementation of effective measures is essential. Digital signature comes into play because of its features of privacy, authentication, integrity, and non-repudiation, which could act as a safeguard to digitally official files. Additionally, it provides citizens, who are targeted recipients, with confidence to give trust to the accuracy of news published by official bodies.

II. Background

2.1 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

$$S = \text{ESK}(\text{MD})$$

- 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.

$$\text{MD} = \text{DPK}(S)$$

- 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

2.2 CRYSTALS-Dilithium

2.2.1 Overview

Dilithium is one of those post-quantum digital signature algorithms (DSA), which is selected by NIST for standardization. Dilithium's security is based on hardness of finding short vectors in lattice i.e. it's a lattice based Post Quantum Cryptographic (PQC) construction.

2.2.2 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

Figure 2.1: Performance of the Dilithium 3 algorithm

2.2.3 Algorithm

- Dilithium is a digital signature scheme that offers strong security against chosen message attacks, relying on the complexity of lattice problems within module lattices. This security concept ensures that an adversary, even with access to a signing oracle, cannot generate a signature for a message they have not previously seen signed, nor can they create an alternative signature for a message they have already observed being signed. Dilithium is among the candidate algorithms submitted to the NIST post-quantum cryptography project.

```

Gen
01  $\mathbf{A} \leftarrow R_q^{k \times \ell}$ 
02  $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$ 
03  $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ 
04 return  $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$ 

Sign( $sk, M$ )
05  $\mathbf{z} := \perp$ 
06 while  $\mathbf{z} = \perp$  do
07    $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$ 
08    $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$ 
09    $c \in B_{60} := \text{H}(M \parallel \mathbf{w}_1)$ 
10    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ 
11   if  $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ , then  $\mathbf{z} := \perp$ 
12 return  $\sigma = (\mathbf{z}, c)$ 

Verify( $pk, M, \sigma = (\mathbf{z}, c)$ )
13  $\mathbf{w}'_1 := \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$ 
14 if return  $\llbracket \|\mathbf{z}\|_\infty < \gamma_1 - \beta \rrbracket$  and  $\llbracket c = \text{H}(M \parallel \mathbf{w}'_1) \rrbracket$ 

```

Figure 2.2: Dilithium algorithm template

2.2.4 Scientific Background

- ❖ Dilithium is designed using Lyubashevsky's "Fiat-Shamir with Aborts" technique, which leverages rejection sampling to develop secure and compact lattice-based Fiat-

Shamir schemes. The method that results in the smallest signature sizes, created by Ducas, Durmus, Lepoint, and Lyubashevsky, is based on the NTRU assumption and uses Gaussian sampling for signatures. However, due to the difficulties in securely and efficiently implementing Gaussian sampling, we opted for uniform distribution instead. Dilithium builds on the most efficient scheme using only uniform distribution, devised by Bai and Galbraith, by introducing an innovative method that cuts the public key size by over half. As far as we know, Dilithium offers the smallest combined public key and signature size among lattice-based signature schemes that use uniform sampling exclusively.

III. Implementation

3.1 Context

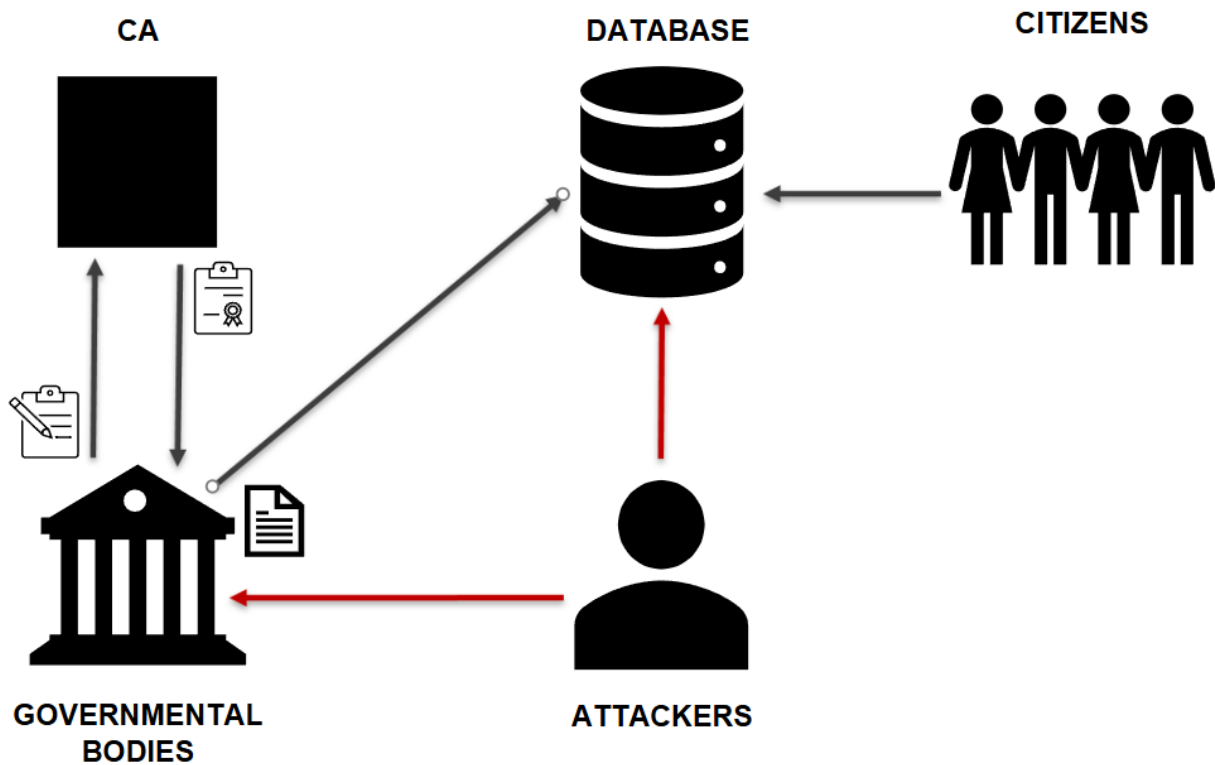


Figure 3.1: Context of issue

3.2 Related parties

- ❖ **Governmental bodies:** they shoulder responsibilities for broadcasting documentations. They sign government news exported in PDF form before uploading them to database

- ❖ Citizens: they are recipients, who are given access to the legal records together with the database in order to keep track with the country's happening
- ❖ Database: this acts as a storage where government upload official records on so that authorized individuals could access, download, and read the documents
- ❖ Attackers: these are organizations or human who crave for modifying documents for unlawful engagement.
- ❖ CA: this is a CA root which is responsible for the provision of certificates for government's public key

3.3 Assets

- Governmental releases under PDF form

3.4 Risks

- Unauthorized entities gaining access to server, leading to the alteration of public key stored in database
- Published documentation's content being modified by intruders
- The modification of digital signed stored in database

3.5 Security requirements

A. Authentication

Authentication serves as proof that an individual has right to publish official records to citizens

B. Integrity

Integrity refers to the ability to protect published news from unintended transformation. By preventing undesirable modifications, digitally official files' integrity can be achieved

C. Non-Repudiation

Non-Repudiation provides the security against denial by third unauthorized party which involved in the communication or having participation in the communication. Hence, when the message is sent to the citizens, then they can prove that it is the alleged sender distributing news

3.6 System design

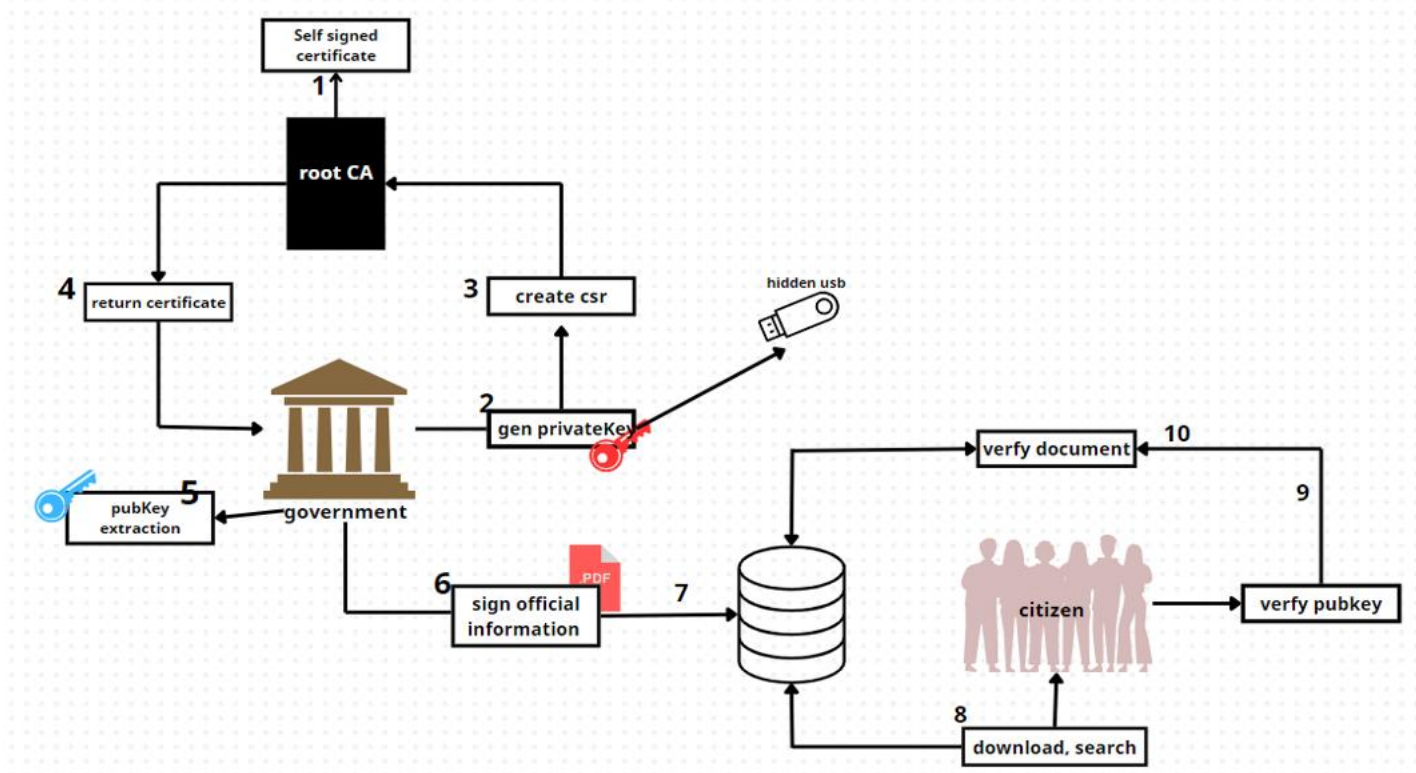


Figure 3.2: The system design

3.6.1 Database

- We utilize MongoDB GridFS because we know MongoDB stores data in BSON format, which has a limit to store only up to 16MB data. The reason behind this is to avoid a single document taking up too much of RAM or over-use the bandwidth during a transaction.
- This is where GridFS comes to the rescue. To store data above 16MB, the GridFS API divides the data into smaller sizes, called chunks. While retrieving, the chunks of data can be combined to get the same data. Each chunk is a binary representation of that part of the data file.

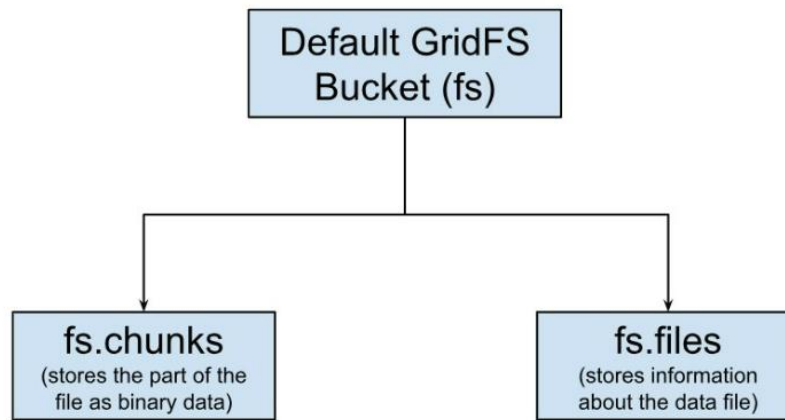


Figure 3.3: Structure of a GridFS Bucket

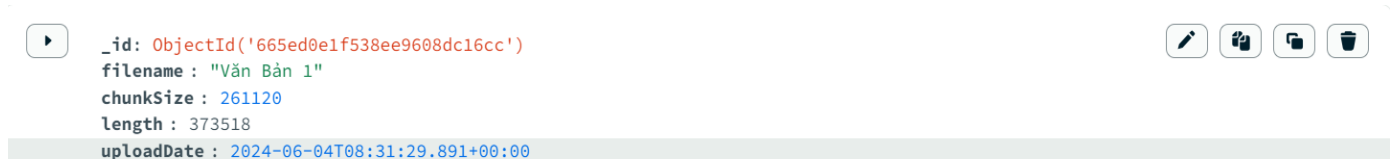


Figure 3.4 The information of the file is stored in the fs.file collection.

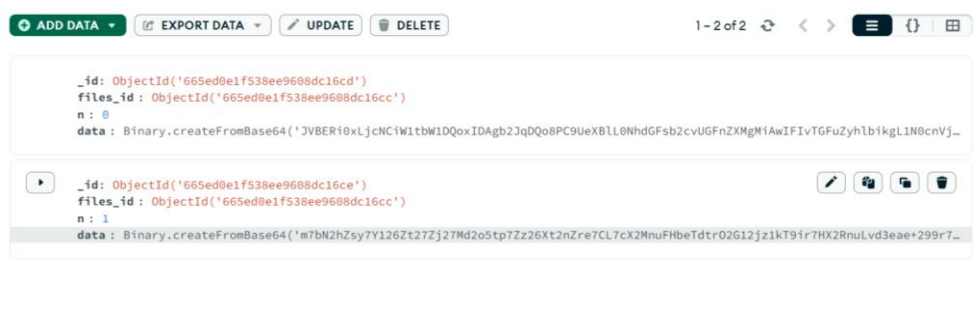


Figure 3.5: The contents of the file are stored in the fs.chunks collection in bytes format

3.6.2 Modules

- Government will enter a password to authenticate permissions and input necessary information to sign the selected governmental document exported under PDF form before publishing it to the database. The signed file will contain a QR code embedding the datetime and publisher information. Citizens can then conduct operations such as searching, downloading, and verifying the downloaded government news.

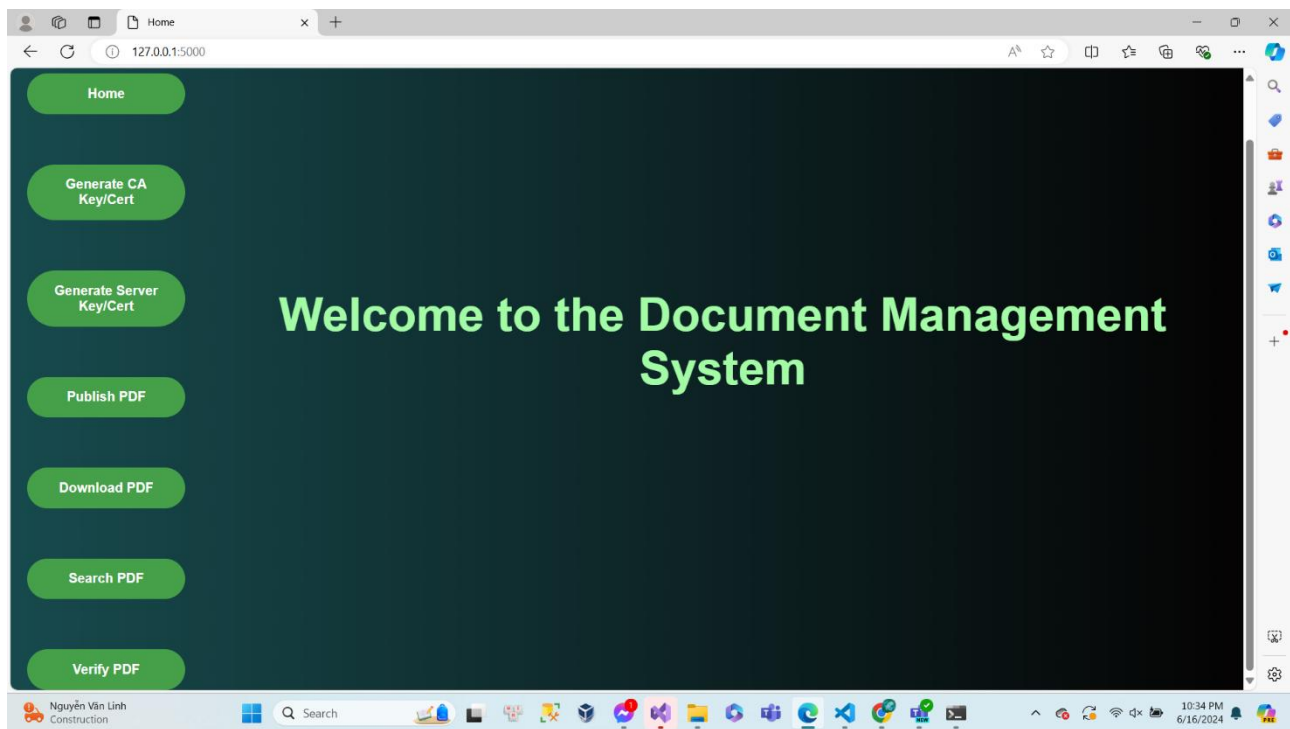


Figure 3.6: Demo web interface

3.6.2.1 Create CA key and CA-self signed certificate

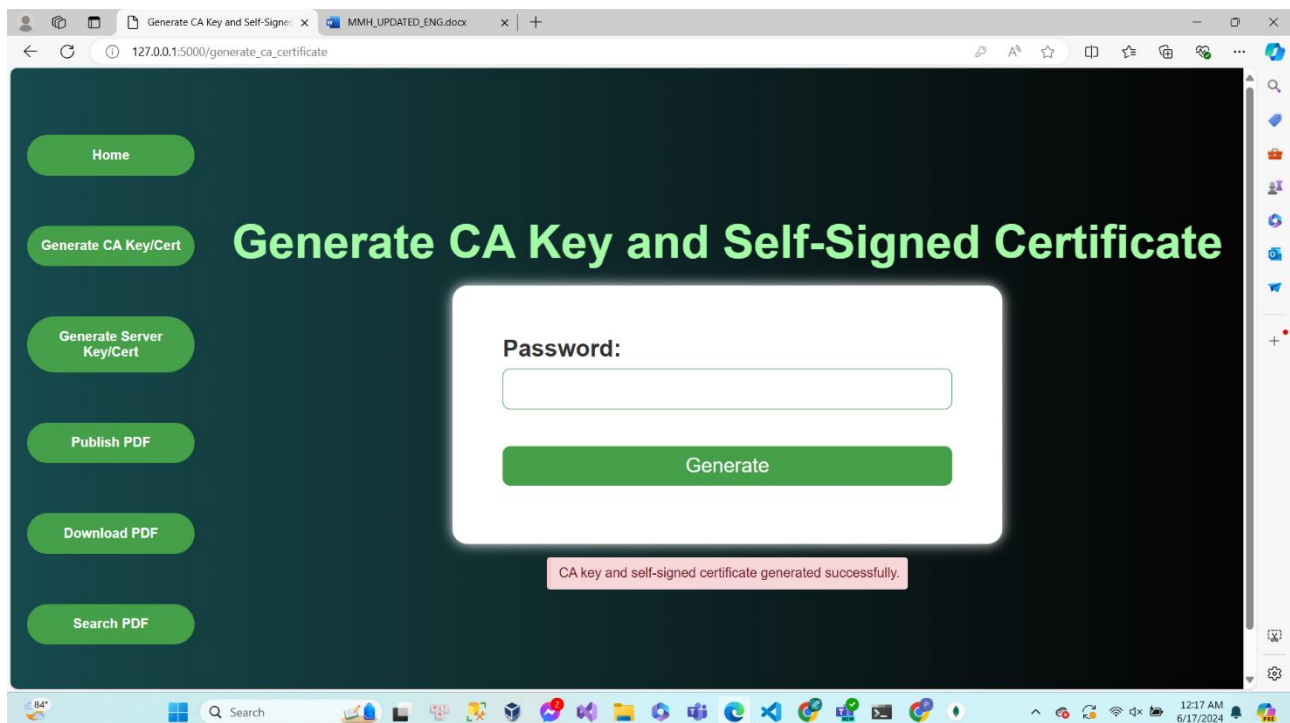


Figure 3.7: Create CA key and CA-self signed certificate

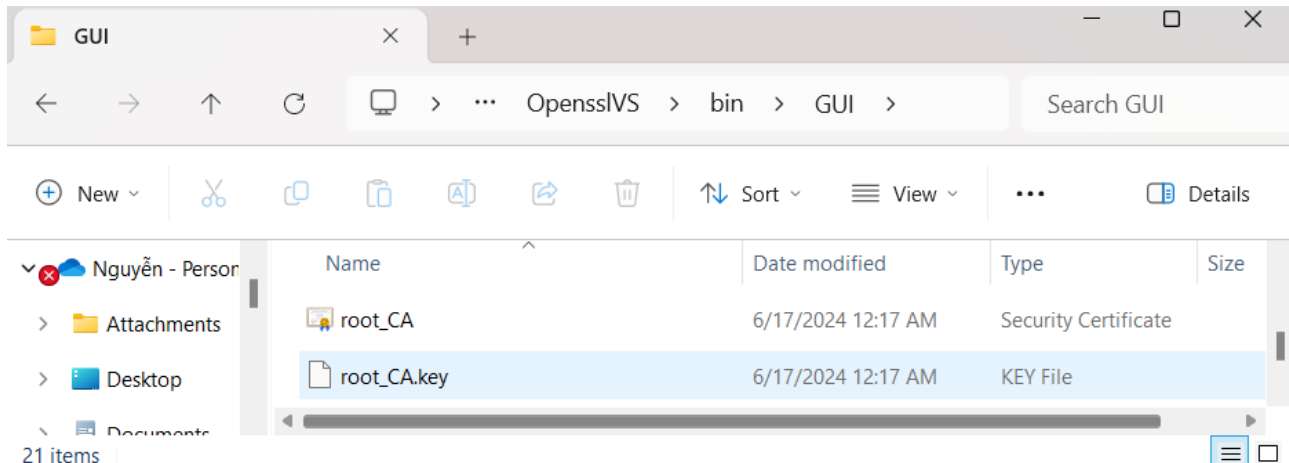


Figure 3.8: CA key and CA-self signed certificate created

- This system enables users to create CA keys and self-signed certificates through the **generate_ca_key_and_self_signed_certificate** function. It provides feedback on the success or failure of the creation process using Flask's flash messages.
- This **generate_ca_key_and_self_signed_certificate** function uses OpenSSL to generate a self-signed CA certificate and key (root_CA.crt and root_CA.key). It uses the dilithium3 algorithm (-newkey dilithium3), which is a post-quantum secure cryptographic algorithm. The certificate is valid for 365 days (-days 365) and is configured to use the oqsprovider provider and a specific OpenSSL configuration file (openssl.cnf).

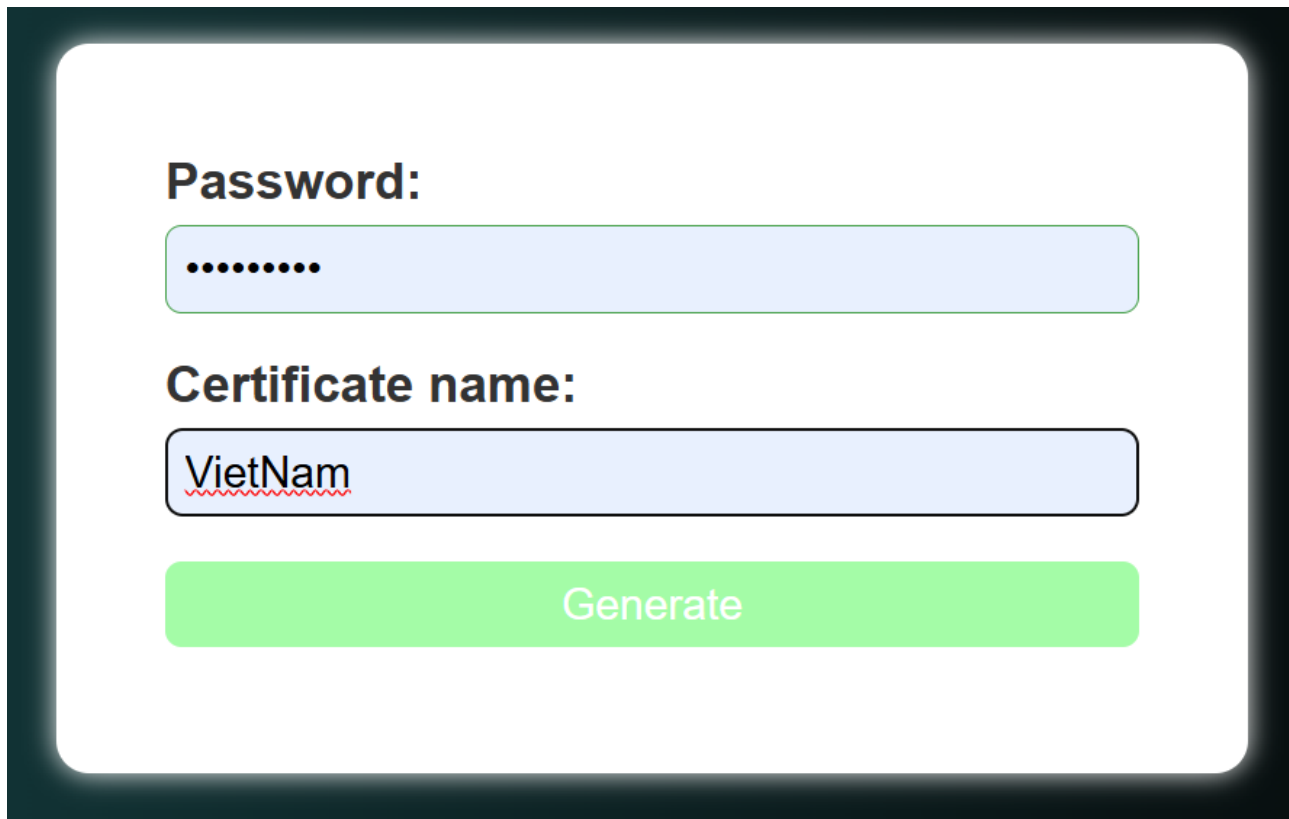
```
1 # Function to generate CA key and self-signed certificate
2 def generate_ca_key_and_self_signed_certificate():
3     try:
4         command = f"{pathOpenssl}" req -x509 -new -newkey dilithium3 -keyout root_CA.key -out root_CA.crt -nodes -subj
5         "/CN=DigiCert" -days 365 -provider oqsprovider -config "C:\\Program Files\\Common Files\\SSL\\openssl.cnf"
6         subprocess.run(command, shell=True, check=True)
7         print("CA key and self-signed certificate generated successfully.")
8     except subprocess.CalledProcessError as e:
9         print(f"Error generating CA key and self-signed certificate: {e}")
10        print(e.stderr.decode())
11
12 @app.route('/generate_ca_certificate', methods=['GET', 'POST'])
13 def generate_ca_certificate():
14     if request.method == 'POST':
15         password = request.form.get('password')
16         if password != '@123admin':
```

```

17     flash("You do not have permission to publish!")
18 else:
19     try:
20         generate_ca_key_and_self_signed_certificate()
21         flash("CA key and self-signed certificate generated successfully.")
22     except Exception as e:
23         flash(f"Failed to generate CA key and certificate: {str(e)}")
24     return redirect(url_for('generate_ca_certificate'))
25 return render_template('generate_ca_certificate.html')
26

```

3.6.2.2 Create server key and server certificate



Password:

.....

Certificate name:

VietNam

Generate

Figure 3.9: Create server key and server certificate (1)

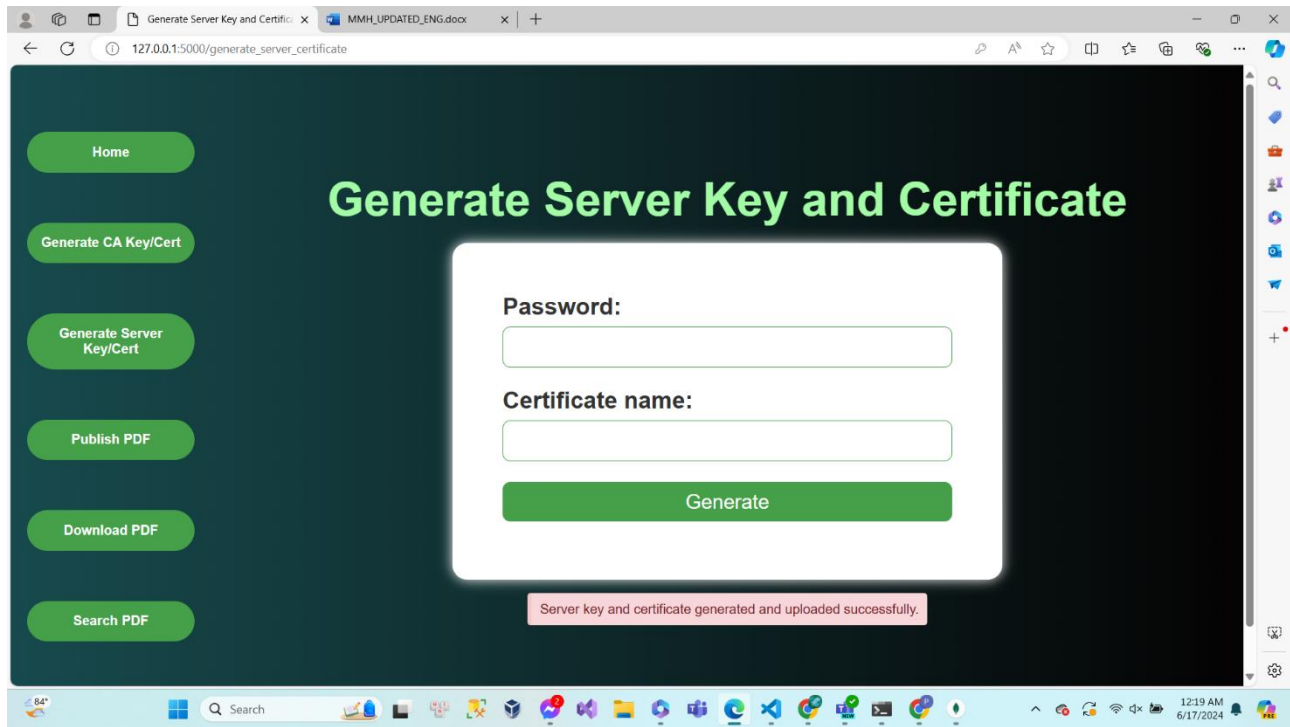


Figure 3.10: Create server key and server certificate (2)

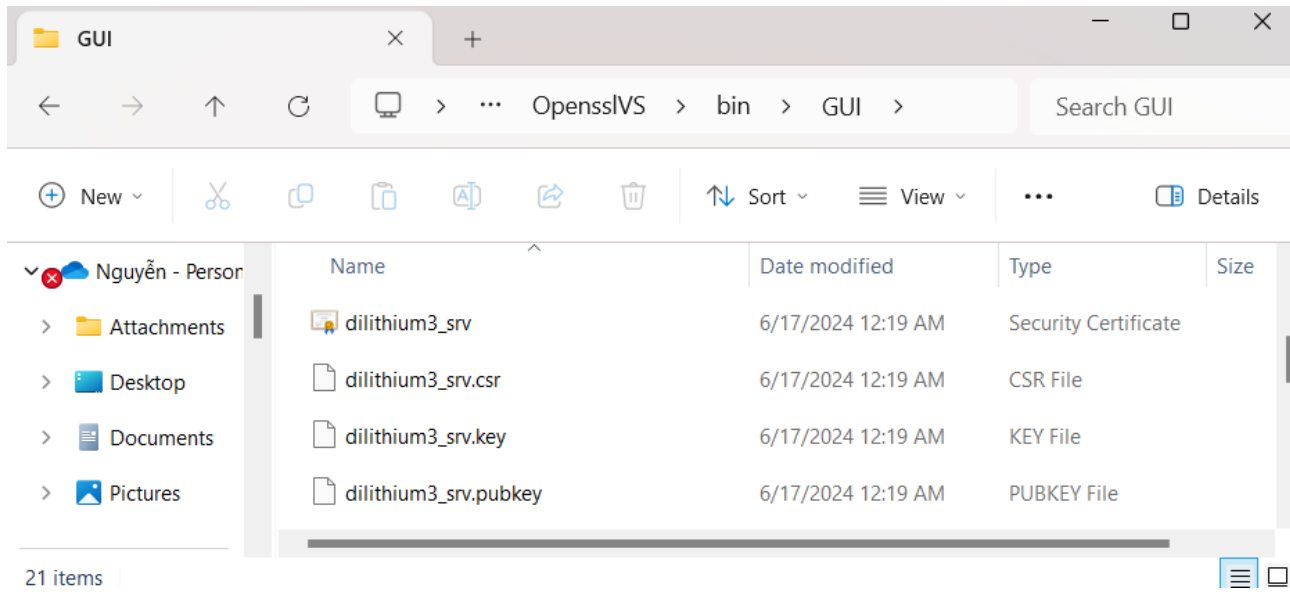


Figure 3.11: Server key and server certificate successfully generated

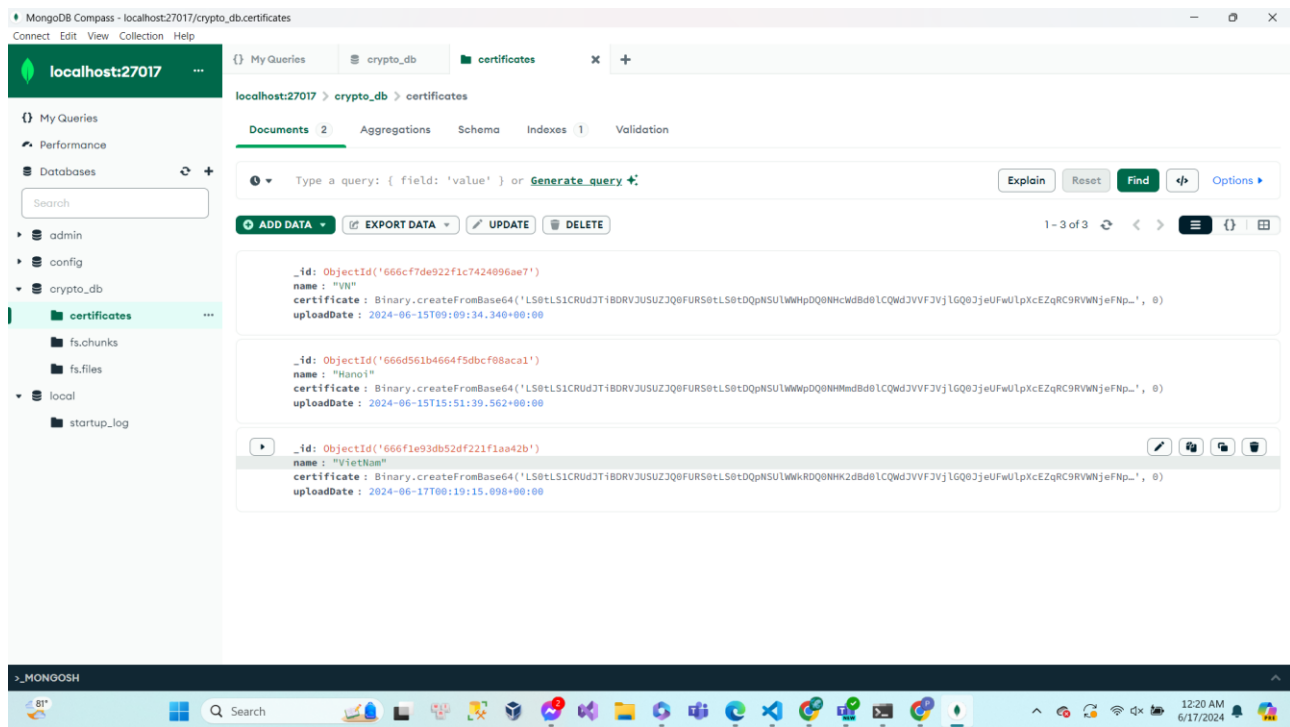


Figure 3.12: Government's certificate uploaded to database

- This system enables authorized users to generate a server key and certificate using the secure post-quantum cryptographic algorithm Dilithium3 through the **generate_server_key_and_certificate** function. It provides feedback on the success or failure of the generation using Flask's flash messages.
- This **generate_server_key_and_certificate** function uses OpenSSL to generate a server key (dilithium3_srv.key) and a certificate (dilithium3_srv.crt). It first generates a private key (genpkey) using the dilithium3 algorithm with the oqsprovider. Then it creates a certificate signing request using the private key and the provided certificate's name for the Common Name (/CN). Finally, it signs (x509) the CSR with the CA key (root_CA.key) and certificate (root_CA.crt), setting a validity period of 365 days.

```

1 # Function to generate server key and certificate
2 def generate_server_key_and_certificate(certname):
3     try:
4         genkey_command = f"{pathOpenssl}" genpkey -algorithm dilithium3 -out dilithium3_srv.key -provider oqsprovider'
5         subprocess.run(genkey_command, shell=True, check=True)
6
7         req_command = f"{pathOpenssl}" req -new -key dilithium3_srv.key -out dilithium3_srv.csr -nodes -subj "/CN={certname}" -
8 provider oqsprovider -config "C:\\Program Files\\Common Files\\SSL\\openssl.cnf"
9         subprocess.run(req_command, shell=True, check=True)

```

```

10
11     sign_command = f"{pathOpenssl}" x509 -req -in dilithium3_srv.csr -out dilithium3_srv.crt -CA root_CA.crt -CAkey root_CA.key
12 -CAcreateserial -days 365 -provider oqsprovider'
13     subprocess.run(sign_command, shell=True, check=True)
14
15     print("Server key and certificate generated successfully.")
16 except subprocess.CalledProcessError as e:
17     print(f"Error generating server key and certificate: {e}")
18     print(e.stderr.decode())
19
20 @app.route('/generate_server_certificate', methods=['GET', 'POST'])
21 def generate_server_certificate():
22     if request.method == 'POST':
23         cert_name = request.form['certname']
24         password = request.form.get('password')
25         if password != '@123admin':
26             flash("You do not have permission to publish!")
27         else:
28             try:
29                 generate_server_key_and_certificate(cert_name)
30                 detachPubKeyFromCert(cert_file, pubkey_file)
31                 upload_certificate_to_db("D:\\OpensslVS\\bin\\GUI\\dilithium3_srv.crt", cert_name)
32                 flash("Server key and certificate generated and uploaded successfully.")
33             except Exception as e:
34                 flash(f"Failed to generate server key and certificate: {str(e)}")
35             return redirect(url_for('generate_server_certificate'))
36     return render_template('generate_server_certificate.html')
37
38
39
40

```

3.6.2.3 Publish government news in PDF form

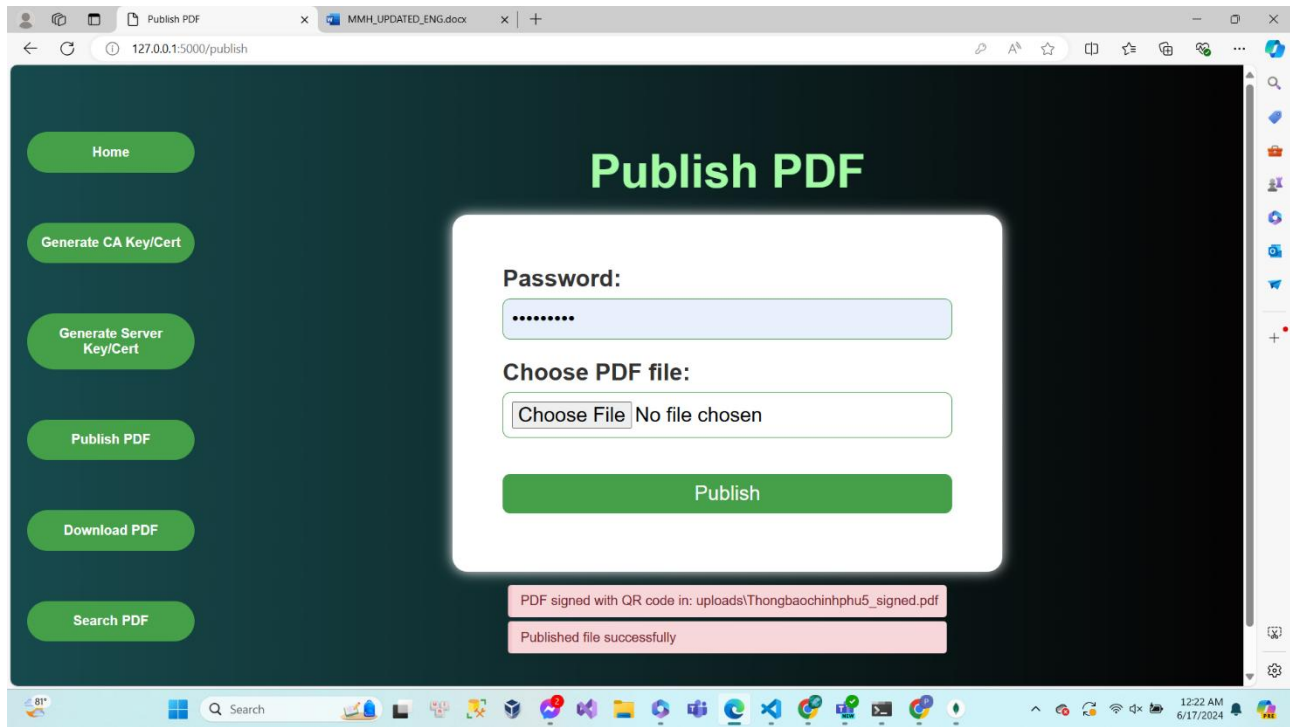


Figure 3.13: Publish governmental documentation

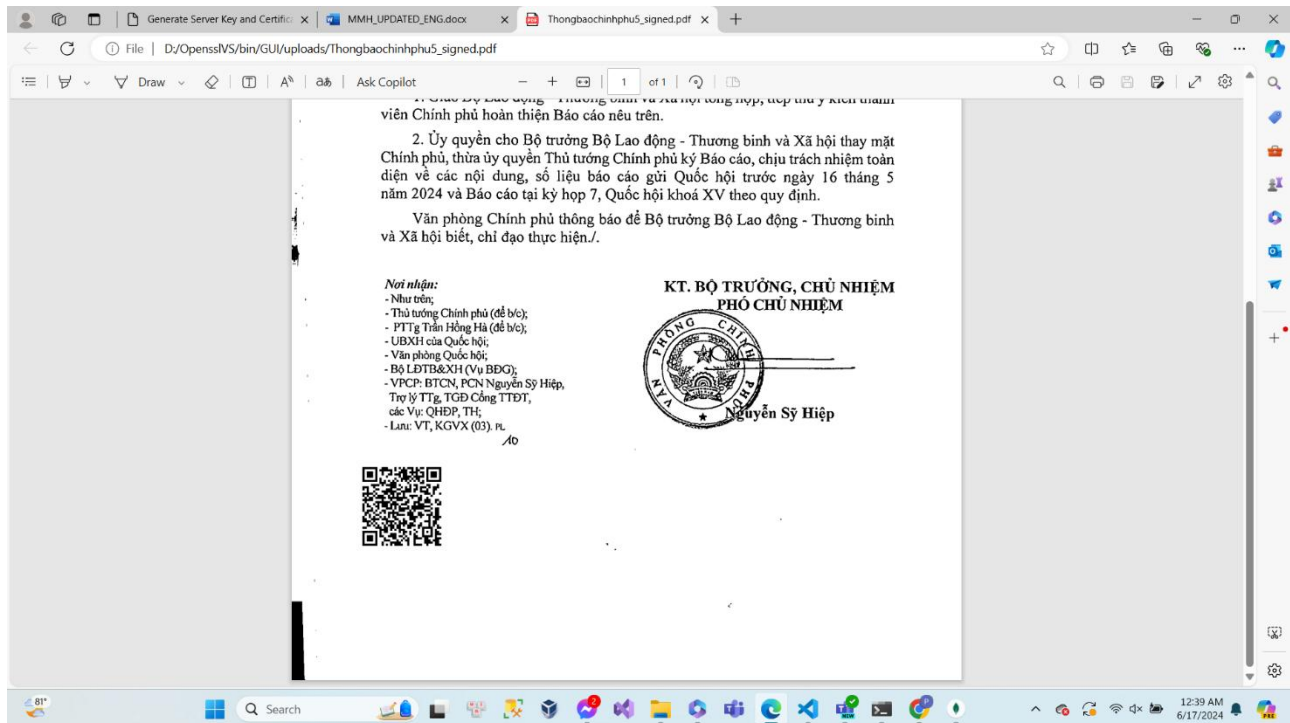


Figure 3.14: A document after being signed contains a QR code

Văn bản

Signed by: chinhphu

Day/time: 17/06/2024 00:22:08

Figure 3.15: QR code after scanning

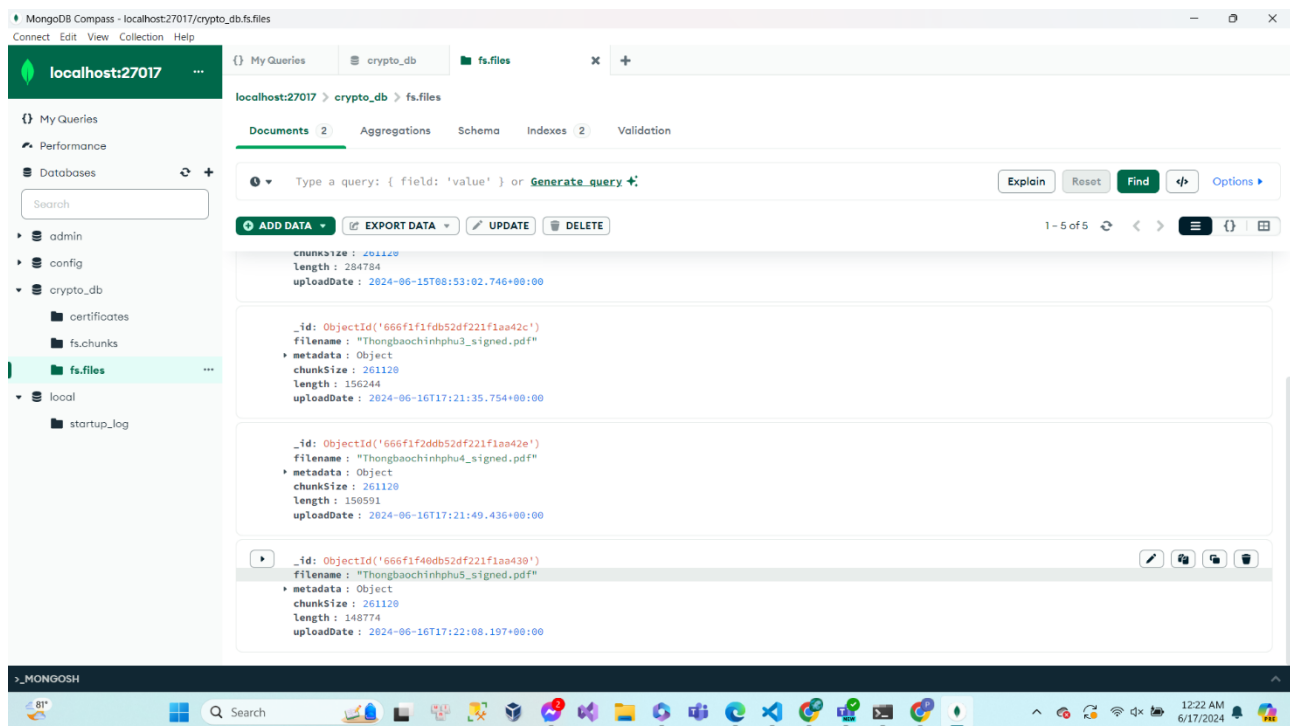


Figure 3.16: Database with published official records

- The **publish** function demonstrates a series of steps to securely publish government records. It commences by validating the request method and verifying the password for authentication. Once authenticated, it retrieves the publisher's account details and the chosen governmental document from the form. After validating the input data, it saves the government news to storage. Next, the function creates a watermark with a QR code containing timestamped information and signs the governmental document exported in PDF form digitally. It then uploads the signed governmental document to MongoDB's GridFS, including metadata like the

public key and signature. Throughout these processes, it handles errors meticulously and provides clear feedback to users, ensuring secure and efficient publication of documents.

```
1 @app.route('/publish', methods=['GET', 'POST'])
2 def publish():
3     if request.method == 'POST':
4         password = request.form.get('password')
5         if password == '@123admin':
6             account = 'chinhphu'
7             pdf_file = request.files.get('pdf_file')
8             if not account or not pdf_file:
9                 flash("Please provide all required inputs: account and PDF file.")
10                return redirect(url_for('publish'))
11            private_key = os.path.abspath('D:\\OpenssIVS\\bin\\GUI\\dilithium3_srv.key')
12            signature_file = "signature"
13
14            # Create a directory to save the uploaded file if it doesn't exist
15            upload_dir = "uploads"
16            if not os.path.exists(upload_dir):
17                os.makedirs(upload_dir)
18
19            # Save the uploaded PDF file
20            pdf_path = os.path.join(upload_dir, secure_filename(pdf_file.filename))
21            try:
22                pdf_file.save(pdf_path)
23            except Exception as e:
24                flash(f"Failed to save uploaded file: {str(e)}")
25                return redirect(url_for('publish'))
26
27            watermark = makeWatermark(account)
28            signed_pdf = makePdf(pdf_path, watermark)
29
30            # Call signData and check if signing was successful
31            if signData(private_key, signed_pdf, signature_file):
32                flash(f"PDF signed with QR code in: {signed_pdf}")
33            else:
34                flash("Failed to sign PDF.")
35                return redirect(url_for('publish'))
36
37            with open("signature", "rb") as sig_file:
38                signature = base64.b64encode(sig_file.read()).decode("utf-8")
39
40            with open("D:\\OpenssIVS\\bin\\GUI\\dilithium3_srv.pubkey", "r") as pubkey_file_obj:
41                public_key = pubkey_file_obj.read()
42
43            if upload_to_gridfs(signed_pdf, public_key, signature):
44                flash("Published file successfully")
45            else:
46                flash("Failed to publish file")
47        else:
48            flash("You do not have permission to publish!")
49            return redirect(url_for('publish'))
50    return render_template('publish.html')
```

3.6.2.3.a Generation QR code

- The **makeWatermark** function generates a governmental document exported in PDF form containing a QR code that includes account information and the timestamp of the signing. The **makePdf** function merges the source PDF file with the watermark file, producing a newly signed official document in PDF. These two functions collaboratively embed a QR code watermark into governmental messages and save the signed file, thereby facilitating the authentication of the document.

```

1 # Function to create QR watermark
2 def makeWatermark(account):
3     watermarkName = "qr.pdf"
4     doc = canvas.Canvas(watermarkName)
5
6     qr = qrcode.QRCode(version=2, error_correction=qrcode.constants.ERROR_CORRECT_H, box_size=10, border=4)
7     now = dt.now()
8     dt_string = now.strftime("%d/%m/%Y %H:%M:%S")
9     qr.add_data(f"Signed by: {account}\nDay/time: {dt_string}")
10    qr.make(fit=True)
11
12    img = qr.make_image(fill_color="black", back_color="white")
13    img_path = "temp_qr.png"
14    img.save(img_path)
15    doc.drawImage(img_path, 20 * mm, 40 * mm, 30 * mm, 30 * mm)
16    doc.save()
17
18    return watermarkName
19
20 # Function to merge PDF with watermark
21 def makePdf(src, watermark):
22     merged = src.replace(".pdf", "_signed.pdf")
23
24     with open(src, "rb") as input_file, open(watermark, "rb") as watermark_file:
25         input_pdf = PdfReader(input_file)
26         watermark_pdf = PdfReader(watermark_file)
27         watermark_page = watermark_pdf.pages[0]
28
29         output = PdfWriter()
30         for i, page in enumerate(input_pdf.pages):
31             if i == 0:
32                 page.merge_page(watermark_page)
33                 output.add_page(page)
34
35         with open(merged, "wb") as merged_file:
36             output.write(merged_file)
37
38     return merged
39

```

3.6.2.3.b Sign PDF

- The **signData** function performs critical steps to create a digital signature for a data file using OpenSSL and a specified private key. First, it constructs an OpenSSL command designed to sign the data file. This command is then executed within a shell environment. The function evaluates the execution result, returning True if the command is successful and False otherwise.

```
1 # Function to sign data
2 def signData(privateKey, dataFile, signatureFile):
3     command = f"{pathOpenssl}" dgst -sha256 -sign "{privateKey}" -out "{signatureFile}" "{dataFile}"
4     result = subprocess.run(command, shell=True)
5     return result.returncode == 0
6
```

3.6.2.3.c Upload to database

- The **upload_to_gridfs** function performs several essential tasks to manage files effectively within MongoDB's GridFS. First, it extracts the file name from a specified path and then proceeds to read the file's content. Subsequently, it creates metadata that includes crucial details such as the author's information, the public key associated with the file, and its digital signature. Once prepared, this metadata accompanies the file as it is uploaded into GridFS, MongoDB's file storage system. Upon successful completion of the upload process, the function prints a notification confirming the successful upload and returns True, indicating the operation's success.

```
1 def upload_to_gridfs(filepath, public_key, signature):
2     filename = os.path.basename(filepath)
3     with open(filepath, "rb") as f:
4         file_data = f.read()
5
6     metadata = {
7         "author": "chinhphu",
8         "public_key": public_key,
9         "signature": signature
10    }
11
12    file_id = fs.put(file_data, filename=filename, metadata=metadata)
13    print(f"File uploaded to GridFS with id: {file_id}")
14    return True
```

3.6.2.4 Download

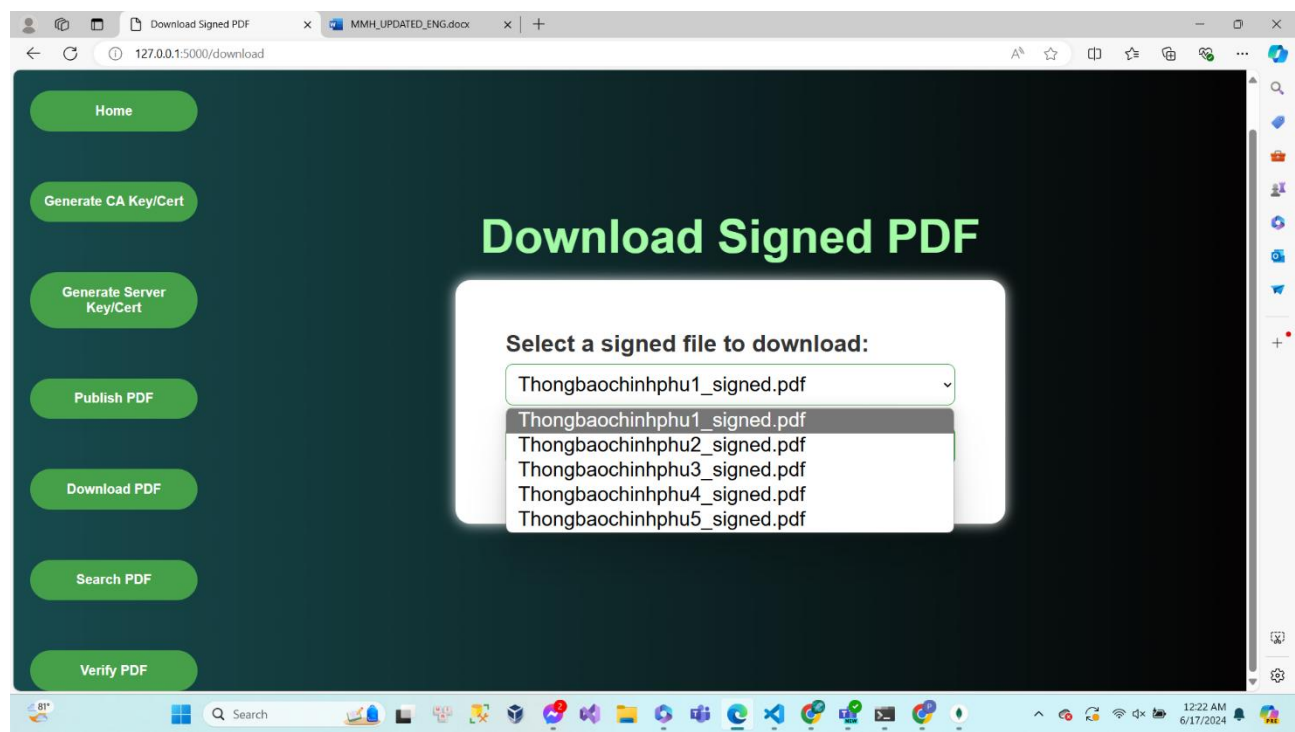


Figure 3.17: List of governmental documentation file in database

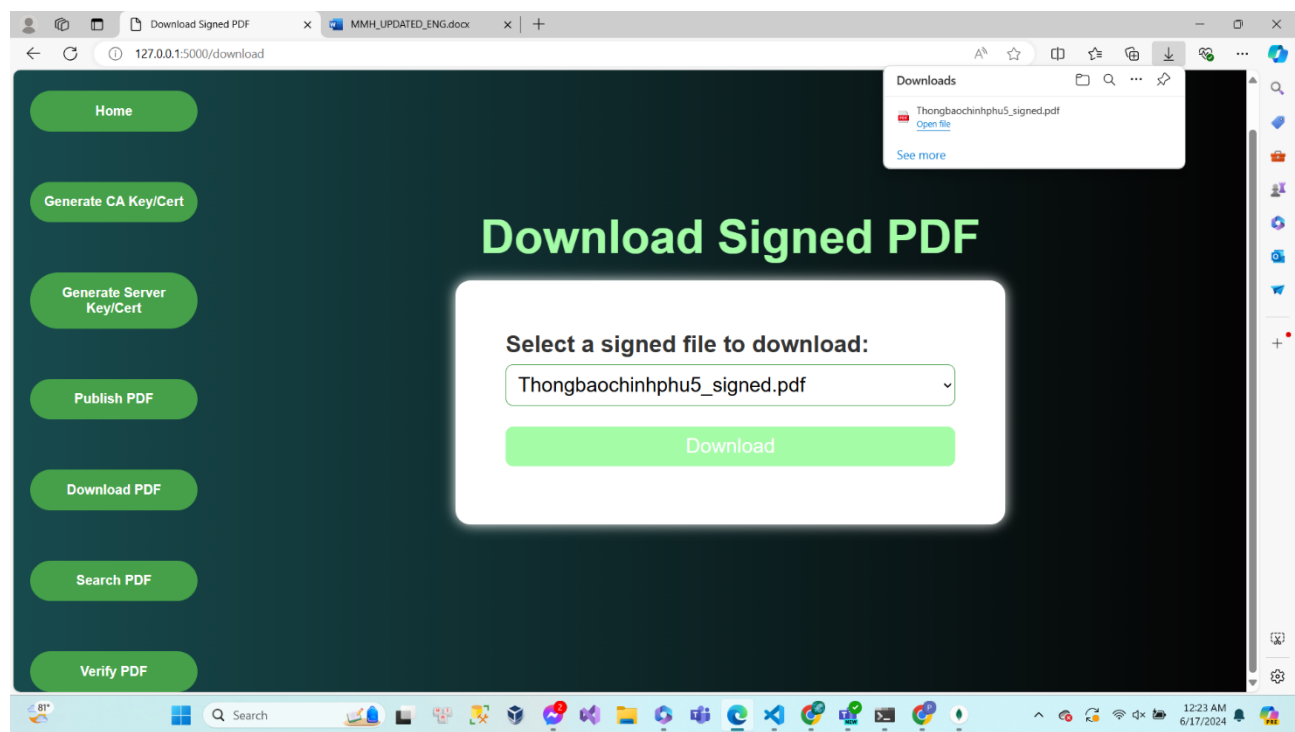


Figure 3.18: Download government news

- The **/download** route facilitates the functionality for users to download signed government news from MongoDB GridFS. The route employs both POST and GET methods to handle different aspects of file retrieval. For the POST method, it processes requests to download a specific signed file based on the **file_id** sent from the form. If the file corresponding to the **file_id** is not found, an error message is displayed, and the user is redirected accordingly. Conversely, the GET method retrieves a list of signed files, enabling users to select and download files of their choice. This approach provides users with flexibility in accessing and retrieving signed documents stored in MongoDB GridFS, ensuring seamless interaction with the application's download functionality.

```
1
2 @app.route('/download', methods=['GET', 'POST'])
3 def download():
4     if request.method == 'POST':
5         file_id = request.form.get('file_id')
6         try:
7             file = db.fs.files.find_one({"_id": ObjectId(file_id), "metadata.signature": {"$exists": True}})
8             if not file:
9                 flash("Signed file not found!")
10                return redirect(url_for('download'))
11
12            grid_out = fs.get(file['_id'])
13            return send_file(grid_out, as_attachment=True, download_name=file['filename'], mimetype='application/pdf')
14
15        except Exception as e:
16            flash(f"Error: {e}")
17            return redirect(url_for('download'))
18
19 # Fetch list of signed files available for download
20 signed_files = db.fs.files.find({"metadata.signature": {"$exists": True}})
21 return render_template('download.html', files=signed_files)
22
```

3.6.2.5 Search

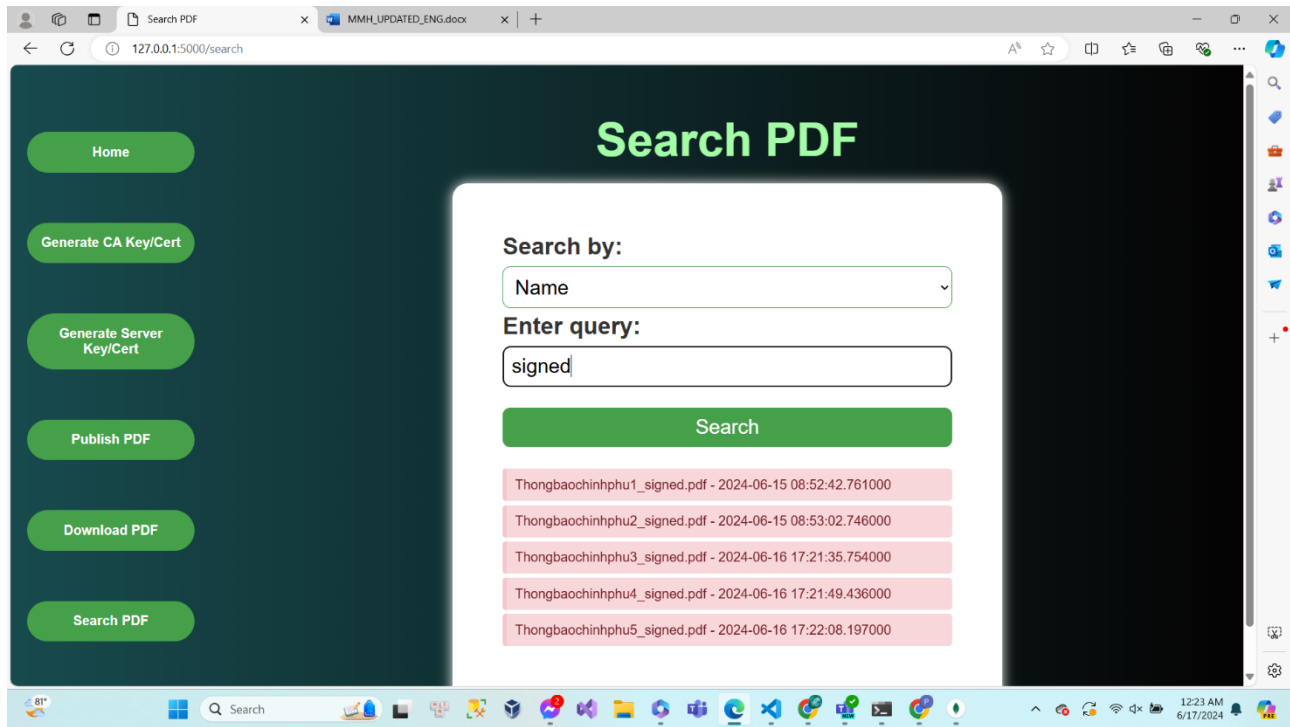


Figure 3.19: Searching file by name

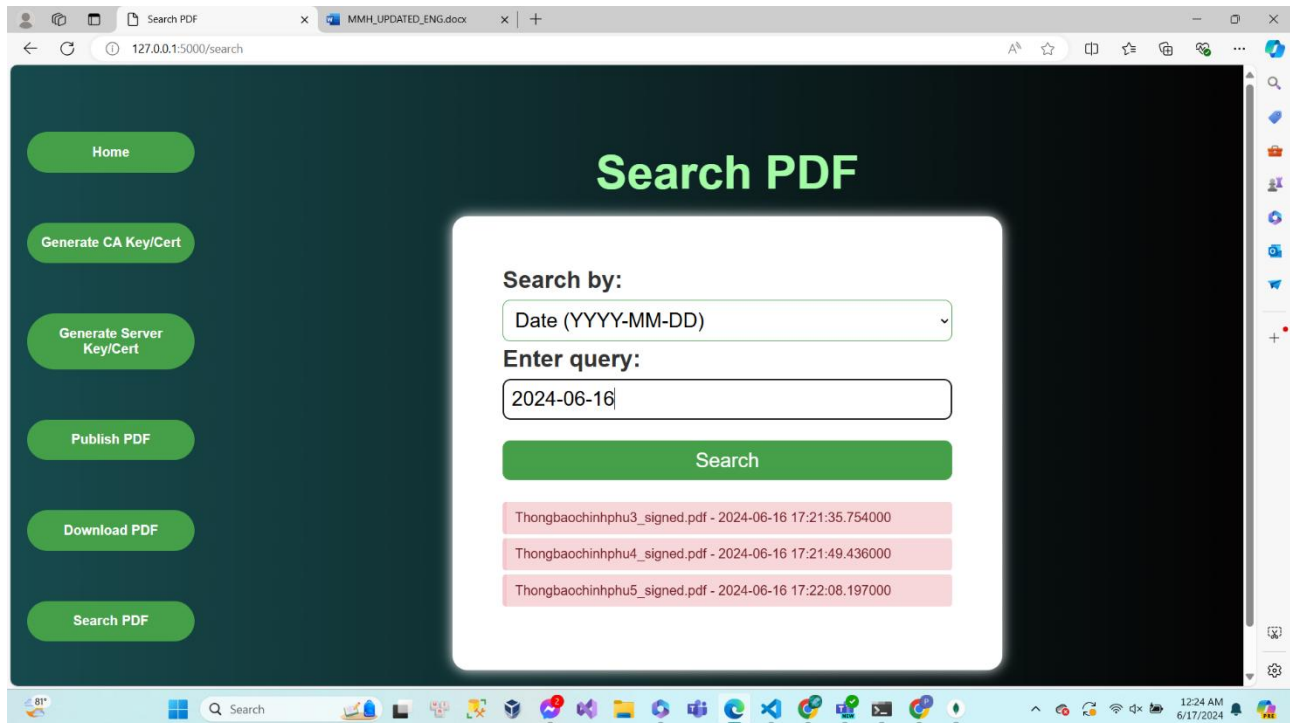


Figure 3.20: Searching file by date

- The functions **find_file_by_name** and **find_file_by_date** are utilized to query and retrieve files from MongoDB GridFS based on file names and upload dates, respectively. These functions are integral components of the ``/search`` route, which provides a user interface for searching and displaying stored files. In operation, the route ``/search`` accommodates requests where users can specify criteria such as file name or upload date through a POST method. Upon receiving such requests, the route executes either **find_file_by_name** or **find_file_by_date** as appropriate. These functions retrieve matching files from the database and return the results as a list formatted for user interface presentation.

```
1
2
3 # Function to find file by name
4 def find_file_by_name(db, partial_filename):
5     regex_pattern = f".*{partial_filename}.*"
6     files = db.fs.files.find({"filename": {"$regex": regex_pattern}})
7     return [{"filename": file['filename'], "uploadDate": file['uploadDate']} for file in files]
8
9 # Function to find file by date
10 def find_file_by_date(db, date_str):
11     date = datetime.strptime(date_str, "%Y-%m-%d")
12     files = db.fs.files.find({"uploadDate": {"$gte": date, "$lt": date + timedelta(days=1)}})
13     return [{"filename": file['filename'], "uploadDate": file['uploadDate']} for file in files]
14
15 @app.route('/search', methods=['GET', 'POST'])
16 def search():
17     if request.method == 'POST':
18         search_type = request.form.get('search_type')
19         query = request.form.get('query')
20
21         if search_type == 'name':
22             files = find_file_by_name(db, query)
23         elif search_type == 'date':
24             files = find_file_by_date(db, query)
25         else:
26             files = []
27
28         return render_template('search.html', files=files)
29
30 return render_template('search.html', files=[])
31
```

3.6.2.6 Verification

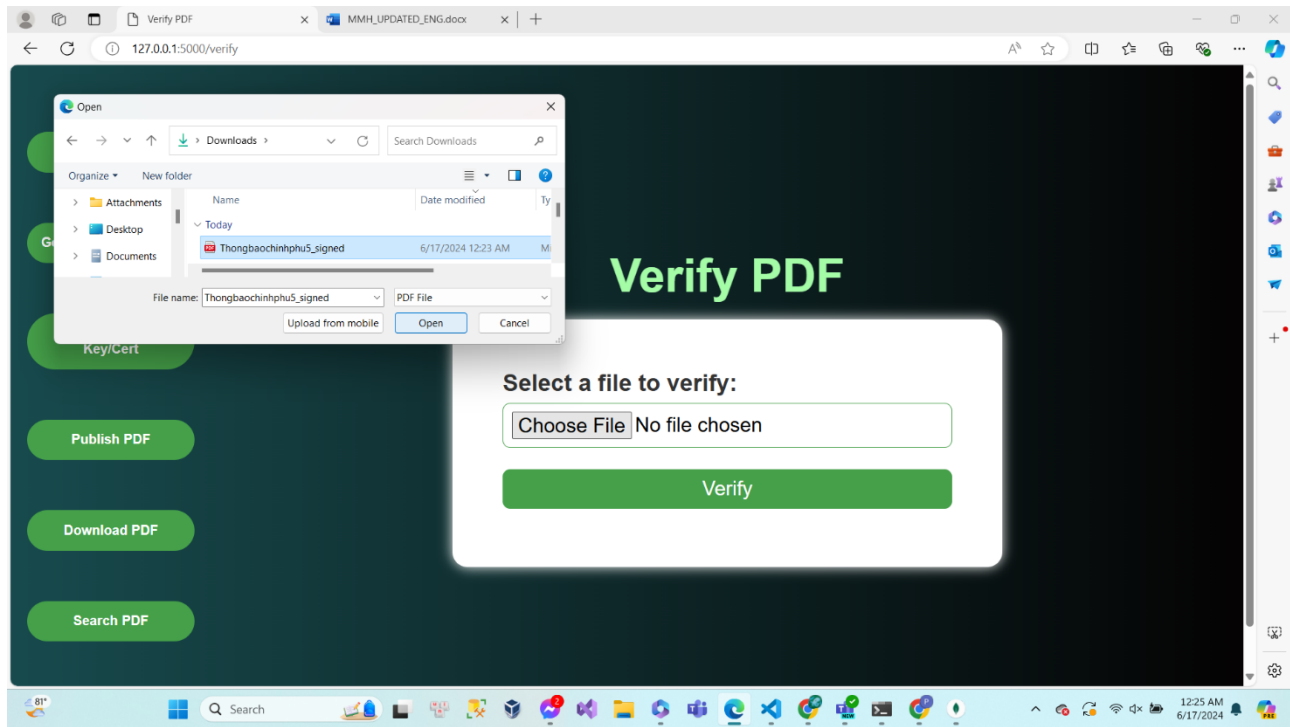


Figure 3.21: Verifying the downloaded government news which is unchanged (1)

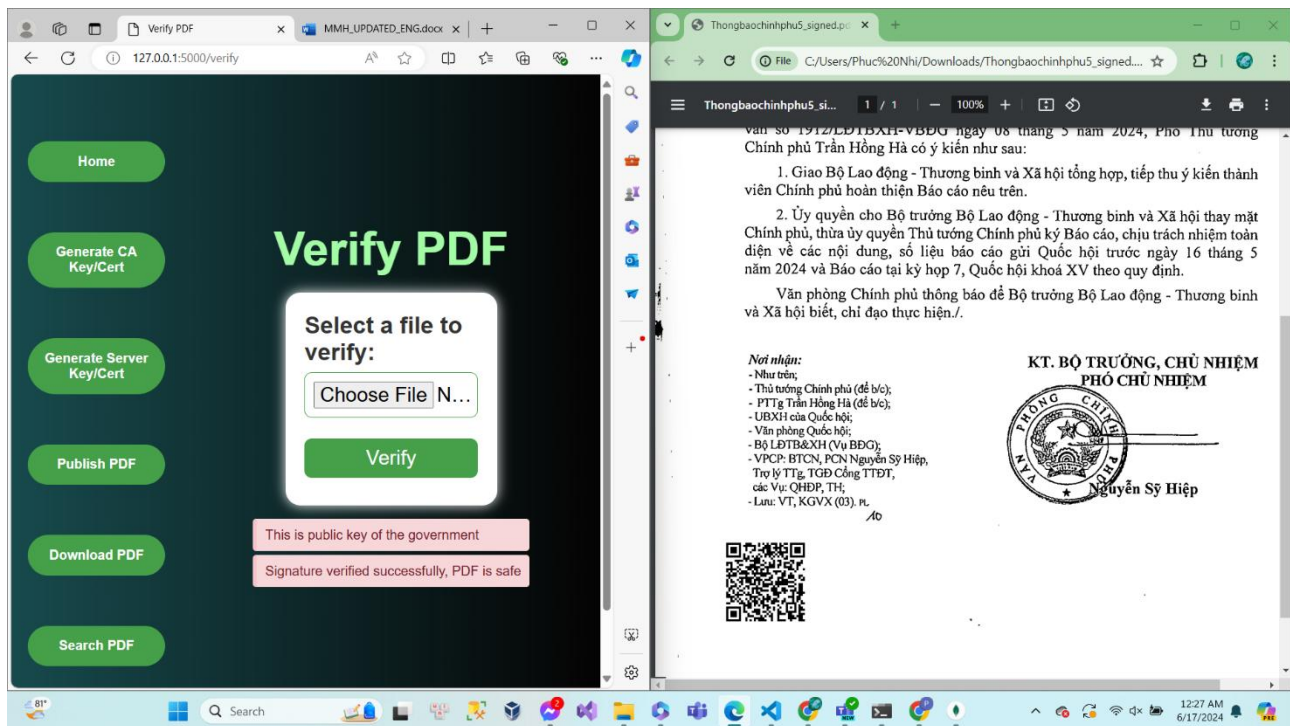


Figure 3.22: Verifying the downloaded government news which is unchanged (2)

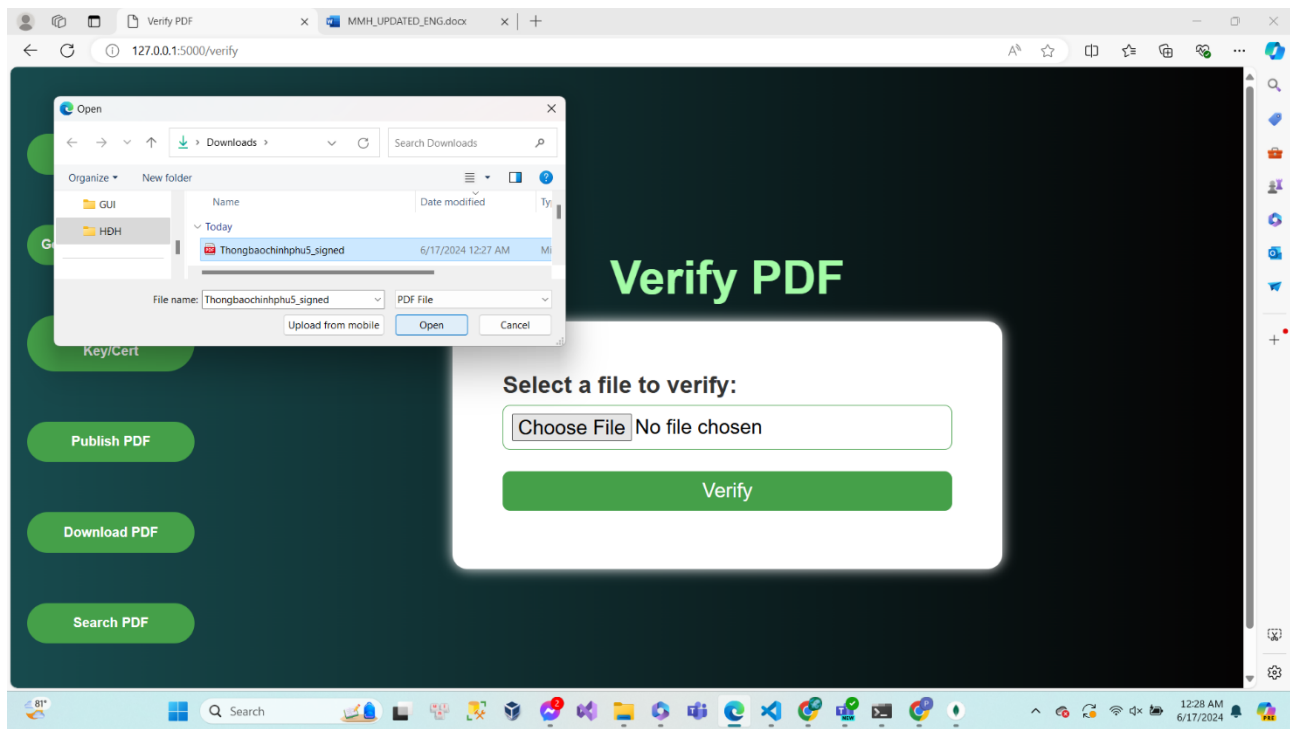


Figure 3.23: Verifying the downloaded government news which is changed (1)

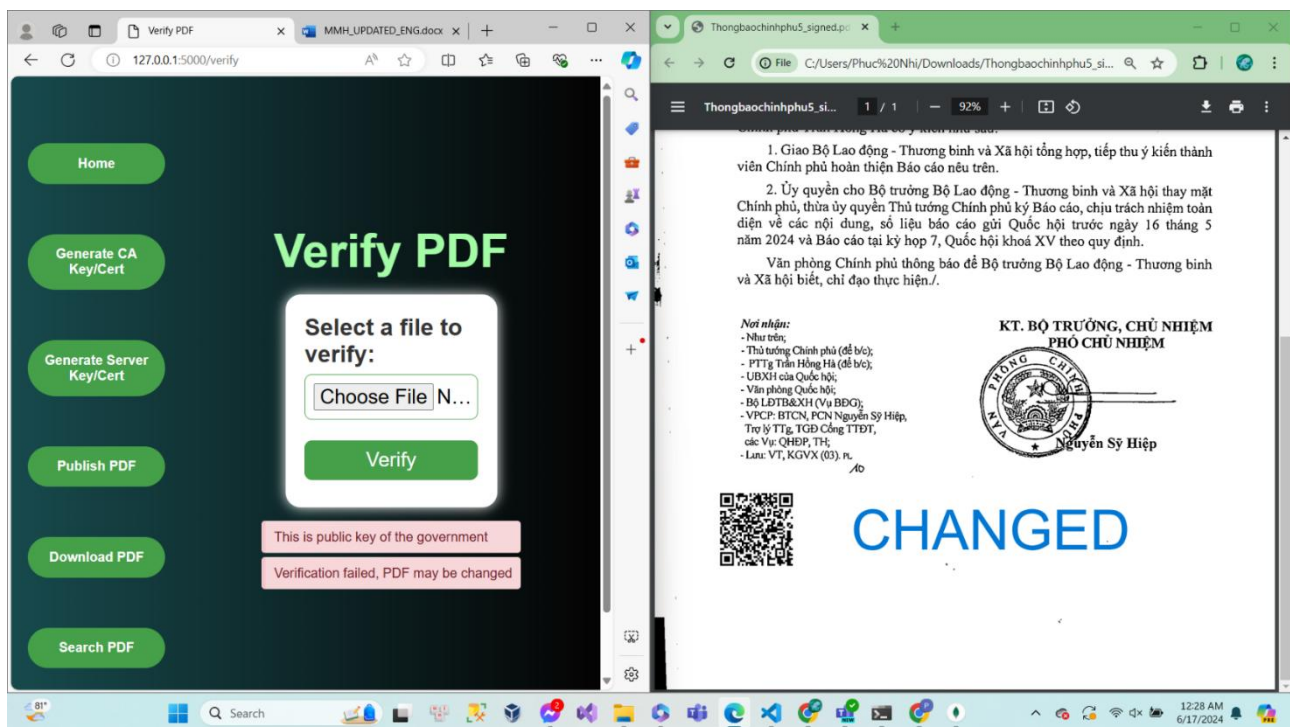


Figure 3.24: Verifying the downloaded government news which is changed (2)

```

▶ _id: ObjectId('666d565a4664f5dbcf08aca2')
  filename: "Thongbaochinhphu1_signed.pdf"
  metadata: Object
    author: "chinhphu"
    public_key: "-----BEGIN PUBLIC KEY-----
                MIIHtDANBgSrBgEEAQKCCwcGBQOCB6EAGIZeQgIuueT..."

▶ _id: ObjectId('666f1f40db52df221f1aa430')
  filename: "Thongbaochinhphu5_signed.pdf"
  metadata: Object
    author: "chinhphu"
    public_key: "-----BEGIN PUBLIC KEY-----
                MIIHtDANBgSrBgEEAQKCCwcGBQOCB6EAkKNDmCTXjhc..."

```

Figure 3.25: Thongbaochinhphu5.pdf with current server public key, Thongtinchinhphu1.pdf with the outdated server public key

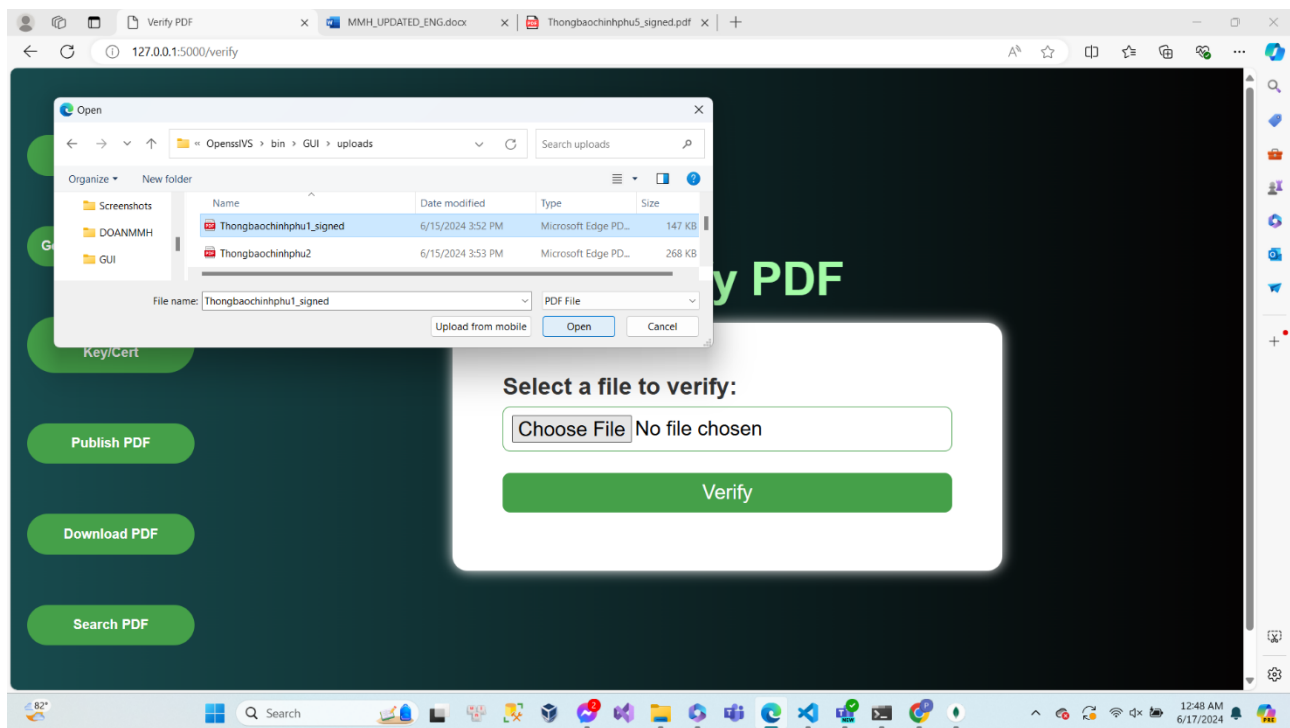


Figure 3.26: Verifying the government news with different public key (1)

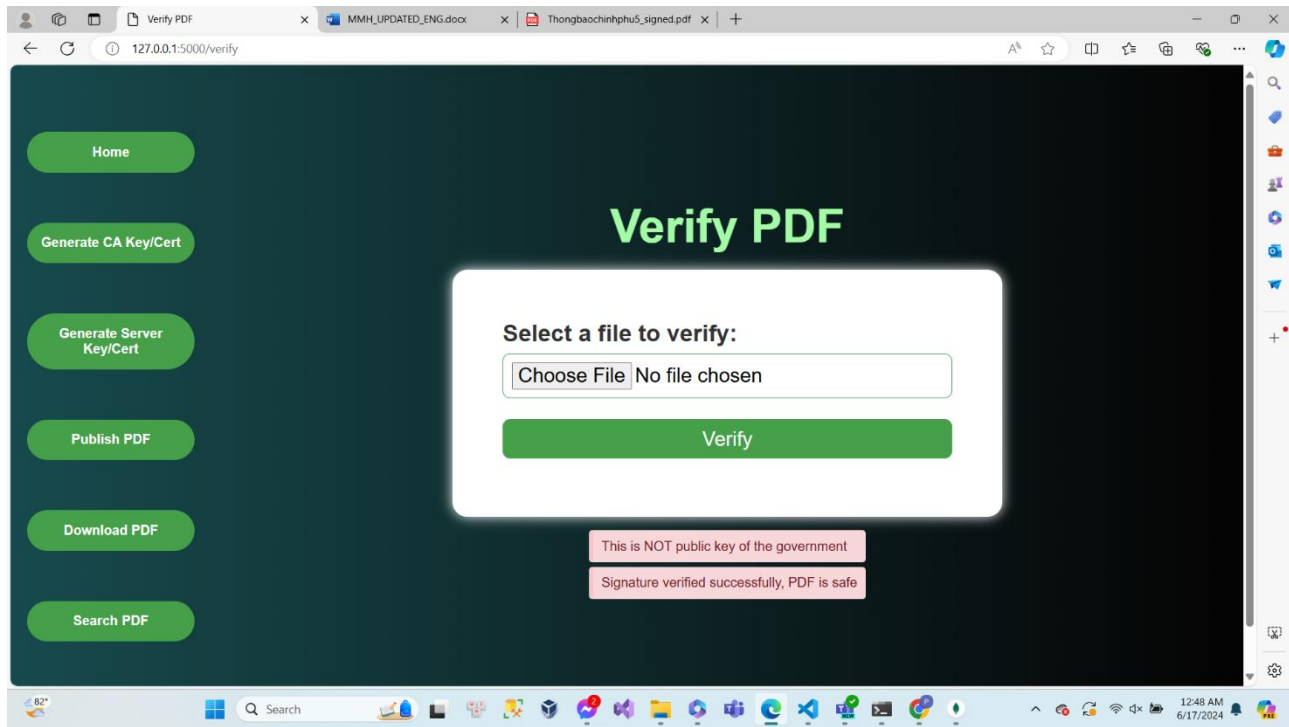


Figure 3.27: Verifying the governmental document with outdated server public key (2)

- The **verify** function is designed to validate digital signatures on governmental document using a systematic approach. Initially, it receives and temporarily stores the official news in PDF form uploaded via a POST request. Subsequently, it queries the MongoDB GridFS database to locate the file based on its filename, retrieving associated metadata such as the public key and signature. The function proceeds by saving the public key to a temporary file and verifying its authenticity through the **verifyPublicKey** function. Following this, it employs the **verifySignature** function to authenticate the governmental document's signature. Upon completion of the verification process, the function informs the user about the outcome, indicating either successful verification or failure. Lastly, to maintain system cleanliness, the function deletes the temporary PDF file and redirects the user back to the verification page.

```

1 @app.route('/verify', methods=['GET', 'POST'])
2 def verify():
3     if request.method == 'POST':
4         # Get the uploaded file
5         pdf_file = request.files.get('pdf_file')
6         if not pdf_file:
7             flash("No file chosen!")
8             return redirect(url_for('verify'))

```

```

9
10 filename = pdf_file.filename
11
12 # Save the uploaded file to a temporary location
13 with tempfile.NamedTemporaryFile(delete=False) as temp_pdf:
14     pdf_path = temp_pdf.name
15     pdf_file.save(pdf_path)
16
17 # Retrieve the file record from the database
18 file_record = db.fs.files.find_one({"filename": filename})
19 if not file_record:
20     flash(f'File '{filename}' not found in the database.')
21     return redirect(url_for('verify'))
22
23 # Extract metadata from the file record
24 metadata = file_record.get('metadata', {})
25
26 # Retrieve public key and signature from metadata
27 public_key_data = metadata.get('public_key', None)
28 signature_data = metadata.get('signature', None)
29
30 if not public_key_data or not signature_data:
31     flash("Public key or signature not found in metadata.")
32     return redirect(url_for('verify'))
33
34 with open("temp_public_key.pub", "w") as pub_file:
35     pub_file.write(public_key_data)
36
37 verifyPublicKey("temp_public_key.pub")
38
39 # Perform signature verification using the temporary files
40 if verifySignature("temp_public_key.pub", pdf_path, signature_data):
41     flash("Signature verified successfully, PDF is safe")
42 else:
43     flash("Verification failed, PDF may be changed")
44
45 # Clean up the temporary PDF file
46 os.remove(pdf_path)
47
48 return redirect(url_for('verify'))
49
50 return render_template('verify.html')

```

3.6.2.6.a Verify public key

- These functions handle SSL/TLS certificates and public keys effectively. The **detachPubKeyFromCert** function extracts the public key from an SSL/TLS certificate and saves it to a file. This separation allows for focused cryptographic operations. The **verifyPublicKey** function validates a provided public key against a stored reference. It communicates the result via a flash message and returns a binary indicator (1 for success, 0 for failure), aiding in decision-making based on key verification status.


```

1 def detachPubKeyFromCert(cert_file, public_key_file):
2     command = f"{pathOpenssl}" x509 -in "{cert_file}" -pubkey -noout -out "{public_key_file}"
3     subprocess.run(command, shell=True, check=True)
4
5 # Function to verify public key
6 def verifyPublicKey(stored_pubkey):
7     # Extract the public key from the certificate
8     extracted_pubkey_file = "extracted_pubkey.pub"
9     detachPubKeyFromCert(cert_file, extracted_pubkey_file)
10
11     with open(extracted_pubkey_file, 'r') as extracted_file:
12         extracted_pubkey = extracted_file.read().strip()
13     with open(stored_pubkey, 'r') as stored_file:
14         stored_pub = stored_file.read().strip()
15
16     if extracted_pubkey == stored_pub:
17         flash("This is public key of the government")
18         os.remove(extracted_pubkey_file)
19         return 1
20     else:
21         flash("This is NOT public key of the government")
22         os.remove(extracted_pubkey_file)
23         return 0
24

```

3.6.2.6.b Verify Signature

- The **verifySignature** function serves to verify digital signatures using OpenSSL. Initially, it decodes the base64-encoded signature and writes it to a temporary file. Subsequently, the function utilizes OpenSSL commands to perform signature verification and returns the verification result. The temporary file is then deleted to ensure no residual data remains in the system after verification, maintaining system cleanliness and security integrity.

```

1 def verifySignature(public_key_file, data_file, signature_base64):
2     # Decode the base64-encoded signature
3     signature = base64.b64decode(signature_base64.encode("utf-8"))
4
5     # Write the decoded signature to a temporary binary file
6     with open("temp_signature.bin", "wb") as sig_file:
7         sig_file.write(signature)
8
9     # Use the temporary signature file in the OpenSSL command
10    command = f"{pathOpenssl}" dgst -sha256 -verify "{public_key_file}" -signature "temp_signature.bin" "{data_file}"
11    result = subprocess.run(command, shell=True, capture_output=True, text=True)
12
13    # Remove the temporary signature file
14    os.remove("temp_signature.bin")
15

```

```
16 return "Verified OK" in result.stdout
17
```

IV. References

- Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., & Stehlé, D. (2021).
Crystals-dilithium algorithm specifications and supporting documentation (Version 3.1)
- 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.