University of Information Technology

Faculty of Computer Network and Communications

PHẠM THANH TÂM – 21522573

PHẠM ĐÔ THÀNH – 21522603

FINAL REPORT

Subject: Cryptography

Class: NT219.N21.ATCL

CRYSTAL-DILITHIUM SIGNATURE TO AUTHENTICATE THE INVOICES IN E-COMMERCE

LECTURER: PhD NGUYỄN NGỌC TỰ

HOCHIMINH CITY, 2023

ACKNOWLEDGEMENTS

To achieve good results in this project, in addition to the wholehearted effort of all members of our team, we also received the dedicated and supportive teaching of our supervisor, Mr. Nguyen Ngoc Tu. His passion in every class has provided us with valuable knowledge not only within the scope of textbooks but also in many other practical and social aspects. With deep and sincere gratitude, we would like to express our appreciation to him for being so enthusiastic and dedicated to his students. It was a great motivation for us to successfully complete this project.

Despite our best efforts, due to limited time and experience, this project may have some shortcomings. We sincerely hope to receive guidance and feedback from our supervisor to supplement and enhance our knowledge to better serve our graduation thesis and future projects. We wish our supervisor good health, success, and always a burning passion for the profession to guide many more generations of students in the future.

We would like to express our sincere thanks!

Pham Thanh Tam

Pham Do Thanh

TABLE OF CONTENTS

1. INTRODUCE

1.1. OVERVIEW

In the digital age, e-commerce is becoming an essential part of the global economy. However, ensuring information security in e-commerce remains a major challenge for businesses and customers.

Invoices and receipts are important documents in the payment process in e-commerce, containing sensitive information such as product information, prices, and buyer and seller information. Protecting the data in invoices and receipts has become a crucial issue to ensure security and trustworthiness in e-commerce.

Applying digital signatures for authentication purposes ensures information integrity, enhances security, and minimizes potential risks and damages. It enables buyers and sellers to authenticate documents from any location and device, making the use of digital signatures essential in maintaining the trust and credibility of invoice data in e-commerce.

1.2. PROBLEM STATEMENT

The problem statement for using digital signatures to secure invoices and receipts is that traditional paper-based invoicing and receipt systems are vulnerable to fraud and tampering, leading to financial losses and legal disputes. These paper-based systems are also time-consuming and costly, making them inefficient for businesses. The use of digital signatures can address these issues by providing a secure and efficient method of verifying the authenticity and integrity of invoices and receipts. However, the adoption and implementation of digital signatures in invoicing and receipt systems face challenges such as the lack of standardization, technical barriers, and resistance to change from stakeholders. These challenges

need to be identified and addressed to promote the widespread use of digital signatures in securing invoices and receipts.

1.2.1 SCENARIO

During the transaction process, the seller creates an invoice with complete invoice information and uses a digital signature to ensure the integrity and security of the data. Afterward, the seller sends the invoice to the buyer.

The buyer receives the invoice from the seller and verifies the validity of the digital signature signed by the seller on the invoice. After successful verification, the buyer creates a receipt and signs it with a digital signature to ensure the integrity and security of the data in the receipt. Once completed, the buyer sends the receipt to the seller and the bank.

The bank receives the receipt from the buyer and proceeds to verify its validity by verifying the validity of the digital signature and other requirements set forth by the bank. If all is valid, the bank transfers the money from the buyer's account to the seller's account.

1.2.2. RELATED PARTIES

- **Seller**: Creates an invoice with complete invoice information and uses a digital signature to ensure the intergrity and security of the data. Send the invoice to the buyer

- **Buyer**: Receives the invoice from the seller and verifies the validity of the digital signature signed by the seller on the invoice. Creates a receipt and

signs it with a digital signature to ensure the integrity and the security of the data in the receipt. Send the receipt to the Seller and the bank

- **Bank**: Receives the receipt from the buyer and proceeds to verify its validity by verifying the validity of the digital signature and other requirements set forth by the bank. If all is valid, the bank transfers the money from the buyer's account to the seller's account

1.2.3. PROTECTION ASSETS

- Invoices and Receipts: The invoices being created are in XML format and the receipts are in HTML format. The consequences of changing the values stored in invoices and receipts can be significant for the payment process and can compromise the integrity of the data. For example, if the amount in the invoice is changed, the payment amount will be incorrect and may result in the buyer or seller not receiving the correct amount. Additionally, if other information in the invoice is changed, such as product information, quantity, or units of measurement, it will affect the authenticity of the invoice and may lead to disputes in the payment process.

1.2.4. SECURITY GOALS

- Integrity: Ensure that data is not modified, altered, or changed in an unauthorized or unreasonable manner without permission or without being recorded.
- Authentication:. Ensuring that the data received by the recipient is sent by the sender and not by someone else.

2. SOLUTION

2.1. OVERVIEW SOLUTION

We are using the CRYSTAL-DILITHIUM, a digital signature algorithm to generate the keys, sign the files, and verify them. These cryptographic operations provide a robust mechanism for ensuring the integrity and authenticity of the invoices and receipts.

Every time an Invoice or Receipt file is created and signed, the PublicKey, the signed Invoice and Receipt files will be stored in MongoDB database.

2.2. CRYSTAL-DILITHIUM DIGITAL SIGNATURE

- Key generation: The key generation algorithm generates a k ×` matrix A each of whose entries is a polynomial in the ring Rq = Zq[X]/(Xn + 1). As previously mentioned, we will always have q = 223 − 2 13 + 1 and n = 256. Afterwards, the algorithm samples random secret key vectors s1 and s2. Each coefficient of these vectors is an element of Rq with small coefficients of size at most η. Finally, the second part of the public key is computed as t = As1 + s2. All algebraic operations in this scheme are assumed to be over the polynomial ring Rq.

Gen

$\mathbf{A} \leftarrow R_q^{k \times \ell}$

$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$

$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$

return $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$

Figure 1. CRYSTAL-DILITHIUM KEY GENERATION

- Signing Procedure: . The signing algorithm generates a masking vector of polynomials y with coefficients less than γ1. The parameter γ1 is set

strategically – it is large enough that the eventual signature does not reveal the secret key (i.e. the signing algorithm is zero-knowledge), yet small enough so that the signature is not easily forged. The signer then computes Ay and sets w1 to be the "high-order" bits of the coefficients in this vector. In particular, every coefficient w in Ay can be written in a canonical way as w = w1 · 2γ2 + w0 where |w0| ≤ γ2; w1 is then the vector comprising all the w1's. The challenge c is then created as the hash of the message and w1. The output c is a polynomial in Rq with exactly τ ±1's and the rest 0's. The reason for this distribution is that c has small norm and comes from a domain of size log2(256 τ) + τ , which we would like to be between 128 and 256. The potential signature is then computed as z = y + cs1.

If z were directly output at this point, then the signature scheme would be insecure due to the fact that the secret key would be leaked. To avoid the dependency of z on the secret key, we use rejection sampling. The parameter β is set to be the maximum possible coefficient of csi . Since c has τ ±1's and the maximum coefficient in si is η, it's easy to see that β ≤ τ · η. If any coefficient of z is larger than γ1 − β, then we reject and restart the signing procedure. Also, if any coefficient of the low-order bits of Az − ct is greater than γ2 − β, we restart. The first check is necessary for security, while the second is necessary for both security and correctness. The while loop in the signing procedure keeps being repeated until the preceding two conditions are satisfied. The parameters are set such that the expected number of repetitions is not too high (e.g. around 4)

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

Figure 2. CRYSTAL-DILITHIUM SIGNING PROCEDURE

- Verification : The verifier first computes w0 1 to be the high-order bits of Az−ct, and then accepts if all the coefficients of z are less than γ1 −β and if c is the hash of the message and w0 1 . Let us look at why verification works, in particular as to why HighBits(Az−ct, 2γ2) = HighBits(Ay, 2γ2). The first thing

to notice is that Az − ct = Ay − cs2. So all we really need to show is that HighBits(Ay, 2γ2) = HighBits(Ay − cs2, 2γ2). (1) The reason for this is that a valid signature will have kLowBits(Ay − cs2, 2γ2)k∞ < γ2 − β. And since we know that the coefficients of cs2 are smaller than β, we know that adding cs2 is not enough to cause any carries by increasing any low-order coefficient to have magnitude at least γ2. Thus Eq. (1) is true and the signature verifies correctly.

$$\underline{\mathsf{Verify}(pk, M, \sigma = (\mathbf{z}, c))}$$
$$\mathbf{w}_1' := \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$$
$$\textbf{if return } [\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!] \textbf{ and } [\![c = \mathsf{H}(M \parallel \mathbf{w}_1')]\!]$$

Figure 3. CRYSTAL-DILITHIUM VERIFICATION

3. IMPLEMENTATION

3.1. TOOLS AND RESOURCES

- Programming Language: Python

Python provides a variety of packages to support this scheme. In this project, our group utilizes Dilithium-py, BeautifulSoup,Element-Tree,…

- Database: MongoDB

MongoDB is an open-source NoSQL database management system designed to store and query data in a document-oriented format, independent of the application schema.

MongoDB is widely used in web applications, mobile applications, IoT systems, and applications that require scalability and flexibility in data storage and querying.

3.2. DESIGN AND CODE

3.2.1 SYSTEM DESIGN

Figure 4. SYSTEM DESIGN

Our design contains 3 main nodes: Seller, Bank, Buyer.

- Seller: Create the invoices, generate the keys, sign the invoices then send to the buyer.
- Buyer: Receives the invoices, create the receipts, generate the keys, sign the receipts then send to the Bank to verify.
- Bank: Receives the receipts, verify the receipts.

3.2.2 SELLER

Đầu tiên người bán sẽ tạo file Invoice ở định dạng XML

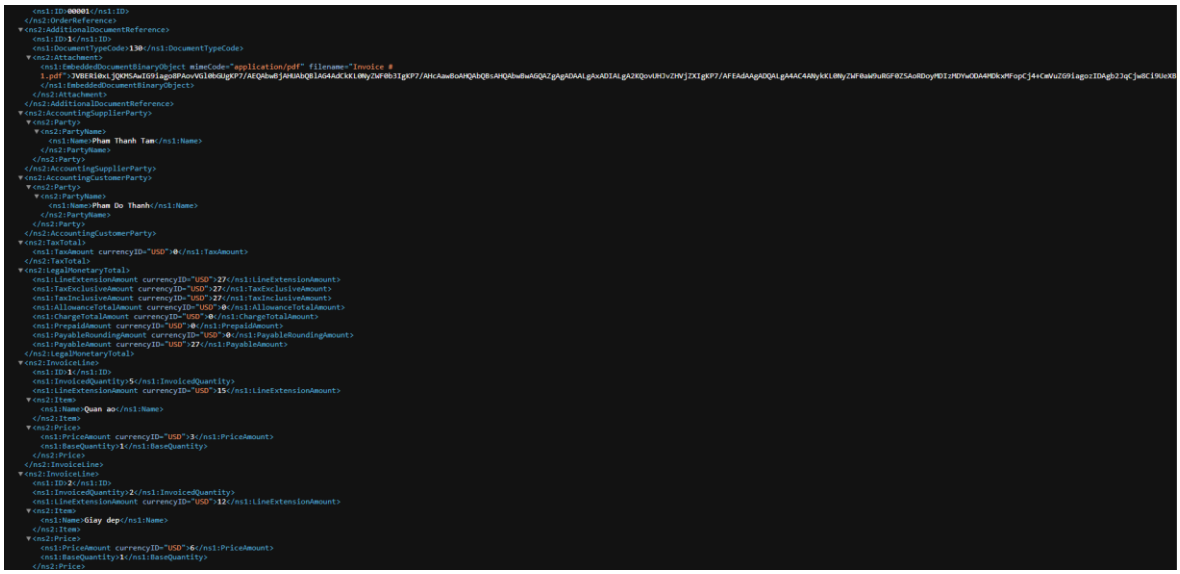


Figure 5. INVOICES

```
def KEYGEN():

    pk, sk = Dilithium2.keygen()

    pk_hex = pk.hex()

    with open("PublicKey.txt", "w") as f:

        f.write(pk_hex)
```

```
    return pk_hex, sk
```

Dilithium2.keygen() generates a new random public-private key pair using the Dilithium2 algorithm.

pk.hex() converts the public key pk to its hexadecimal string representation.

The with statement opens a file named "PublicKey.txt" in write mode and writes the hexadecimal string representation of the public key to the file using the write() method.

```
def SIGN(file_path, sk, client):
    pk = KEYGEN()
    #read the XML file
    with open(file_path, 'r') as f:
        xml_string = f.read()

    # Parse the XML string and get the root element
    root = ET.fromstring(xml_string)

    # Get the canonical representation of the XML string
    canonical_xml = ET.tostring(root, encoding="unicode", method="xml")

    # Sign the canonical XML using Dilithium
    sig = Dilithium2.sign(sk, canonical_xml.encode())

    # save the signature to mongoDB
    db = client["Invoices"]
    collection = db["signed_xml_files"]
```

```
result = collection.insert_one({"xml_file": xml_string, "signature": sig, "publickey": pk})


# add signature element to the XML file
signature_element = ET.SubElement(root, "signature")
signature_element.text = sig.hex()


# save the XML file with signature
xml_with_signature = ET.tostring(root)
with open(file_path, 'w') as f:
    f.write(xml_with_signature.decode())

return sig
```

The XML file at the file_path location is read using the open() function in read mode.

The XML string is parsed and the root element is obtained using ET.fromstring(xml_string).

The canonical representation of the XML string is obtained using ET.tostring(root, encoding="unicode", method="xml").

The Dilithium2.sign() method is called with the secret key sk and the canonical XML string to generate the digital signature sig.

The sig and xml_string are saved to MongoDB using the insert_one() method on a collection named "signed_xml_files" in the "Invoices" database. The public key is also saved to the database as "publickey".

A new XML element named "signature" is created using ET.SubElement(root, "signature"), and its text is set to the hexadecimal representation of sig using signature_element.text = sig.hex().

The XML file with the newly added signature element is saved to the original file path using with open(file_path, 'w') as f: f.write(xml_with_signature.decode()).

Figure 6. INVOICES WITH SIGNATURE

3.2.3 BUYER

```
def verify(public_key_file, xml_file):
    # Load the public key
    with open(public_key_file, 'r') as f:
        pk_hex = f.read()
    pk = bytes.fromhex(pk_hex)

    # Load the XML file and extract the signature value
    tree = ET.parse(xml_file)
    root = tree.getroot()
    signature_element = root.find("signature")
    while signature_element is not None:
        # Extract the signature value and remove the signature element
        signature_value = signature_element.text.strip()
        root.remove(signature_element)

        # Get the canonical representation of the XML string
        canonical_xml = ET.tostring(root, encoding="unicode", method="xml")

        # Verify the signature using Dilithium
        try:
            is_valid = Dilithium2.verify(pk, canonical_xml.encode(), bytes.fromhex(signature_value))
            if is_valid:
                print("The signature is valid.")
```

```
        else:
            print("The signature is not valid.")
    except Exception as e:
        print("An error occurred while verifying the signature:", str(e))

    # Find the next signature element
    signature_element = root.find("signature")

# Write the updated XML file
tree.write(xml_file)
```

The code defines three functions for signing and verifying XML files using the Dilithium2 algorithm for digital signatures and MongoDB for data storage.

The KEYGEN() function generates a new public-private key pair using Dilithium2 and saves the public key to a file named "PublicKey.txt".

The SIGN(file_path, sk, client) function signs an XML file using the Dilithium2 algorithm and saves the signature, XML file, and public key to MongoDB. The function also adds the signature element to the XML file.

The verify(public_key_file, xml_file) function verifies the digital signature of an XML file using the Dilithium2 algorithm and the corresponding public key, and removes the signature element from the XML file if it is valid.

Overall, these functions provide a basic implementation of digital signatures using the Dilithium2 algorithm for securing XML files and storing them in a database.

```
def SIGN(file_path, sk, client):
    pk = KEYGEN()
    #read the HTML file
```

```
with open(file_path, 'r') as f:
    html_string = f.read()

# Parse the HTML string and get the root element
soup = BeautifulSoup(html_string, 'html.parser')

# Get the canonical representation of the HTML string
canonical_html = str(soup)

# Sign the canonical HTML using Dilithium
sig = Dilithium2.sign(sk, canonical_html.encode())

# save the signature to MongoDB
db = client["Receipt"]
collection = db["signed_receipt_files"]
result = collection.insert_one({"html_file": html_string, "signature": sig, "publickey": pk})

# add signature element to the HTML file
signature_element = soup.new_tag("signature")
signature_element.string = sig.hex()

if soup.body is not None:
    soup.body.append(signature_element)
elif soup.html is not None:
    soup.html.append(signature_element)
else:
    soup.append(signature_element)
```

```
    # save the HTML file with signature
    html_with_signature = str(soup)
    with open(file_path, 'w') as f:
        f.write(html_with_signature)

    return sig
```

The SIGN function signs an HTML file using the Dilithium2 algorithm, saves the signature, HTML file, and public key to MongoDB, and adds the signature element to the HTML file.

The function reads an HTML file, parses it using BeautifulSoup, generates a digital signature for the canonical representation of the HTML file using Dilithium2, saves the signature, HTML file, and public key to MongoDB, and appends a new signature element to the HTML file. The signature element is appended to the body or html element of the HTML file, depending on which one is present. If neither is present, the signature element is appended to the end of the HTML file.

Once the signature element has been appended to the HTML file, the function saves the updated HTML file to the original file path and returns the digital signature

```
def generate_receipt(xml_file_path, html_file_path):
    # Định nghĩa các định danh namespace
    ns0 = 'urn:oasis:names:specification:ubl:schema:xsd:Invoice-2'
    ns1 = 'urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2'
    ns2 = 'urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2'

    # Đọc dữ liệu từ tài liệu XML
    xmlData = ET.parse(xml_file_path)

    # Trích xuất các giá trị tương ứng từ tài liệu XML
```

```
    supplierName = xmlData.findtext(f'.//{{{ns2}}}AccountingSupplierParty/{{{ns2}}}Party/{{{ns2}}}PartyName/{{{ns1}}}Name')
    customerName = xmlData.findtext(f'.//{{{ns2}}}AccountingCustomerParty/{{{ns2}}}Party/{{{ns2}}}PartyName/{{{ns1}}}Name')
    lineItems = []
    for item in xmlData.getroot().findall(f'.//{{{ns2}}}InvoiceLine'):
        itemID = item.find(f'.//{{{ns1}}}ID').text
        itemName = item.find(f'.//{{{ns2}}}Item/{{{ns1}}}Name').text
        itemQuantity = item.find(f'.//{{{ns1}}}InvoicedQuantity').text
        itemPrice = item.find(f'.//{{{ns2}}}Price/{{{ns1}}}PriceAmount').attrib['currencyID'] + ' ' + item.find(f'.//{{{ns2}}}Price/{{{ns1}}}PriceAmount').text
        itemTotal = item.find(f'.//{{{ns1}}}LineExtensionAmount').attrib['currencyID'] + ' ' + item.find(f'.//{{{ns1}}}LineExtensionAmount').text
        lineItems.append({
            'id': itemID,
            'name': itemName,
            'quantity': itemQuantity,
            'price': itemPrice,
            'total': itemTotal
        })

    # Tạo đoạn mã HTML tương ứng
    htmlContent = f"""
        <h1>Receipt</h1>
        <p>Supplier: {supplierName}</p>
        <p>Customer: {customerName}</p>
        <table>
            <thead>
```

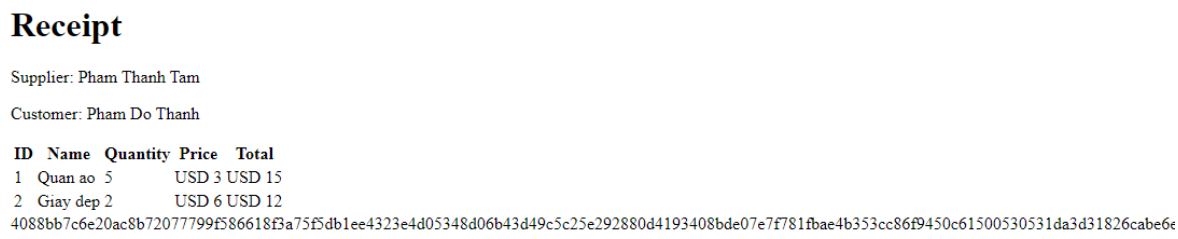
```
            <tr>
                <th>ID</th>
                <th>Name</th>
                <th>Quantity</th>
                <th>Price</th>
                <th>Total</th>
            </tr>
        </thead>
        <tbody>
'''

for item in lineItems:
    htmlContent += f'''
        <tr>
            <td>{item['id']}</td>
            <td>{item['name']}</td>
            <td>{item['quantity']}</td>
            <td>{item['price']}</td>
            <td>{item['total']}</td>
        </tr>
    '''

htmlContent += '''
        </tbody>
    </table>
'''

# Ghi nội dung HTML vào file receipt.html
```

```
    with open(html_file_path, 'w') as f:
        f.write(htmlContent)
```

The generate_receipt function reads an XML file containing invoice data, extracts the relevant information such as supplier name, customer name, and line items, generates an HTML receipt file with the extracted information, and saves it to the specified path.

The function uses namespaces to extract data from the XML file, including supplier and customer names, and line item details such as ID, name, quantity, price, and total. It then generates the HTML content for the receipt using this extracted data and saves it to a file at the specified path.

The resulting HTML file contains a table with the invoice data, including information about the supplier, customer, and line items.



Receipt

Supplier: Pham Thanh Tam

Customer: Pham Do Thanh

ID	Name	Quantity	Price	Total
1	Quan ao	5	USD 3	USD 15
2	Giay dep	2	USD 6	USD 12

4088bb7c6e20ac8b72077799f586618f3a75f5db1ee4323e4d05348d06b43d49c5c25e292880d4193408bde07e7f781fbae4b353cc86f9450c61500530531da3d31826cabe6e

Figure 7. RECEIPT WITH SIGNATURE

3.2.4 BANK

```
def verify(public_key_file, html_file):
    # Load the public key
    with open(public_key_file, 'r') as f:
        pk_hex = f.read()
    pk = bytearray.fromhex(pk_hex)

    # Load the HTML file and extract the signature value
    with open(html_file, 'r') as f:
        html_string = f.read()
```

```
soup = BeautifulSoup(html_string, 'html.parser')
signature_element = soup.find("signature")
while signature_element is not None:
    # Extract the signature value and remove the signature element
    signature_value = signature_element.string.strip()
    signature_element.extract()

    # Get the canonical representation of the HTML string
    canonical_html = str(soup)

    # Verify the signature using Dilithium
    try:
        is_valid = Dilithium2.verify(pk, canonical_html.encode(), bytes.fromhex(signature_value))
        if is_valid:
            print("The signature is valid.")
        else:
            print("The signature is not valid.")
    except Exception as e:
        print("An error occurred while verifying the signature:", str(e))

    # Find the next signature element
    signature_element = soup.find("signature")

# Write the updated HTML file
with open(html_file, 'w') as f:
    f.write(str(soup))
```

The verify function verifies the digital signature of an HTML file using the Dilithium2 algorithm and a public key, and removes the signature element from the HTML file if the signature is valid. If there are multiple signatures in the HTML file, the function verifies all of them.

The function loads the public key from a file and the HTML file from the specified path, and uses BeautifulSoup to parse the HTML string and extract the signature element(s). It iterates over each signature element in the HTML file, extracts the signature value, and verifies it using the Dilithium2 algorithm and the public key. If the signature is valid, the function removes the corresponding signature element from the HTML file. After all signature elements have been verified, the updated HTML file is written to the original file path.

The resulting HTML file has all signature elements removed if their corresponding signatures were valid.

3.2.5. DATABASE

Whenever a successful signature is applied to an invoice or receipt file, all of its data will be taken and stored on MongoDB along with the applied PK.

Figure 7. INVOICE WAS STORED IN MONGODB

Figure 8. RECEIPT WAS STORED IN MONGODB

4. **Demo:**

SELLER:

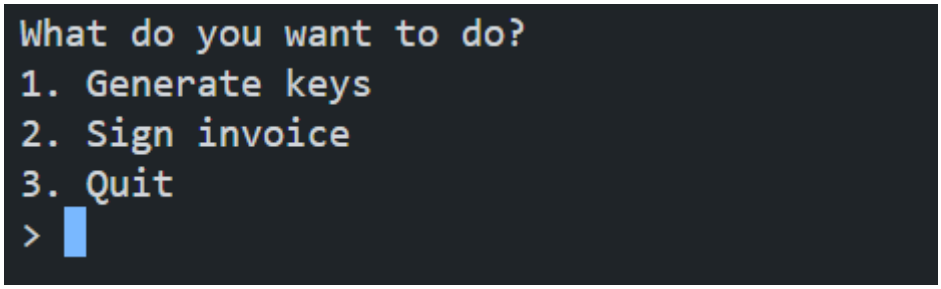


Figure 9. MENU

First, we will create a key by typing the command "1". Then it's gonna generate the keys.

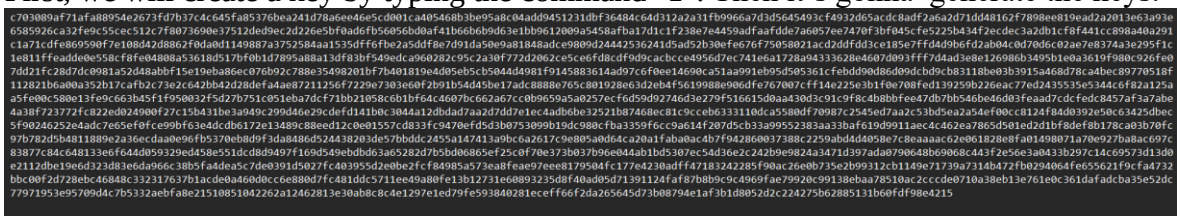


Figure 10. PUBLIC KEY FILE

After that, we will Sign the invoices by typing the command “2”. Then the invoice will be signed

Figure 11. THE INVOICE WITH SIGNATURE

BUYER:

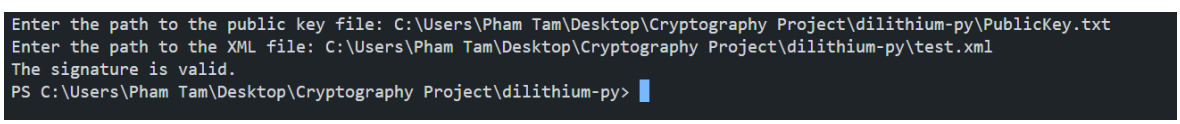


Figure 12. VERIFICATION BY BUYER

First, we need to write the path to the file containing the PublicKey.

Then, we need to write the path to the invoice file that have been signed by the Seller to verify. If the data in the Invoice file has not been modified, the verify function will return "The signature is valid" if the signature is valid and has been successfully verified using the public key. Otherwise, it will return "The signature is not valid" if the signature is invalid or has been tampered with.

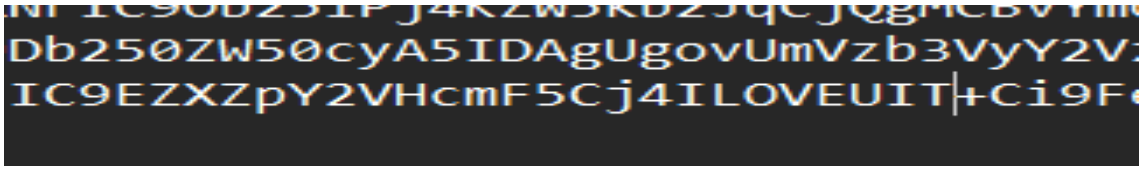


Figure 13. TAMPERED THE INVOICE

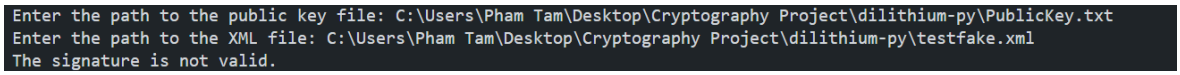


Figure 14. THE SIGNATURE INVALID

```
PS C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py> py CreateReceipt.py
Enter the XML File Path: C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py\test.xml
Enter the HTML File Path: C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py\test.html
File 'C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py\test.html' has been generated successfully.
PS C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py>
```

Figure 15. CREATING RECEIPT

```
What do you want to do?
1. Generate keys
2. Sign receipt
3. Quit
> 1
```

Figure 16. MENU RECEIPT

First, we will create a key by typing the command "1". Then it's gonna generate the keys.

After that, we will Sign the invoices by typing the command "2". Then the receipt will be signed

Receipt

Supplier: Pham Thanh Tam

Customer: Pham Do Thanh

ID	Name	Quantity	Price	Total
1	Quan ao	5	USD 3	USD 15
2	Giay dep	2	USD 6	USD 12

4088bb7c6e20ac8b72077799f586618f3a75f5db1ee4323e4d05348d06b43d49c5c25e292880d4193408bde07e7f781fbae4b353cc86f9450c61500530531da3d31826cabe6e

Figure 17. RECEIPT WITH SIGNATURE

BANK:

```
PS C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py> py Bank.py
Enter the path to the public key file: C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py\PublicKey.txt
Enter the path to the HTML file: C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py\test.html
The signature is valid.
PS C:\Users\Pham Tam\Desktop\Cryptography Project\dilithium-py>
```

Figure 18. BANK VERIFICATION

The Bank will verify the receipt have been sent by the Buyer. First, we need to write the path to the file containing the PublicKey.

Then, we need to write the path to the receipt file that have been signed by the Buyer to verify. If the data in the Invoice file has not been modified, the verify function will return "The signature is valid" if the signature is valid and has been successfully verified using the public key. Otherwise, it will return "The signature is not valid" if the signature is invalid or has been tampered with.

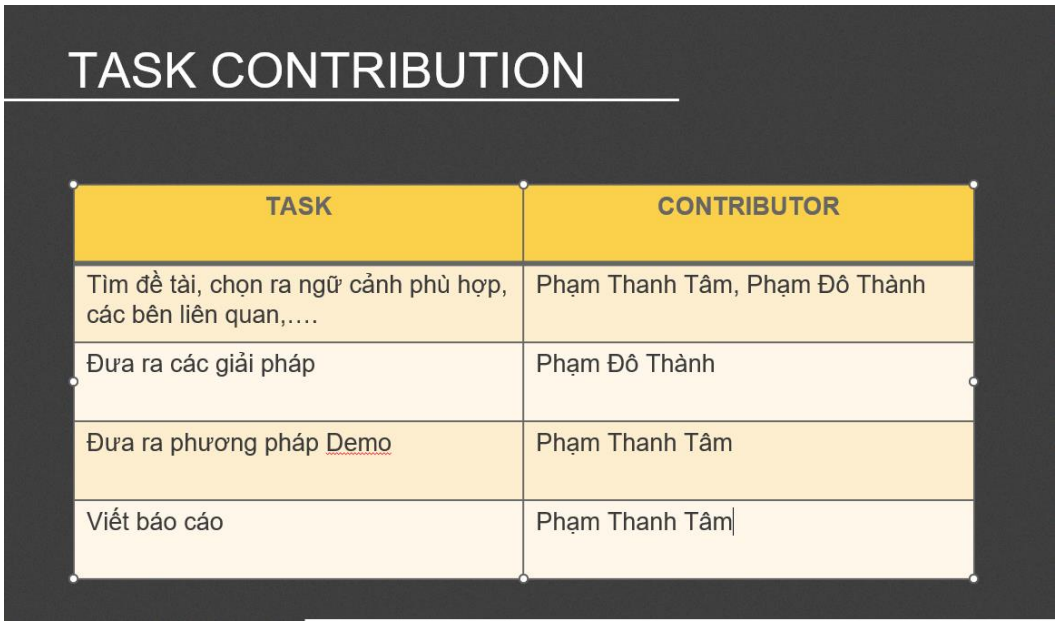


Figure 19. TASK CONTRIBUTION

5. **References:**

[1] Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., … & Bai, S. (2020). Crystal-dilithium. *Algorithm Specifications and Supporting Documentation.*