
Security of Computer Systems

Project Report

Authors:
Krzysztof, Madajczak, 188674
Piotr, Wesołowski, 188923

Version: 2.0

Versions

Version	Date	Description of changes
1.0	18.04.2024	Creation of the document
2.0	13.06.2024	Final adding the encryption and decryption

1. Project – control term

1.1 Description

We created the application in python which would help the user handling the private and signing documents

1.2 Results

For now our application has such functionality:

Generating and managing RSA keys:

- Generating a pair of RSA keys with a length of 2048 bits.
- Saving the private and public keys to files.
- Loading the private and public keys from files.

Encrypting and decrypting files:

- Encrypting files using the public key.
- Decrypting files using the private key.

Signing and verifying signatures:

- Signing documents using the private key.
- Verifying document signatures using the public key.

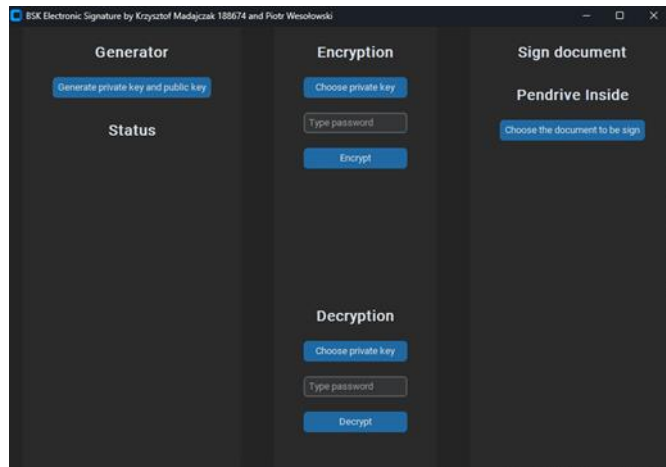
Creating XAdES signatures:

- Generating XAdES signatures for documents, including document information, signatures, and timestamps.

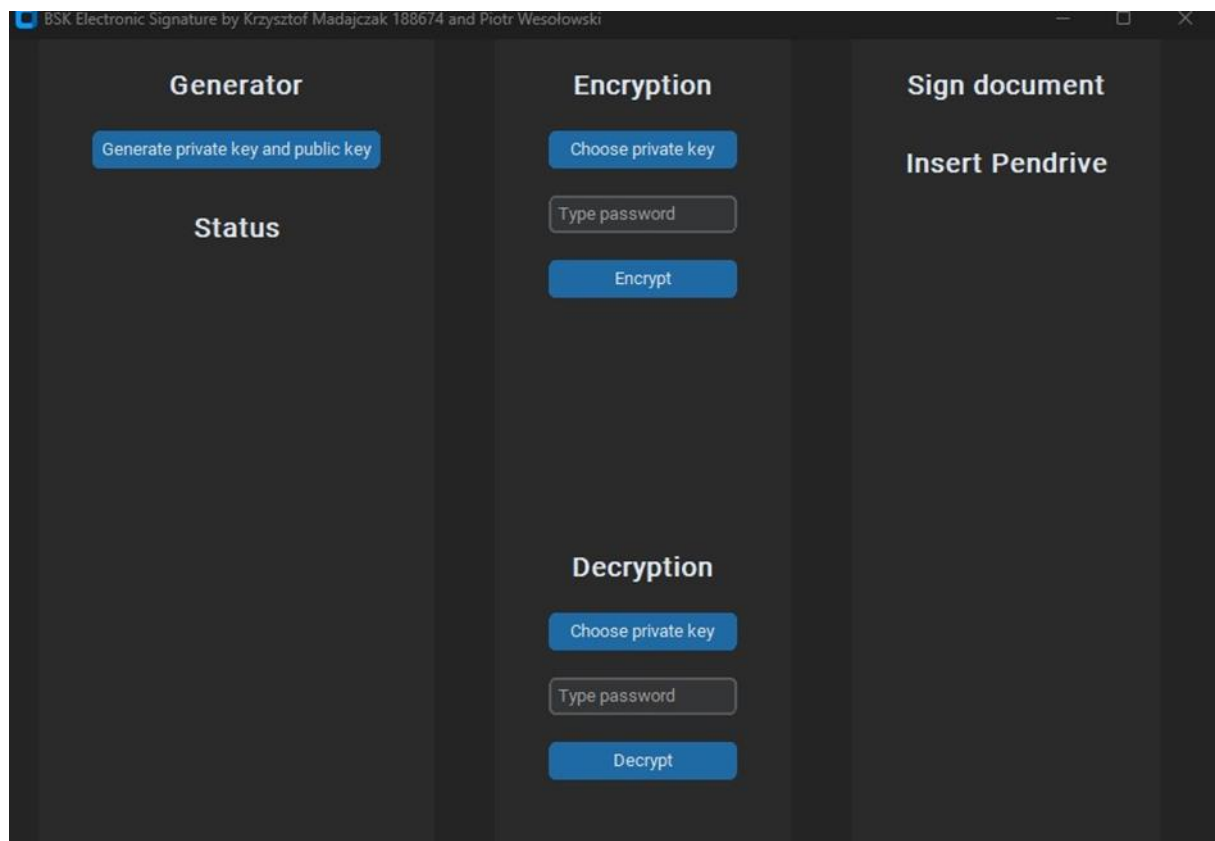
User interaction:

- Displaying a file selection dialog window.
- Providing success or error messages during operations.
- Providing status of USB stick.

example when pendrive with key is present.



When pendrive is not inside:



1.3 Summary

Our application implements various functionality but not all are implemented inside the GUI interface, we need to add some features such as password encryption and description. We have prototype version of such functions but they are not ready for production.

2. Project – Final term

2.1 Description

This Python application provides a comprehensive solution for secure file encryption, decryption, and digital signature verification using RSA encryption and SHA-256 hashing. The primary functionalities of this application include generating RSA key pairs, encrypting files with a recipient's public key, decrypting files with a private key, and digitally signing files to ensure integrity and authenticity.

Key Features:

RSA Key Pair Generation:

- The application generates a pair of RSA keys (private and public) with a key size of 4096 bits.
- The private key is encrypted with a user-defined password to enhance security.
- The generated keys are stored securely for future use.

File Encryption and Decryption:

- Users can encrypt files using the recipient's public key. This ensures that only the intended recipient, who possesses the corresponding private key, can decrypt and access the file.
- The recipient can decrypt the received file using their private key, which is protected by the password set during the key generation.

Digital Signing and Verification:

- The application can create a digital signature for a file by hashing the file content using SHA-256 and then encrypting the hash with the user's private key.
- The digital signature is stored in an XML file, ensuring it is easily accessible for verification purposes.
- Another user can verify the file's integrity by decrypting the digital signature with the owner's public key and comparing the resulting hash with the hash of the received file. If the hashes match, the file is confirmed to be unaltered.

2.2 Code Description

This code generates an RSA key pair consisting of a private key and a public key. The private key is created with a public exponent of 65537 and a size of 4096 bits, and then the corresponding public key is derived from the private key.

```
3 def generate_rsa_keypair():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=4096,
        backend=default_backend()
    )
    public_key = private_key.public_key()
    return private_key, public_key
```

Keys generating.

```
def encrypt_key(password, private_key):
    salt =
b'\x98A\xb9?J\xec\xe81v\x1c\xbb\xad\x1b\x85\x8a\x19\x89\x9d\x97t\xe6\xe7\
xc3\x03\t"ht\xdda\xde\xcc' # Można użyć get_random_bytes(32) w praktyce
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    key = kdf.derive(password.encode())
    cipher = AES.new(key, AES.MODE_CBC)
    private_key_bytes = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    )
    padded_private_key = pad(private_key_bytes, AES.block_size)
    cipher_data = cipher.encrypt(padded_private_key)
    hasz = hashlib.sha256(password.encode()).hexdigest()
    with open('keys/encrypted_private_key.bin', 'wb') as f:
        f.write(salt)
        f.write(hasz.encode())
        f.write(cipher.iv)
        f.write(cipher_data)
```

This code is used to encrypt the private key with a user's password. It uses KDF (PBKDF2HMAC) to generate a symmetric key from the password, and then encrypts the private key using the AES algorithm in CBC mode, saving the encrypted data along with the salt, password hash, and initialization vector to a binary file.

Encrypting private key.

3 – decrypting private key

```
def decrypt_key(password, path= 'keys'):
    try:
        with open(path, 'rb') as f:
            salt = f.read(32)
            hasz = f.read(64)
            iv = f.read(16)
            cipher_data = f.read()
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,
            backend=default_backend()
        )
        key = kdf.derive(password.encode())
        cipher = AES.new(key, AES.MODE_CBC, iv)
        decrypted_padded_private_key = cipher.decrypt(cipher_data)
        private_key_bytes = unpad(decrypted_padded_private_key,
AES.block_size)
        private_key = serialization.load_pem_private_key(
            private_key_bytes,
            password=None,
            backend=default_backend()
        )
        return private_key, hasz
    except Exception as e:
        print(f"Password wrong")
        return False, False
```

1. Reading the encrypted private key, salt, password hash, and initialization vector (IV) from the file.
2. Generating a symmetric key from the password using KDF (PBKDF2HMAC).
3. Decrypting the private key using the AES algorithm in CBC mode and deserializing it.

Detecting USB

```
def start_usb_detection_thread():
    detection_thread = threading.Thread(target=usb_detection_loop,
    daemon=True)
    detection_thread.start()
def usb_detection_loop():
    while True:
        usb_path = detect_usb()
        if usb_path and usb_path != "False":
            globals.usb_path_queue.put(usb_path)
            globals.usb_detected_event.set()
            time.sleep(5) # Check every 5 seconds

def detect_usb():
    for partition in psutil.disk_partitions():
        if 'removable' in partition.opts:
            removable_disk = partition.mountpoint
            if not globals.foundUSB:
                print(f"Found USB drive at {removable_disk} (Name:
{partition.device})")
                key_file_path = os.path.join(removable_disk,
'encrypted_private_key.bin')
                if os.path.exists(key_file_path):
                    if not globals.foundUSB:
                        print("Found key.pem file on the removable disk.")
                        globals.foundUSB = True
                        globals.path_private_key = key_file_path
                        return key_file_path
                else:
                    globals.foundUSB = False
                    print("key.pem file not found on the removable disk.")

    globals.foundUSB = False
    return "False"
```

To accomplish this task, we created a thread that checks for connected devices and updates the UI with new functionality if a device containing the encrypted private key is detected.

This code starts a thread that detects connected USB devices and checks if they contain an encrypted private key file. The `start_usb_detection_thread()` function initializes and starts the thread, which runs the `usb_detection_loop()` function, checking every 5 seconds for connected USB devices. The `detect_usb()` function searches mounted partitions for removable drives and checks if they contain the `encrypted_private_key.bin` file. When such a file is found, its path is added to the global queue `usb_path_queue`, and the event `usb_detected_event` is set, signaling that a USB device with the required file has been detected.

Encryption of the file:

```
def encrypt_file():
    if globals.public_key is None:
        print("YOU MUST SELECT PUBLIC KEY TO ENCRYPT")
        return
    print("SZYFROWANIE")
    file_path = filedialog.askopenfilename()
    with open(file_path, 'rb') as f:
        plaintext = f.read()
    ciphertext = globals.public_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    directory, filename = os.path.split(file_path)
    new_filename = f"szyfr_{filename}"
    new_file_path = os.path.join(directory, new_filename)

    with open(new_file_path, 'wb') as f:
        f.write(ciphertext)
```

Check for Public Key: The function first checks if a public key is available in the global scope. If not, it prints a message indicating that a public key is required for encryption and exits the function.

Read Plaintext File: It prompts the user to select a file using a file dialog. It then reads the contents of this file in binary mode.

Encrypt File Content: The function encrypts the read content using the public key with OAEP padding (using SHA-256 as the hashing algorithm).

Save Encrypted File: It constructs a new file name by prefixing "szyfr_" to the original filename, then writes the encrypted content to this new file in the same directory as the original file.

Decryption of the file:

```
def decrypt_file():
    if globals.correctPassword:
        print("Odszyfrowywanie")
        private_key, hasz = decrypt_key(password = globals.key_password,
```



```

path = globals.path_private_key)
    if not hasz:
        return
    file_path = filedialog.askopenfilename()

    with open(file_path, 'rb') as f:
        ciphertext = f.read()

    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    directory, filename = os.path.split(file_path)
    new_filename = f"odszy_{filename}"
    new_file_path = os.path.join(directory, new_filename)
    print(f"Odszyfrowana wiadomość {plaintext}")
    with open(new_file_path, 'wb') as f:
        f.write(plaintext)
else:
    print("Złe hasło")

```

- **Verify Password:** The function checks if the global flag correctPassword is set to True. If not, it prints "Złe hasło" ("Wrong password") and exits the function.
- **Decrypt Private Key:** If the password is correct, it prints "Odszyfrowywanie" ("Decrypting") and retrieves the private key by calling decrypt_key with the global password and private key path. If the hash verification fails, it exits the function.
- **Read Encrypted File:** The user is prompted to select a file using a file dialog. The function reads the contents of this selected file in binary mode.
- **Decrypt File Content:** The function decrypts the read content using the private key with OAEP padding (using SHA-256 as the hashing algorithm), prints the decrypted plaintext, and saves it to a new file with the prefix "odszy_" in the same directory as the original file.

Signing the document:

```

def create_xades_signature(document_path, private_key):
    try:
        # Pobierz informacje o dokumencie
        document_size = os.path.getsize(document_path)
        document_extension = os.path.splitext(document_path)[-1]
        modification_date =
datetime.fromtimestamp(os.path.getmtime(document_path)).isoformat()

```

```

        # Podpisz dokument
        with open(document_path, "rb") as file:
            document_content = file.read()

        document_hash = hashes.Hash(hashes.SHA256(),
backend=default_backend())
        document_hash.update(document_content)
        calculated_digest = document_hash.finalize()

        print(f"Calculated digest (signing): {calculated_digest.hex()}")
# Debugowanie

        signature = sign_document(calculated_digest, private_key)
        print(f"Signature: {base64.b64encode(signature).decode()}") #
Debugowanie

        root = ET.Element("XAdES_Signature")
        document_info = ET.SubElement(root, "DocumentInfo")
        document_info.set("size", str(document_size))
        document_info.set("extension", document_extension)
        document_info.set("modification_date", modification_date)
        ET.SubElement(root, "SigningUserInfo", name="User A")
        ET.SubElement(root, "Timestamp").text =
datetime.now().isoformat()
        ET.SubElement(root, "EncryptedHash").text =
base64.b64encode(signature).decode()

        xml_tree = ET.ElementTree(root)
        directory = os.path.dirname(document_path)
        temp_path = os.path.join(directory, 'signature.xml')
        temp_path = os.path.normpath(temp_path)
        xml_tree.write(temp_path)
        print("Signature xml written to {}".format(temp_path))
        return signature

    except Exception as e:
        print("Error creating XAdES signature:", e)
        return None

```

- **Retrieve Document Information:** The function obtains the size, extension, and last modification date of the specified document. It reads the document content in binary mode.
- **Calculate Document Hash:** It computes the SHA-256 hash of the document content and prints the calculated digest for debugging purposes.
- **Sign the Document:** The function signs the calculated digest using the provided private key, prints the base64-encoded signature for debugging purposes, and constructs an XML structure for the XAdES signature. This structure includes document information, signing user information, a timestamp, and the encrypted hash.
- **Save Signature to XML:** The function saves the XML structure containing the XAdES signature to a file named `signature.xml` in the same directory as the document. It prints the file path of the saved XML and returns the signature. If any error occurs during the process, it catches the exception, prints an error message, and returns `None`.

```

def verify_signature(document_path, signature_xml_path, public_key):
    if public_key is None:
        print("Public key is None")
        return
    try:
        document_size = os.path.getsize(document_path)
        document_extension = os.path.splitext(document_path)[-1]
        modification_date =
datetime.fromtimestamp(os.path.getmtime(document_path)).isoformat()
        with open(document_path, "rb") as file:
            document_content = file.read()
            document_hash = hashes.Hash(hashes.SHA256(),
backend=default_backend())
            document_hash.update(document_content)
            calculated_digest = document_hash.finalize()

        print(f"Calculated digest (verification):
{calculated_digest.hex()}")

        tree = ET.parse(signature_xml_path)
        root = tree.getroot()
        signature_base64 = root.find("EncryptedHash").text
        signature = base64.b64decode(signature_base64)
        doc_info = root.find("DocumentInfo")
        if (doc_info.get("size") != str(document_size) or
            doc_info.get("extension") != document_extension or
            doc_info.get("modification_date") != modification_date):
            print(f"Expected size: {document_size}, extension:
{document_extension}, modification_date: {modification_date}")
            print(f"Found size: {doc_info.get('size')}, extension:
{doc_info.get('extension')}, modification_date:
{doc_info.get('modification_date')}")
            raise ValueError("Document information mismatch.")
        public_key.verify(
            signature,
            calculated_digest,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        signing_user_info = root.find("SigningUserInfo")
        if signing_user_info is None or signing_user_info.get("name") !=
"User A":
            raise ValueError("Invalid signing user information.")

        # Odczytaj znacznik czasu
        timestamp = root.find("Timestamp").text
        print(f"Signature timestamp: {timestamp}")

        print("Signature verified successfully.")
    except Exception as e:
        print("Signature verification failed:", e)

```

- **Check for Public Key:** The function starts by checking if the provided public key is None. If it is, it prints a message indicating the absence of a public key and exits the function.
- **Retrieve Document Information and Calculate Hash:** It obtains the document's size, extension, and last modification date. It reads the document

content in binary mode and computes its SHA-256 hash, printing the calculated digest for debugging purposes.

- **Parse Signature XML and Validate Document Information:** The function parses the signature XML file to retrieve the base64-encoded signature and decodes it. It also extracts the document information from the XML and compares it with the actual document information. If there is a mismatch, it prints the expected and found document information and raises a `ValueError`.
- **Verify Signature:** It uses the public key to verify the signature against the calculated document digest with PSS padding and SHA-256 hash. It checks the signing user information in the XML, ensuring it matches the expected value ("User A"). It then reads and prints the signature timestamp, and if all verifications pass, it prints a success message. If any step fails, it catches the exception and prints an error message indicating signature verification failure.





2.3 Results

Our program is divided into three sections:

First section is responsible for the generation of an encrypted private key and its corresponding public key. A password is required to perform this task. Depending on the action, we receive feedback through a status label: "Password required" if a password is not provided, and "Keys Generated" if the operation is successful.

<div><h3>Generator - ETAP 1</h3><input type="text" value="Enter PIN"/> <input type="button" value="Generate private key and public key"/> Status</div>	<div><h3>Generator - ETAP 1</h3><input type="text" value="Enter PIN"/> <input type="button" value="Generate private key and public key"/> password required</div>
<div><h3>Generator - ETAP 1</h3><input type="text" value="12345"/> <input type="button" value="Generate private key and public key"/> Keys Generated</div>	

Second section is responsible for the encrypting file using public key and checking signature of the document also with using of that public key. After performing that action we get the name of the file with the prefix "szyfr_"

	szyfr_tekst.txt	13/06/2024 13:47	Dokument tekstowy	1 KB
	szyfr_test.txt	13/06/2024 23:25	Dokument tekstowy	1 KB
	tekst.txt	12/06/2024 23:54	Dokument tekstowy	1 KB
	test.txt	13/06/2024 13:49	Dokument tekstowy	1 KB

```
z0C ,.ö~Žác`Lł-"óÄ| Ät~Ž°@@,,k...ýI'g"ö$Sk ,!~++Éhc! 1Ö*2Ń.ĐĐnmHÚIjtösÄŘ  
šš-1ÜÜiÜ@A$μBŃKZK@ER@ÉÄéÄ @{(ž@Ž""ŃLŃŃEÚp ,_ÎNs6e{Í3'»Ä@ý@É@@@I+E@-gÜ]ö@SLhUT]ä{5gHú@*1Uťćo-N%+ň`@  
#"CdIö;KμN#jôžD@!"8@Eñ^@a/ůTŁ~žŃ'b'1_Ń@,/,'xñG'`@15=h«ú k[Š e öědu·4PKtC·a!aß@ XOf1$š@>á.  
e9E,5%€÷k.°ÝrTđÍ@aUy€đf@,Á@84÷"»^Íf'ÍÍú O'@`3tĚ·SŃh'n°ç[Šo/öT+`m†@'š.( 'ě@iKšİž" a...$TđSQéć~ n  
@8ě%đě -ÇL@5r@@@'@@éi@@čoEtúŃ{Ý°@[4†Ńn»Ě@
```


The second functionality is possibility of checking the signature of the sign document to accomplish that we must choose the public key that the signature is made of the and original document and the .xml containing signature information.

```
C:/Studia/Semestr6/Aplikacje/BSK/BSK-Electronic-Signature/krzysztof/test/test.txt
C:/Studia/Semestr6/Aplikacje/BSK/BSK-Electronic-Signature/krzysztof/test/signature.xml
Calculated digest (verification): a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3
Signature timestamp: 2024-06-13T23:48:49.015933
Signature verified successfully.
```

The screenshot shows a web application interface with a dark background. At the top, the text "check Sign/Encryption" is displayed in a light blue font. Below this, the word "Encryption" is centered in a larger, bold, light blue font. Under "Encryption", there are two blue buttons: "public key" and "Choose the document to be Encrypted". Further down, the text "Check signature" is centered in a bold, light blue font. Below this, there are three blue buttons: "choose document", "choose XML", and "Check signature".


The third section is responsible for signing the document and describing its contents. To sign the document, we must enter the password. If everything goes well, we receive the corresponding label "Signature Complete". Subsequently, a new file named "signature.xml" is created in the chosen folder path, which contains:

- General identification of the document (size, extension, date of modification).
- Information about the signing user.
- Encrypted by RSA private key hash of the document.
- Timestamp of the signature (local time of the user A).

 signature.xml	13/06/2024 23:35	Plik XML	1 KB
---	------------------	----------	------

Signature: eu/Jn6tngwBu16zA4Xf66Kej7xMHR1mV/0W7DISE6WyXosRHf309nCtEEq6Kc6YJXVF3UrtPH7TKzH0EUSmWvV7Locwe3HNisMNq6344
Signature xml written to C:\Studia\Semestr6\Aplikacje\BSK\BSK-Electronic-Signature\krzysztof\test\signature.xml

The next functionality involves decrypting a file using a private key. The user selects the encrypted file and provides a password. After successful decryption, the decrypted file is prefixed with "odszy_".

Nazwa	Data modyfikacji	Typ	Rozmiar
 odszy_szyfr_tekst.txt	13/06/2024 13:47	Dokument tekstowy	1 KB

Sign/Decryption

Decryption

Choose the document to be Decrypted

Sign

Enter PIN

Signature complete

check password

Choose the document to be sign

E:\encrypted_private_key.bin

2.4 Summary

This Python project is a robust application designed for secure file encryption, decryption, and digital signature verification. It leverages RSA encryption with a key size of 4096 bits and SHA-256 for hashing to ensure data security and integrity. Key functionalities include:

- **RSA Key Pair Generation:** Users can generate RSA key pairs, with the private key encrypted using a user-defined password for added security.
- **File Encryption:** Files can be encrypted using the recipient's public key, ensuring only the intended recipient can decrypt and access the contents.
- **File Decryption:** Recipients can decrypt encrypted files using their private key, which is protected by a password.
- **Digital Signing:** Files can be digitally signed by creating a SHA-256 hash of the file content and encrypting the hash with the user's private key. The digital signature is stored in an XML file.
- **Signature Verification:** Recipients can verify the integrity and authenticity of a signed file by comparing the decrypted hash from the digital signature with a newly generated hash of the received file.

This application provides a comprehensive solution for secure file handling, ensuring confidentiality, authenticity, and integrity of data during transmission and storage.

3. Literature

[1] Cryptography Hazmat Documentation

<https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/>

[2] XML Handling in Python:

<https://docs.python.org/3/library/xml.etree.elementtree.html>

[3] Password-based Encryption:

<https://en.wikipedia.org/wiki/PBKDF2>

[4] RSA Algorithm:

[https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

[5] PyCryptodome Documentation :

https://pycryptodome.readthedocs.io/en/latest/src/public_key/public_key.html

[6] Tkinter Filedialog Documentation:

<https://docs.python.org/3/library/tkinter.filedialog.html>