Security of Computer Systems

Project Report

Authors:

Krystian Przybysz 188918

Nikodem Keller 184853

1. Projekt - final term

1.1 Destription

This project created an advanced software tool that mimics qualified electronic signatures and performs essential encryption functions in accordance with the XAdES standard. It smoothly integrates with a hardware device (pendrive) to securely store encrypted RSA keys. The application's graphical user interface (GUI) enables users to easily select documents for digital signing and encryption, ensuring strong security by encrypting the RSA private key with AES, connected to the user's PIN.

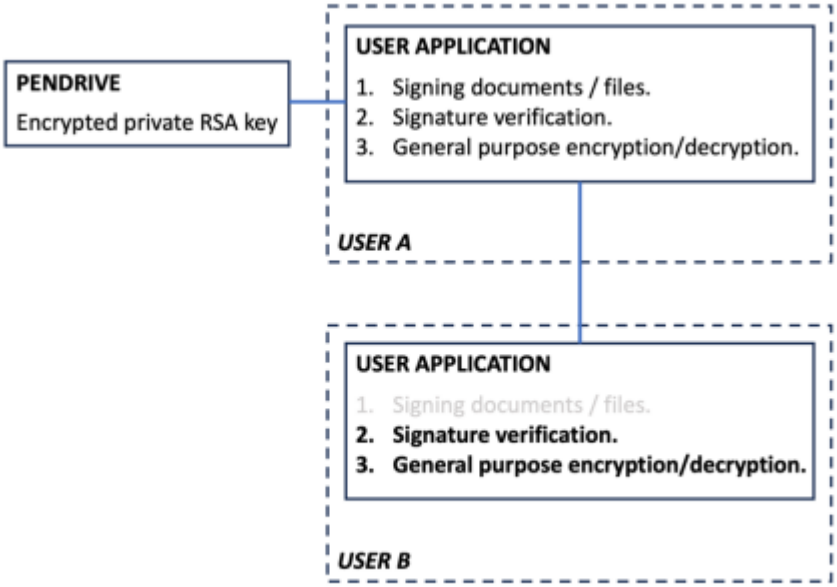


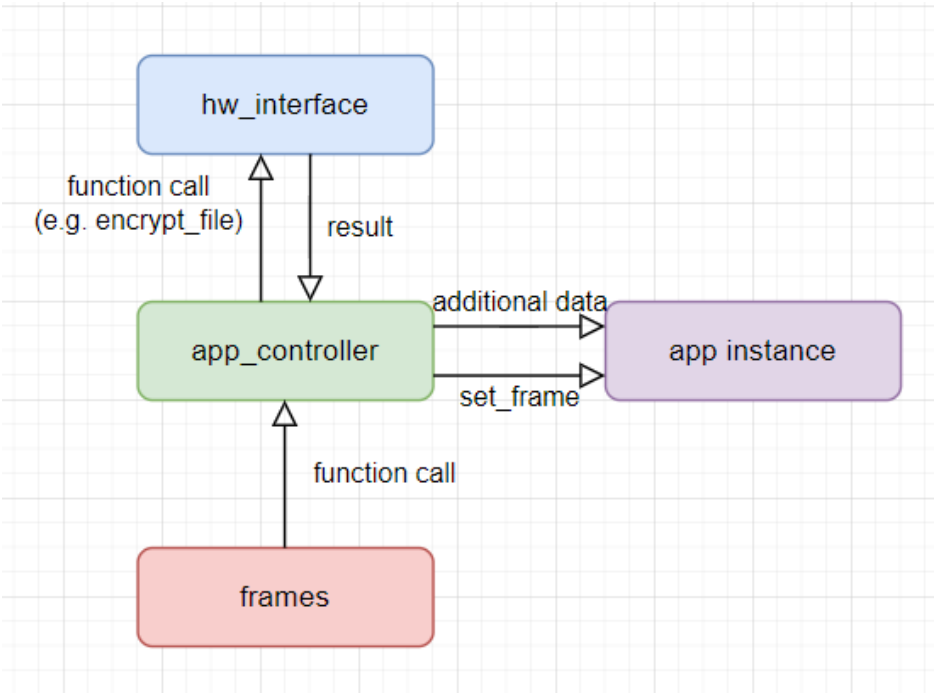
Fig. 1 – Block diagram of the project concept.

Whole project was created before control term. The project is coded in python using few specific libraries:

- Cryptodome
- Hashlib
- Customtkinter

The allowed file extensions for encrypting/decrypting and signing are:

- .html
- .txt



Application flow

1.2 Code Description

The project comprises several modules and files that collaborate to enable secure digital signing and file encryption. **app_storage** defines global variables and functions to manage keys and file paths within the application. **app_controller** oversees the application logic, handling tasks such as file signing, encryption, file selection for verification, and management of public and private keys. **hw_interface** facilitates interaction with external devices and implements cryptographic operations including file encryption, decryption, signing, and signature verification. **Frames** defines GUI interfaces of the application, allowing users to conveniently browse and select files for cryptographic operations, integrated with the application controller.

1.2.1 Code highlights

```python
def set_frame(self, frame, *args, **kwargs):
    """
    Sets the current frame of the application.

    Args:
        frame: The frame to set.
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.
    """
    if self.frame is not None:
        self.frame.pack_forget()

    self.frame = frame(self, self, *args, **kwargs)
```

set_frame function for setting the current frame in Customtkinter application

```python
def encrypt_file_click(app, file_path, public_key_file_path):
    """
    Handles the event when the 'Encrypt File' button is clicked.

    Args:
        app: The application instance.
        file_path (str): The path to the file to encrypt.
        public_key_file_path (str): The path to the public RSA key.
    """
    try:
        encrypted_file_path = hw_interface.encrypt_file(file_path, public_key_file_path)
        if encrypted_file_path != '':
            app.set_frame(ShowResultFrame, result="File encrypted successfully",
                          success=True, path=encrypted_file_path)
        else:
            app.set_frame(ShowResultFrame, result="Error while encrypting the file",
                          success=False)
    except Exception as e:
        print(f"An error occurred while encrypting the file: {str(e)}")
```

"Encrypt File" button event handler in app_controller using "set_frame" function to change between frames in Customtkinter

```
encrypt_button = ctk.CTkButton(self, text="Encrypt file", font=("Courier new", 20),
                               width=app.width * 0.5,
                               height=app.height * 0.1,
                               command=lambda: app_controller.encrypt_file_click(app,
                                                                                 file_path=file_path,
                                                                                 public_key_file_path=public_key_file_path))
```

Calling the ”encrypt_file_click” function in the” frames” module

```
def encrypt_file(file_path, public_key_path):
    """
    Encrypts a file with a given public RSA key.

    Args:
        file_path (str): The path to the file to encrypt.
        public_key_path (str): The path to the public RSA key to use for encryption.

    Returns:
        bytes: The encrypted file content. None if an error occurred during encryption.
    """
    try:
        with open(file_path, "rb") as file:
            file_content = file.read()

        with open(public_key_path, "rb") as key_file:
            public_key = RSA.import_key(key_file.read())

        cipher = PKCS1_OAEP.new(public_key)
        encrypted_file = cipher.encrypt(file_content)

        with open(file_path, "wb") as file:
            file.write(encrypted_file)

        return encrypted_file

    except (ValueError, KeyError) as e:
        print(f"An error occurred during decryption: {e}")
        return None
```

Function for file encryption in hw_interface, PKCS#1 OAEP is an asymmetric cipher based on RSA and the OAEP padding. It is described in RFC8017

```python
def decrypt_file(file_path, decrypt_key):
    """
    Decrypts a file with a given RSA key.

    Args:
        file_path (str): The path to the file to decrypt.
        decrypt_key (RSA key): The RSA key to use for decryption.

    Returns:
        bytes: The decrypted file content. None if an error occurred during decryption.
    """
    try:
        with open(file_path, "rb") as file:
            encrypted_file = file.read()

        cipher = PKCS1_OAEP.new(decrypt_key)
        decrypted_file = cipher.decrypt(encrypted_file)

        with open(file_path, "wb") as file:
            file.write(decrypted_file)

        return decrypted_file

    except (ValueError, KeyError) as e:
        print(f"An error occurred during encryption: {e}")
        return None
```

Function for file decryption in hw_interface, PKCS#1 OAEP is an asymmetric cipher based on RSA and the OAEP padding. It is described in RFC8017

```python
def detect_external_devices():
    """
    Detects all external devices connected to the system.

    Returns:
        list: A list of paths to the detected external devices.
    """
    drives = win32api.GetLogicalDriveStrings()
    drives = drives.split('\000')[:-1]
    external_devices = []
    for drive in drives:
        drive_type = win32file.GetDriveType(drive)
        if drive_type == win32file.DRIVE_REMOVABLE:
            external_devices.append(drive)
    return external_devices
```

```python
def find_pem_files(drives):
    """
    Finds all .pem files in the given drives.

    Args:
        drives (list): A list of paths to the drives to search in.

    Returns:
        list: A list of paths to the found .pem files. None if no .pem files were found.
    """
    pem_files = []
    for drive in drives:
        for root, dirs, files in os.walk(drive):
            for file in files:
                if file.endswith(".pem"):
                    pem_files.append(os.path.join(root, file))
    if not pem_files:
        return None
    return pem_files
```

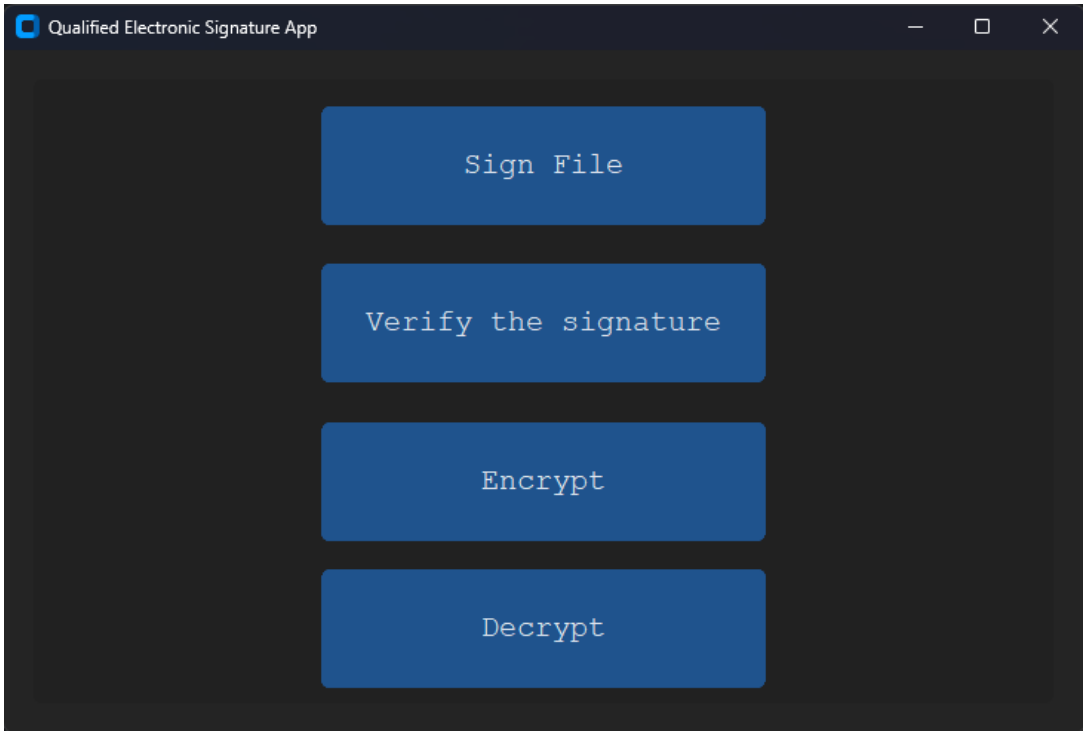
Function for finding all .pem files in external devices

```python
def sign_file(file_path, key):
    """
    Signs a file with a given RSA key.

    Args:
        file_path (str): The path to the file to sign.
        key (RSA key): The RSA key to sign the file with.

    Returns:
        tuple: A tuple containing the XML signature as a string and the path to the saved signature file.
    """
    with open(file_path, "rb") as file:
        file_content = file.read()

    file_hash = SHA256.new(file_content)

    signer = PKCS1_v1_5.new(key)
    signature = signer.sign(file_hash)

    signature_xml = ET.Element("XAdES")

    file_info = ET.SubElement(signature_xml, tag: "FileInfo")
    file_info_dict = {
        "Size": str(os.path.getsize(file_path)),
        "Extension": os.path.splitext(file_path)[1][1:],  # remove leading dot
        "ModificationDate": str(datetime.fromtimestamp(os.path.getmtime(file_path)))
    }
    for key, value in file_info_dict.items():
        ET.SubElement(file_info, key).text = value

    user_info = ET.SubElement(file_info, tag: "UserInfo")
    user_info_dict = {
        "SigningUserName": str(os.getlogin()),
        "SigningUserMAC": str(hex(get_mac()))
    }
    for key, value in user_info_dict.items():
        ET.SubElement(user_info, key).text = value

    ET.SubElement(signature_xml, tag: "EncryptedHash").text = base64.b64encode(signature).decode()
    ET.SubElement(signature_xml, tag: "Timestamp").text = str(datetime.now())

    signature_xml_str = parseString(ET.tostring(signature_xml)).toprettyxml()

    signature_file_name = f"signature-{os.path.splitext(os.path.basename(file_path))[0]}.xml"
    with open(signature_file_name, "w") as xml_file:
        xml_file.write(signature_xml_str)

    return signature_xml_str, signature_file_name
```

Function for signing a file with a given RSA key

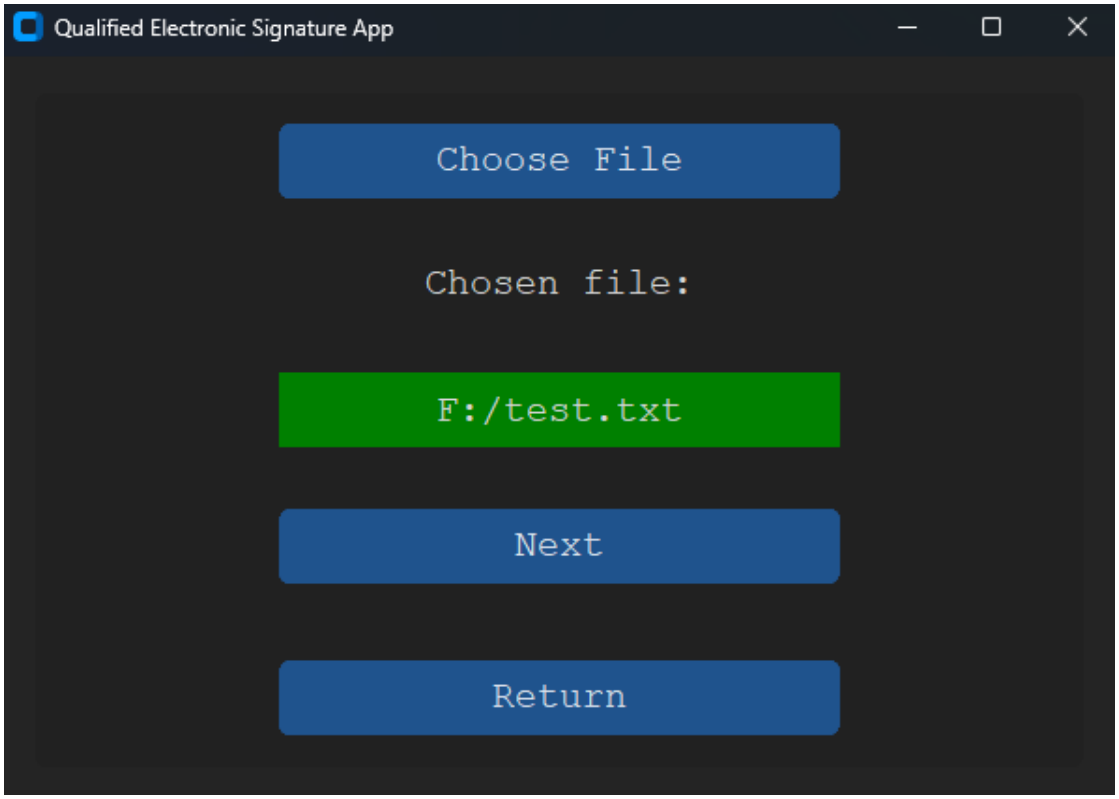
1.3 GUI –walktrough

1.3.1 Main Menu

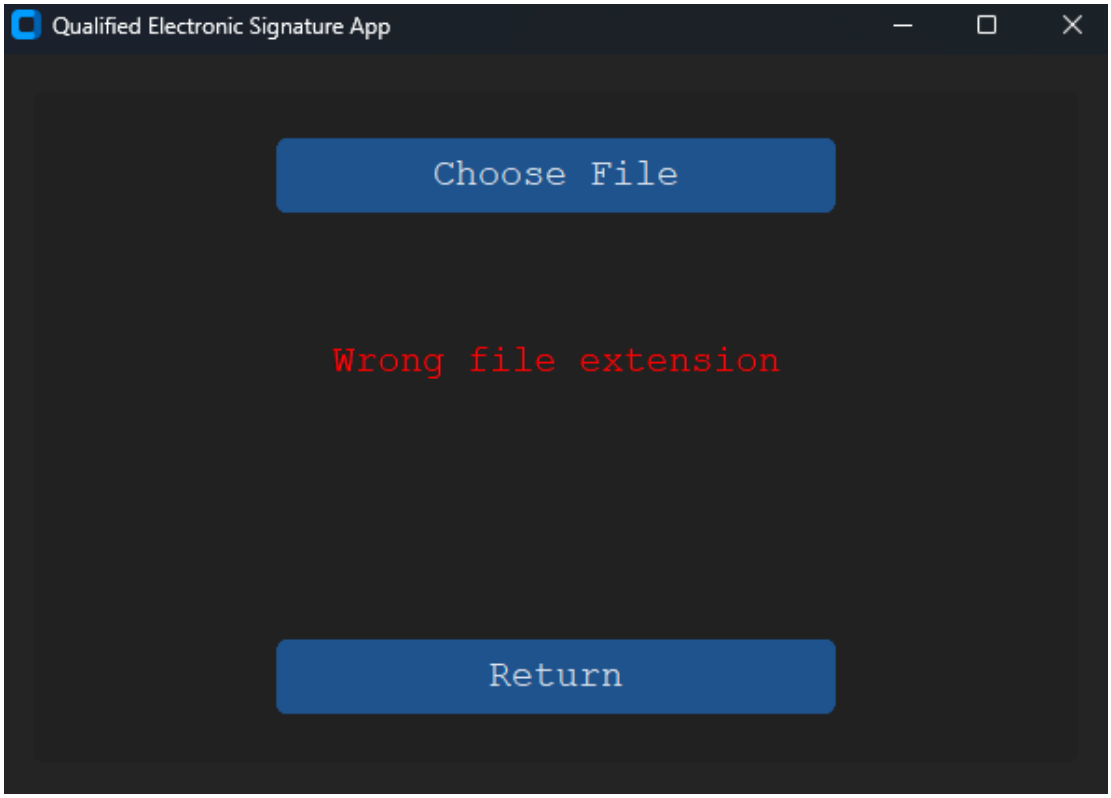


Main menu of application

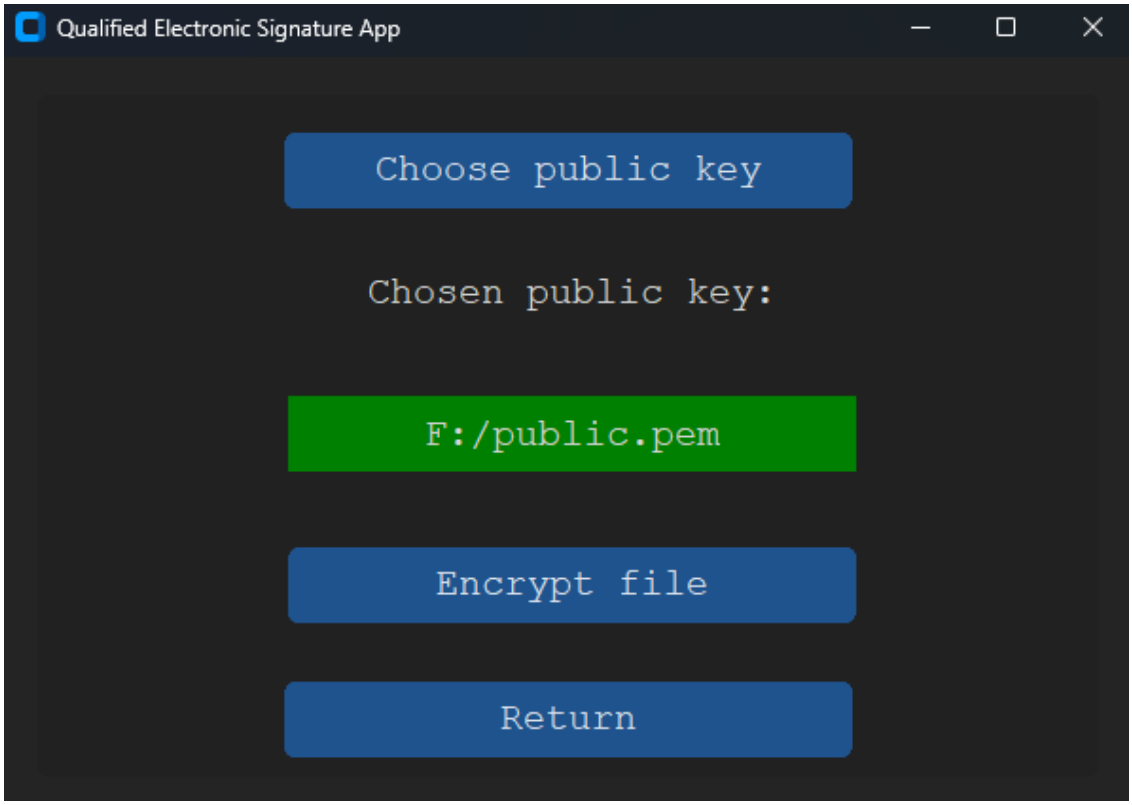
1.3.2 General purpose encryption



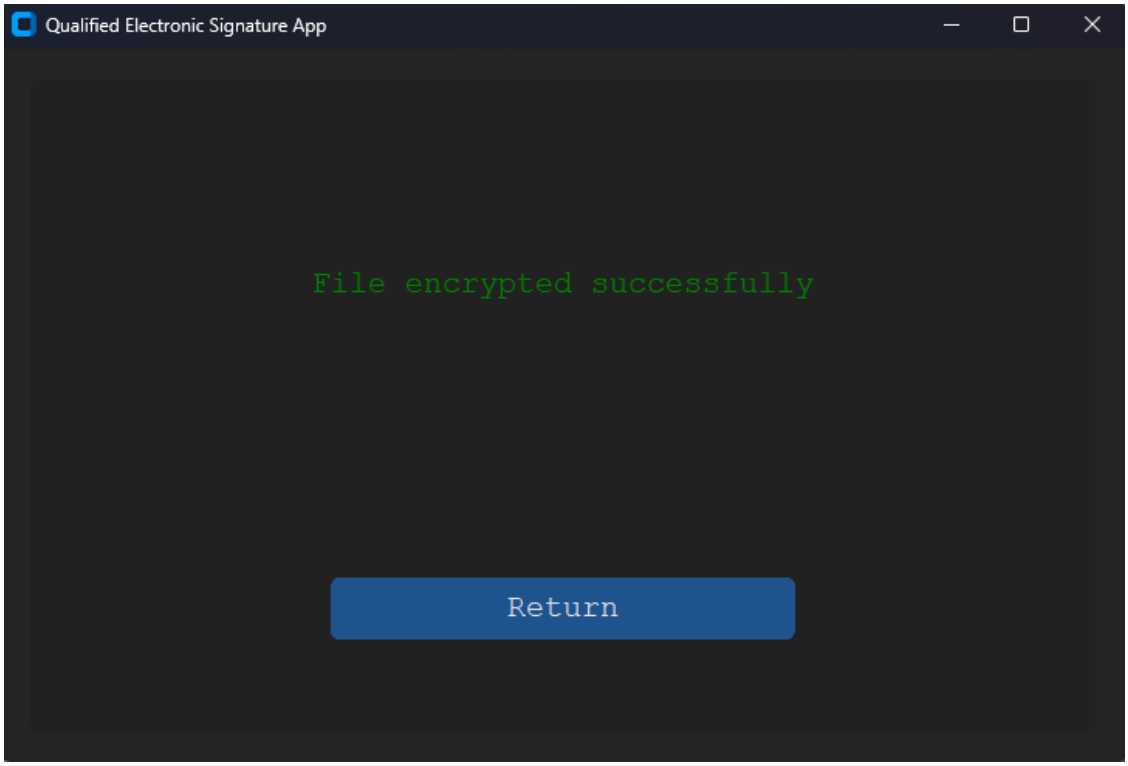
Encrypt Page 1 – chosen file with valid extension



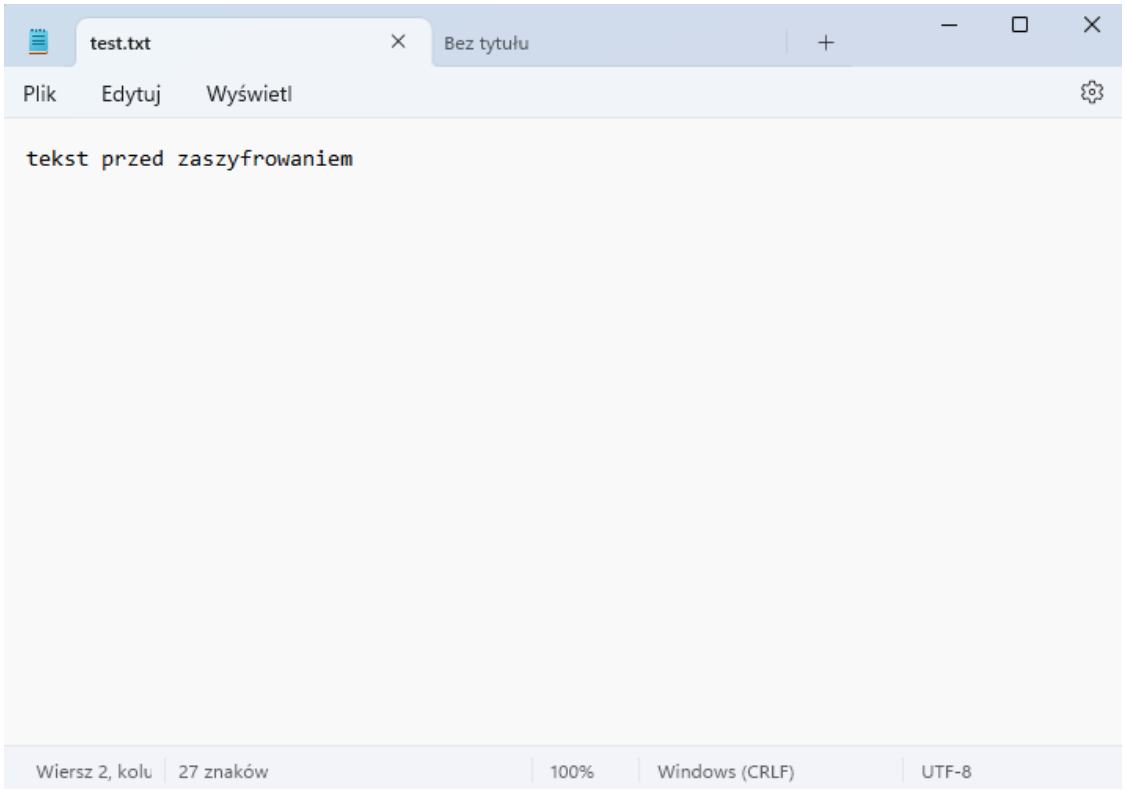
Encrypt Page 1 – chosen file with wrong file extension



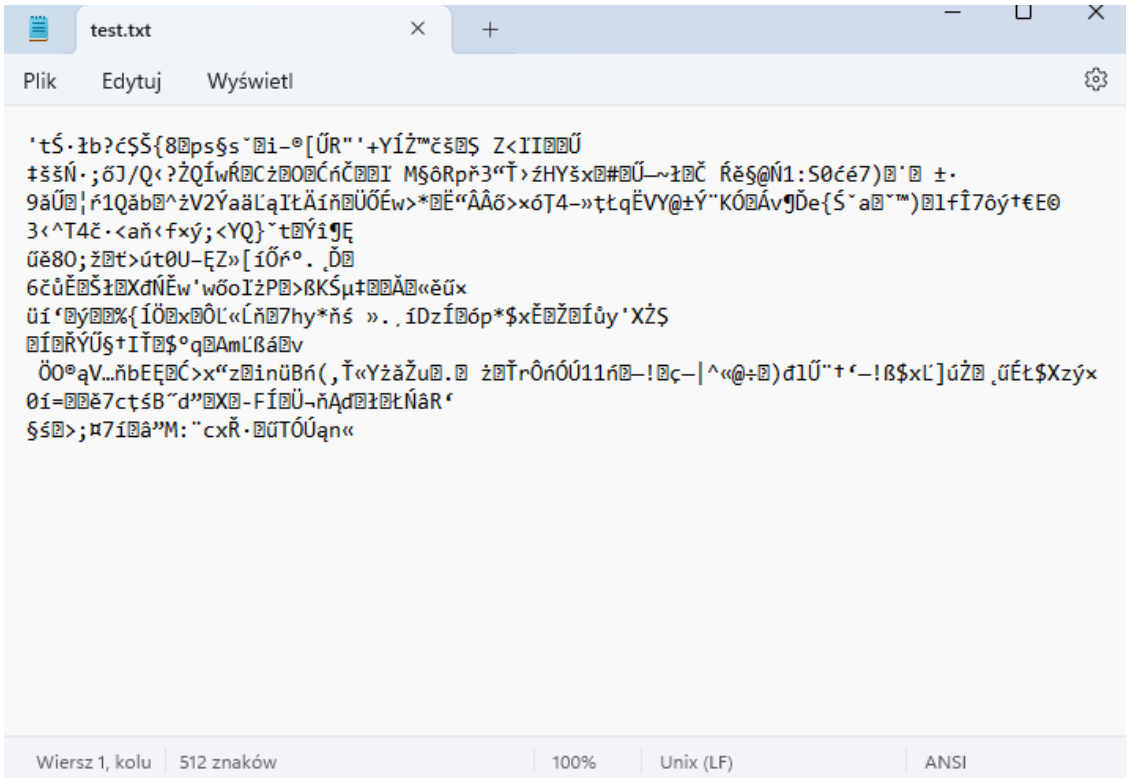
Encrypt Page 2 –chosen public key with valid extension (.pem)



Encrypt Page 3 - succes

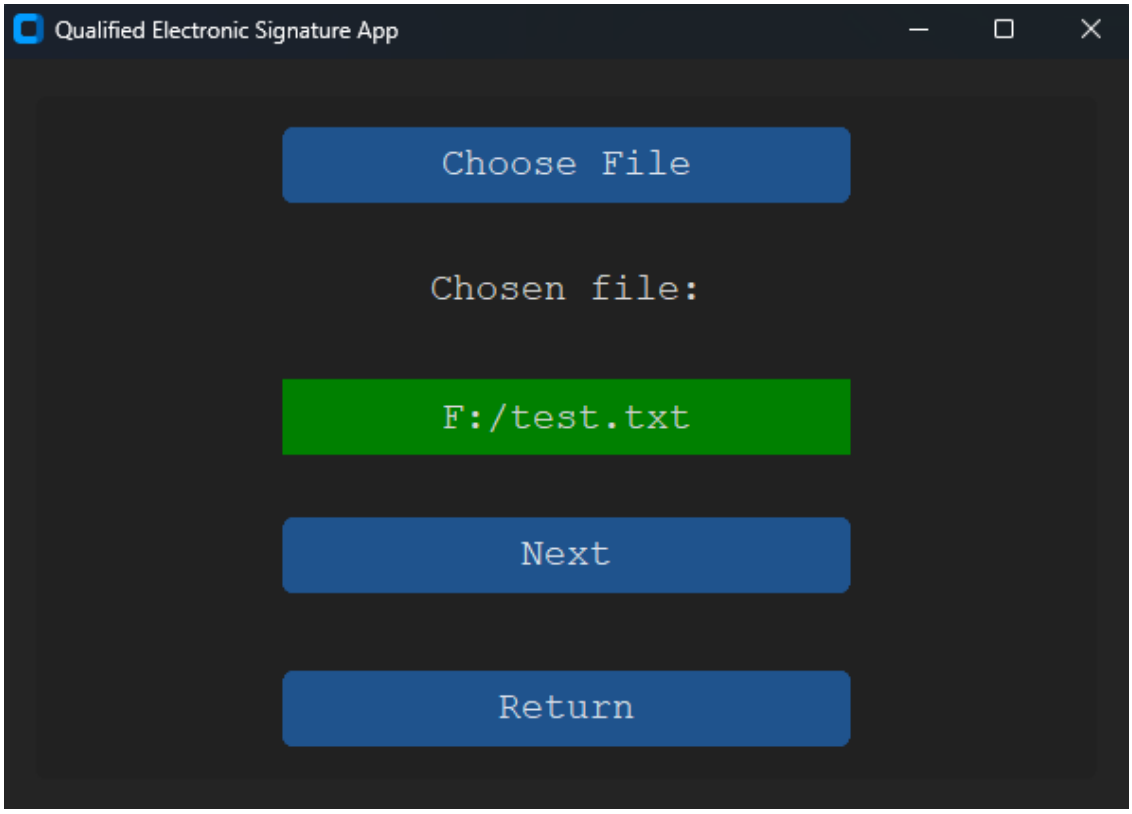


File before encryption

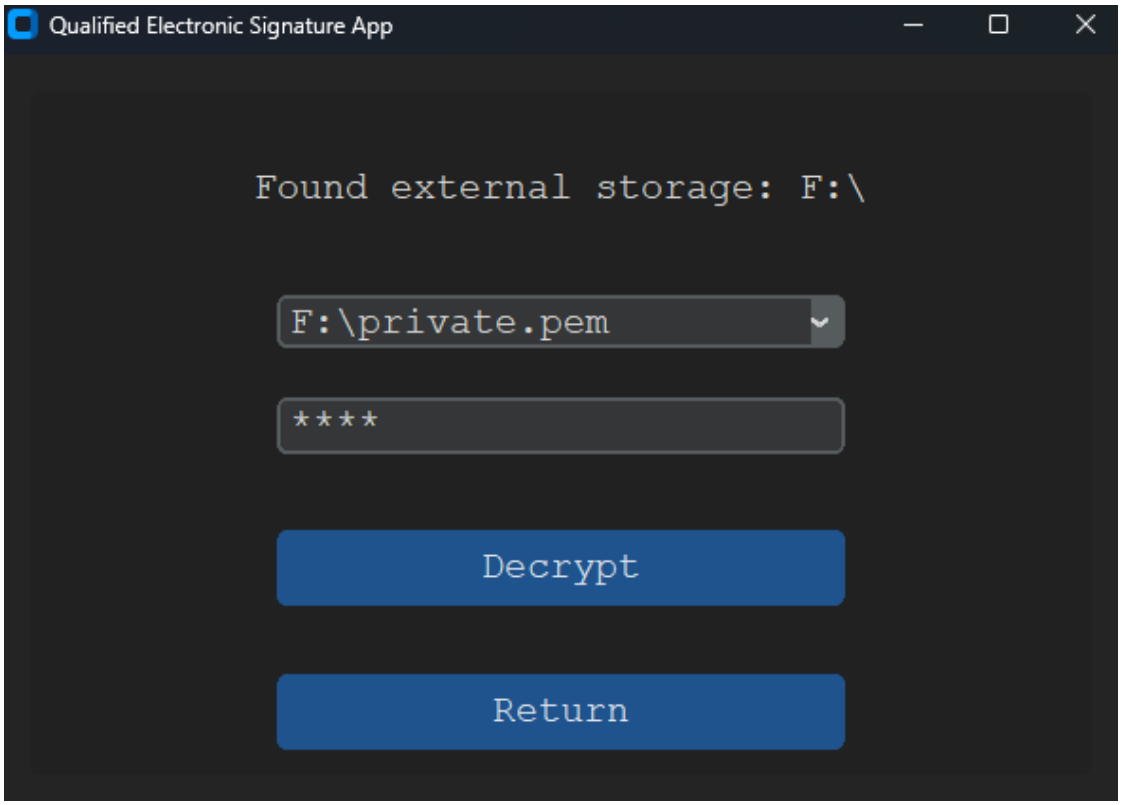


Encrypted file

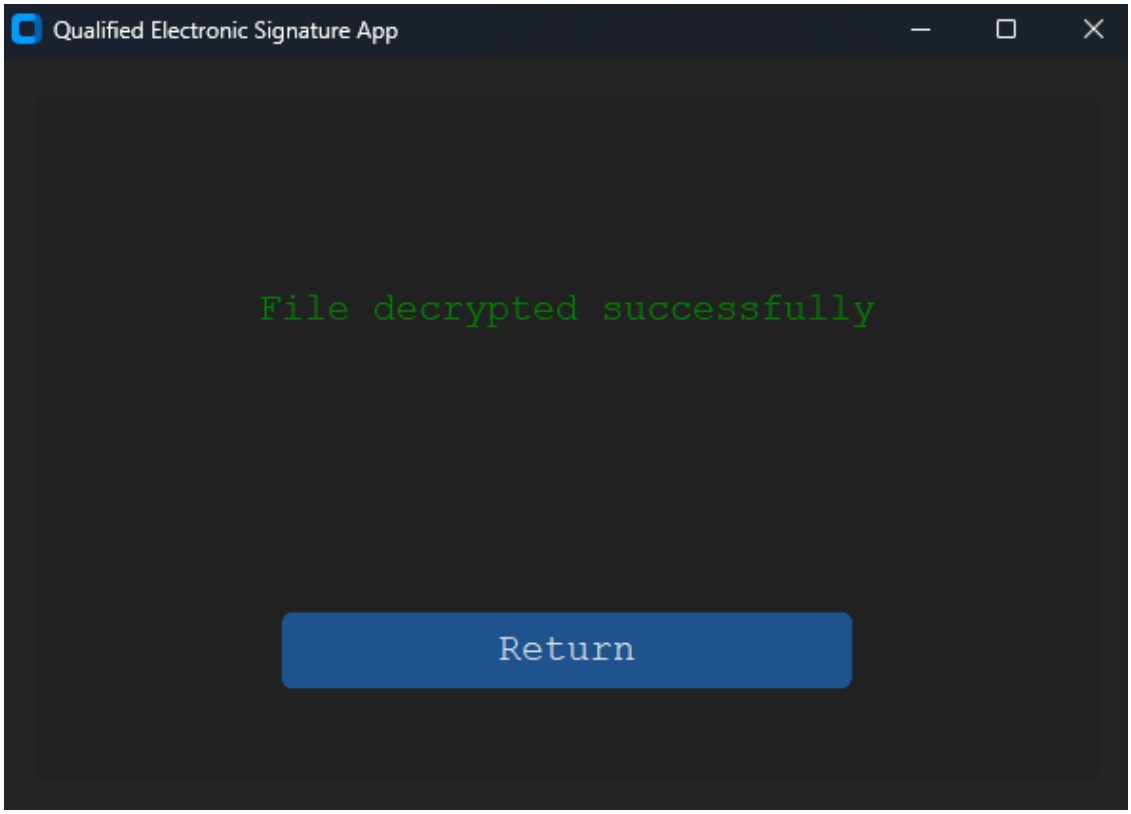
1.3.3 General purpose decryption



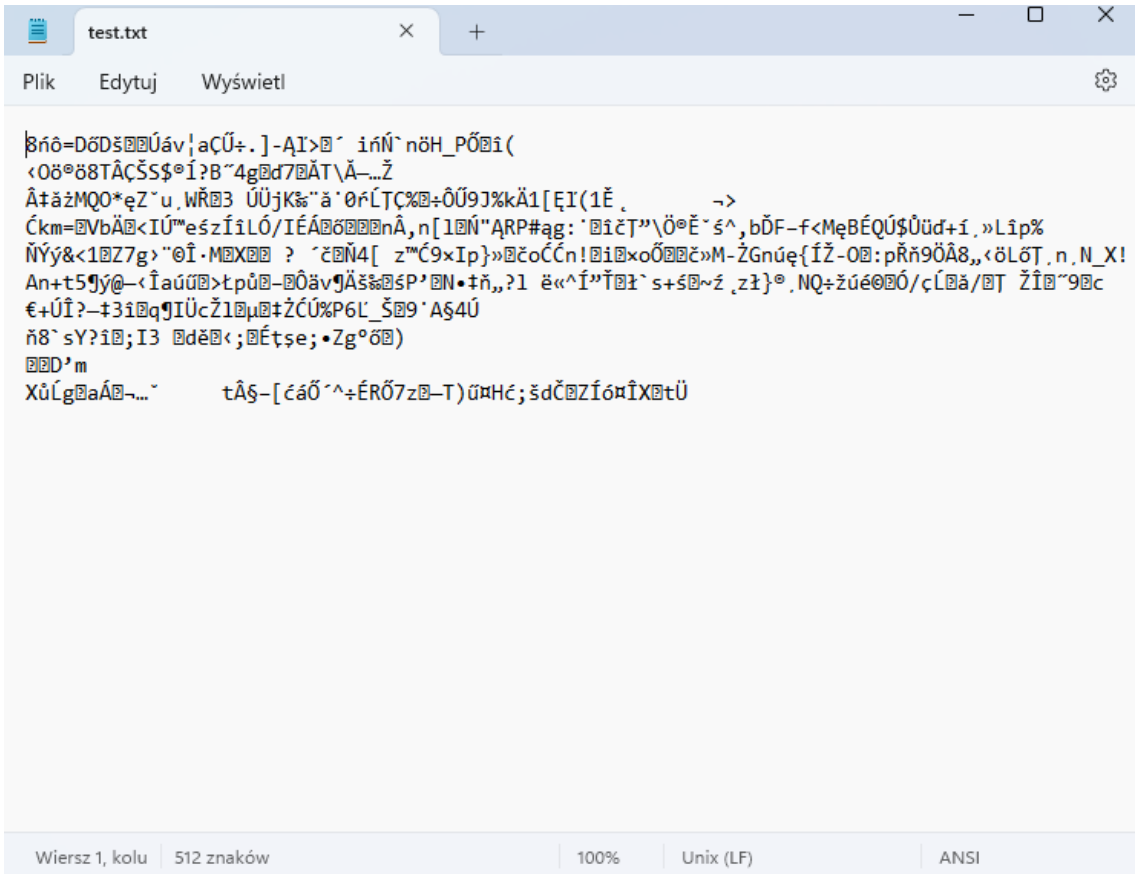
Decrypt Page 1 – Chosen file to decrypt



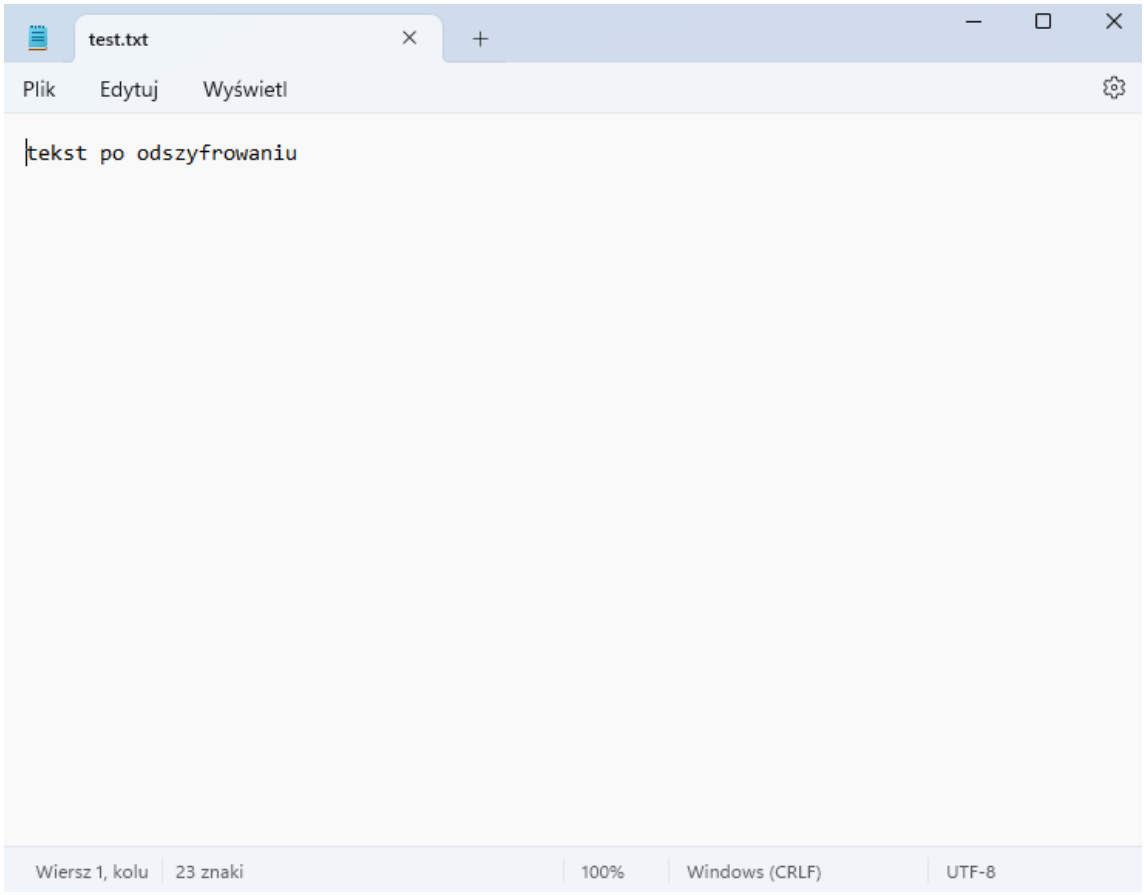
Decrypt Page 2 - Choosing private key from external device and entering the pin for key decryption



Decrypt Page 3 – Successful decryption

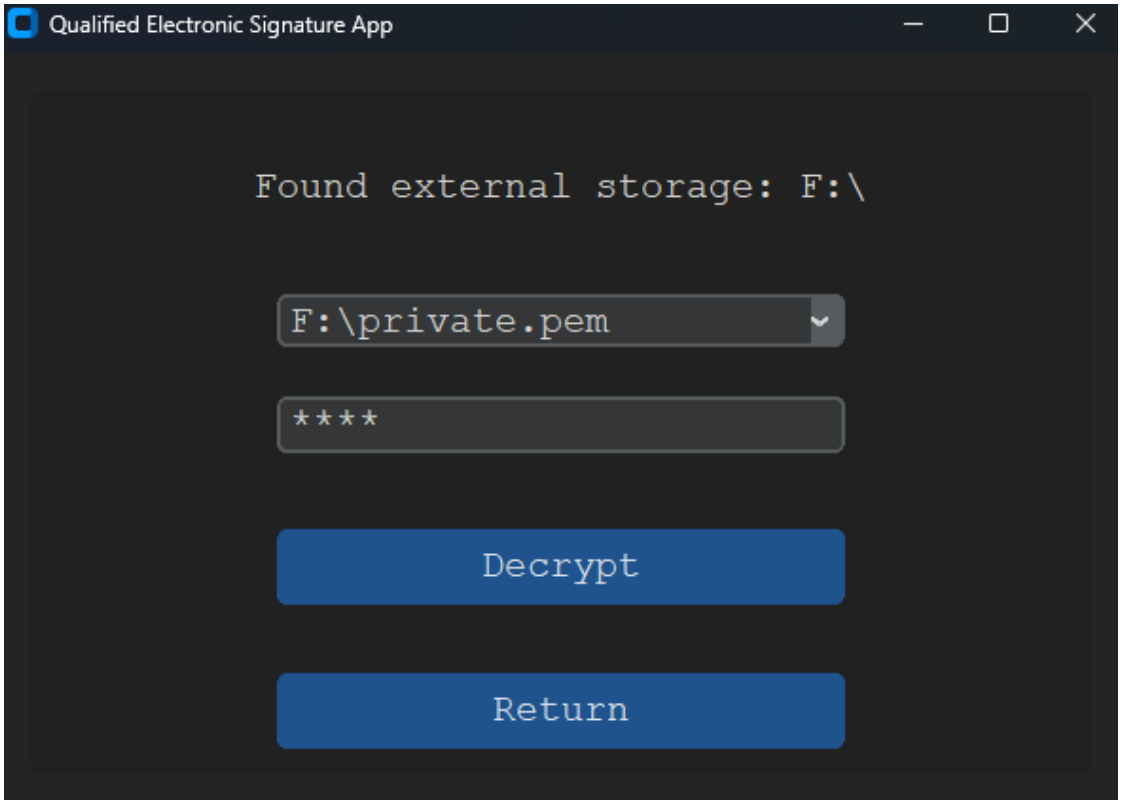


Encrypted file

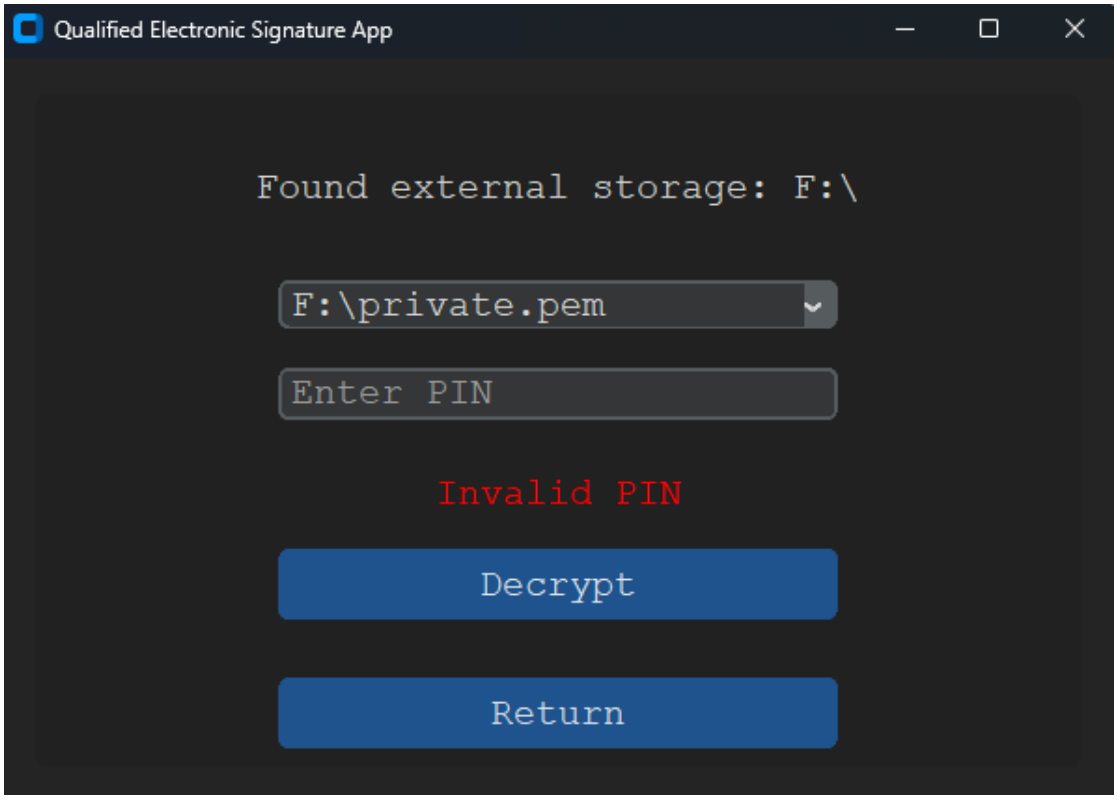


Decrypted file

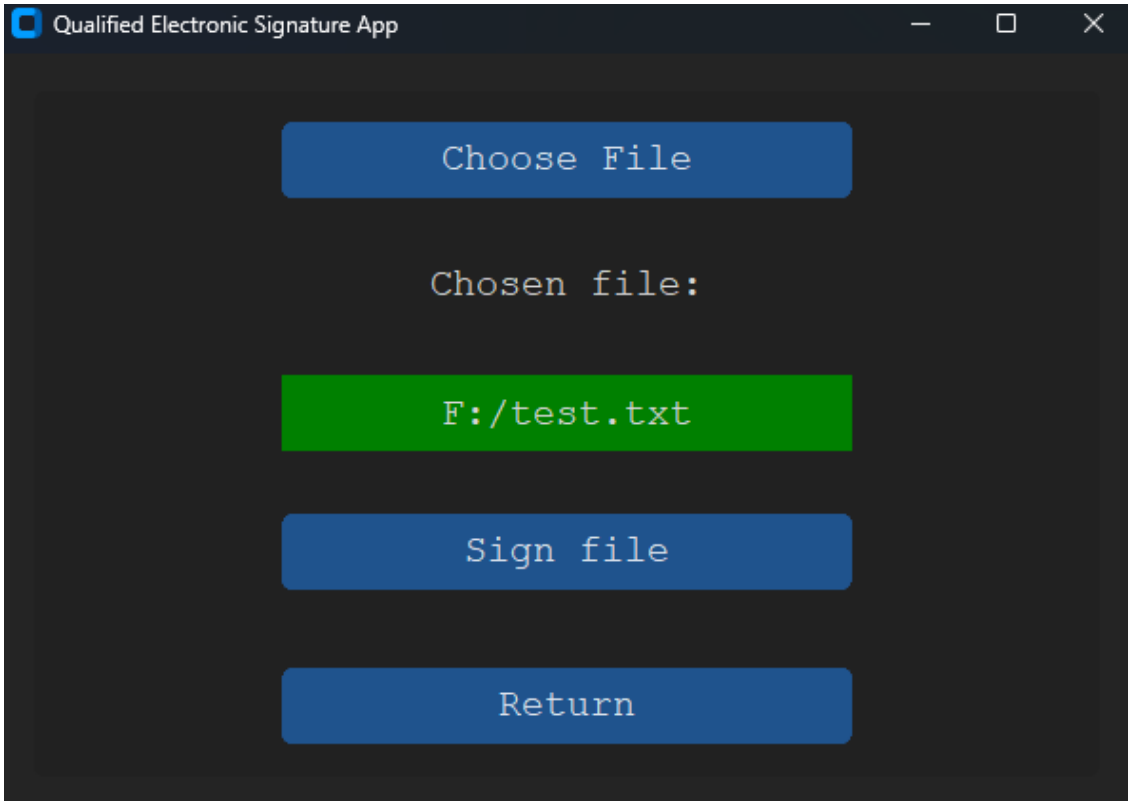
1.3.4 Signing file



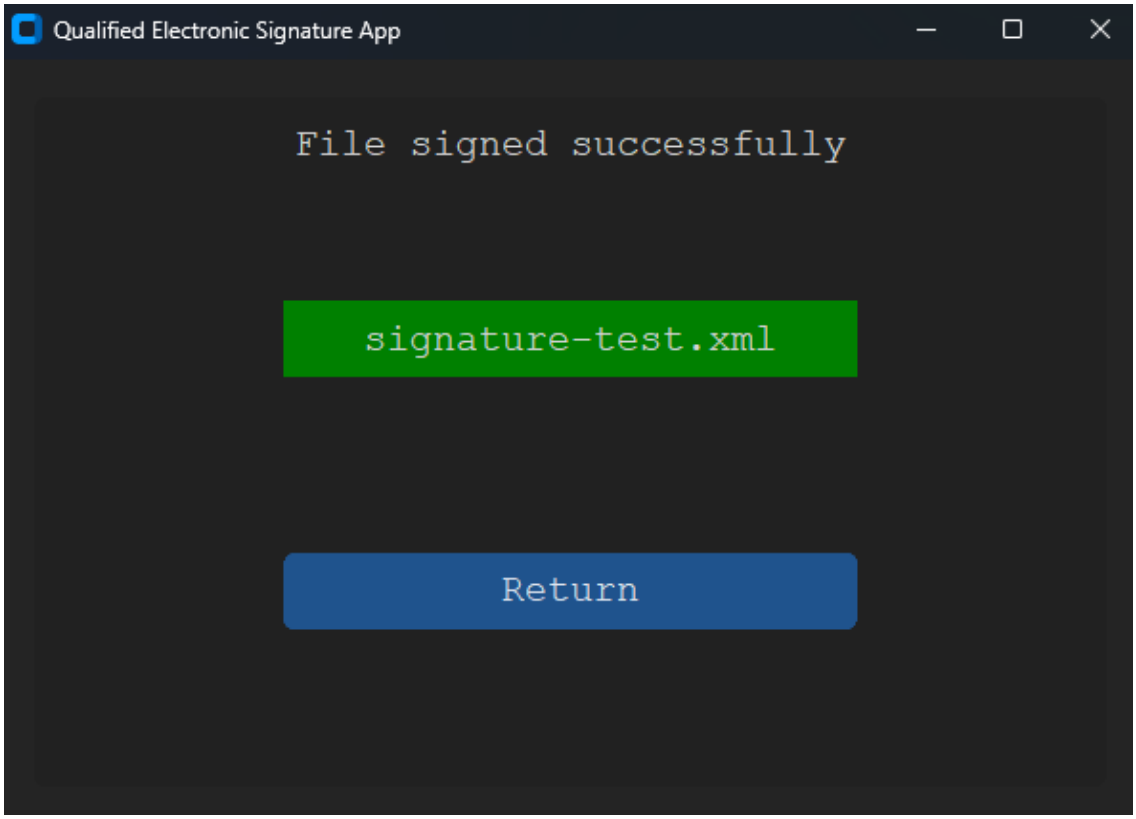
Sign file Page 1 – Choosing private key from external device and entering the pin for key decryption



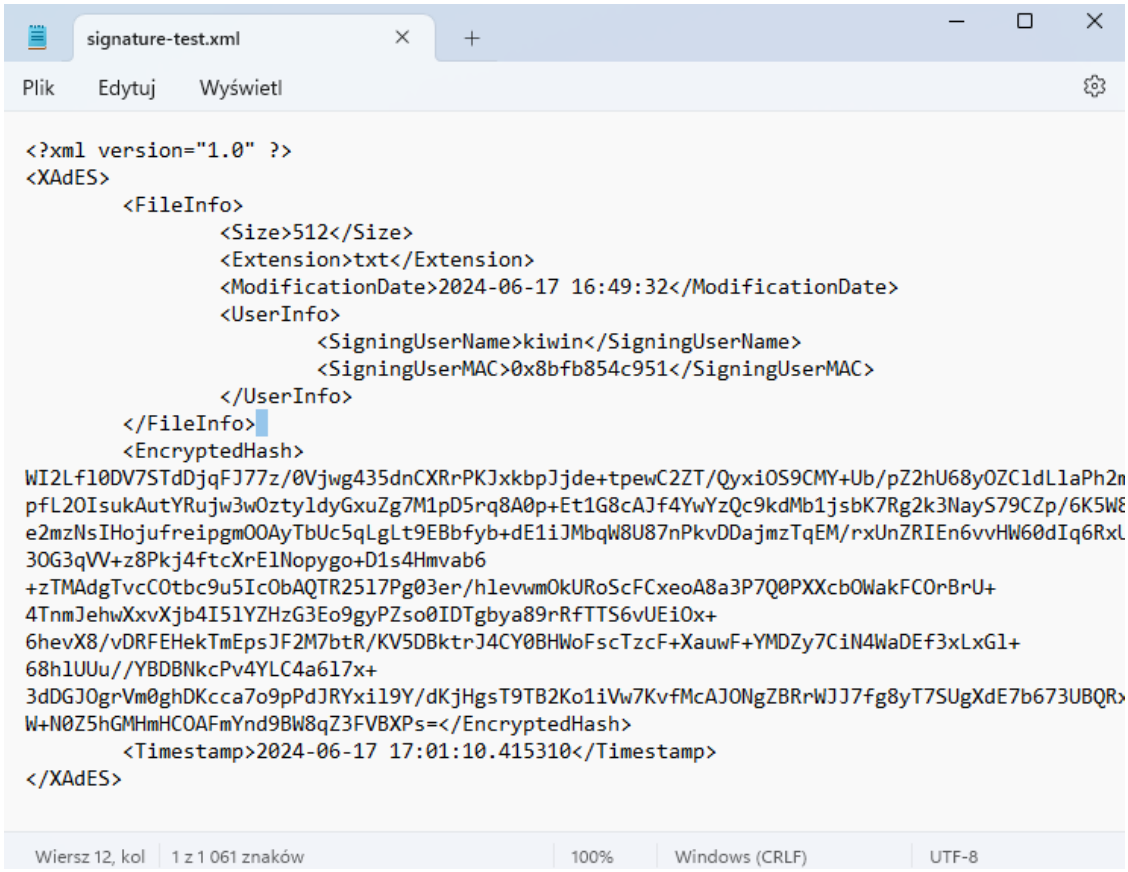
Sign file Page 1 – Invalid pin information



Sign file Page 2 – Choosing file to sign

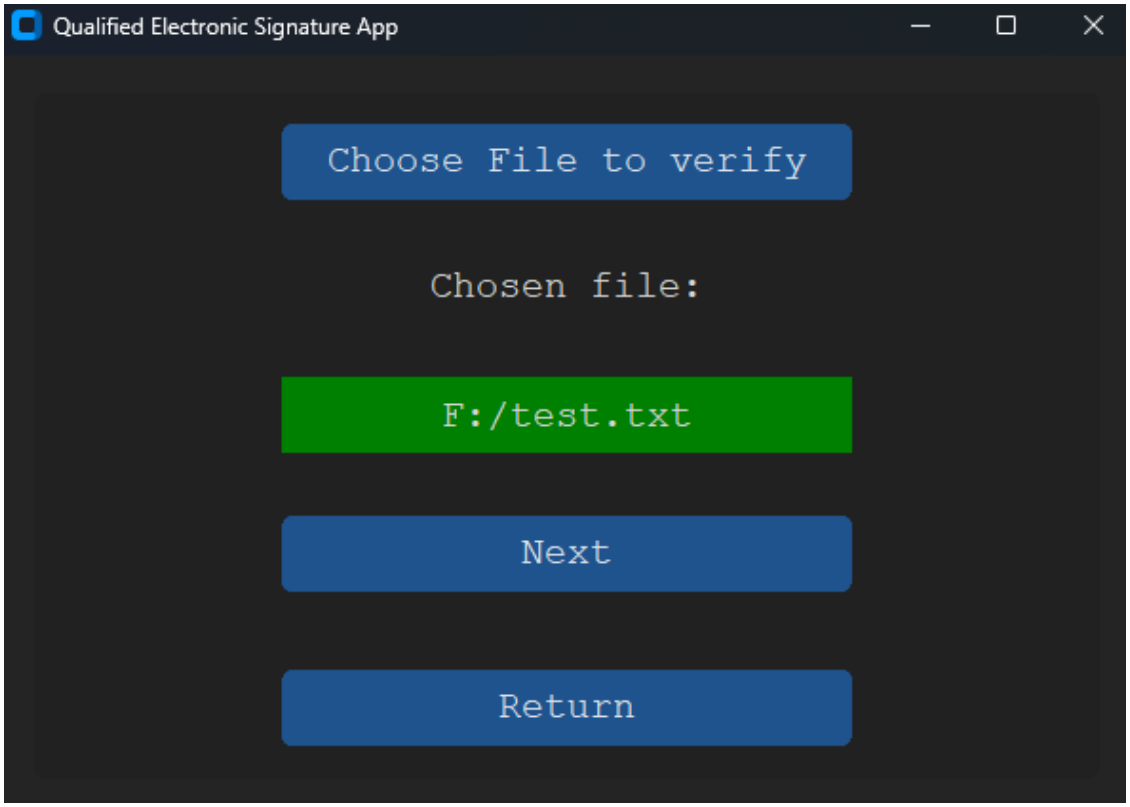


Sign file Page 3 - Success

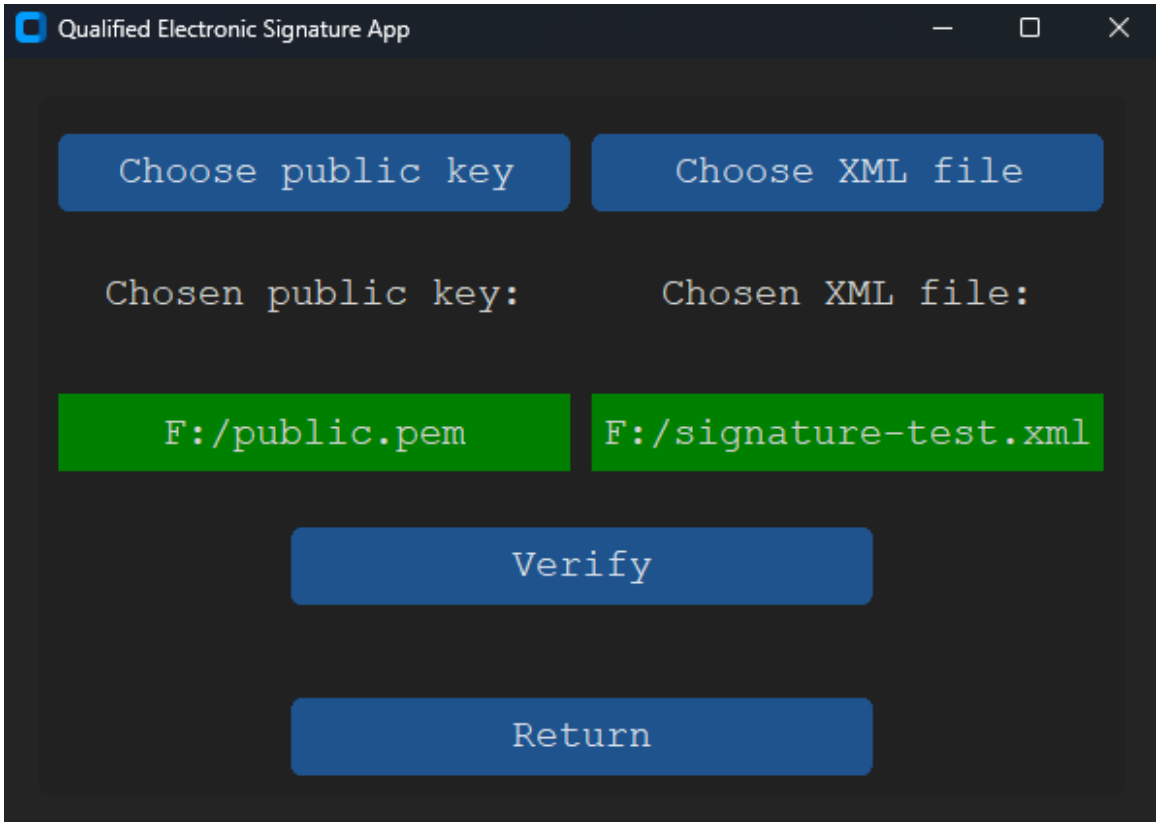


Signature

1.3.5 Signature verification

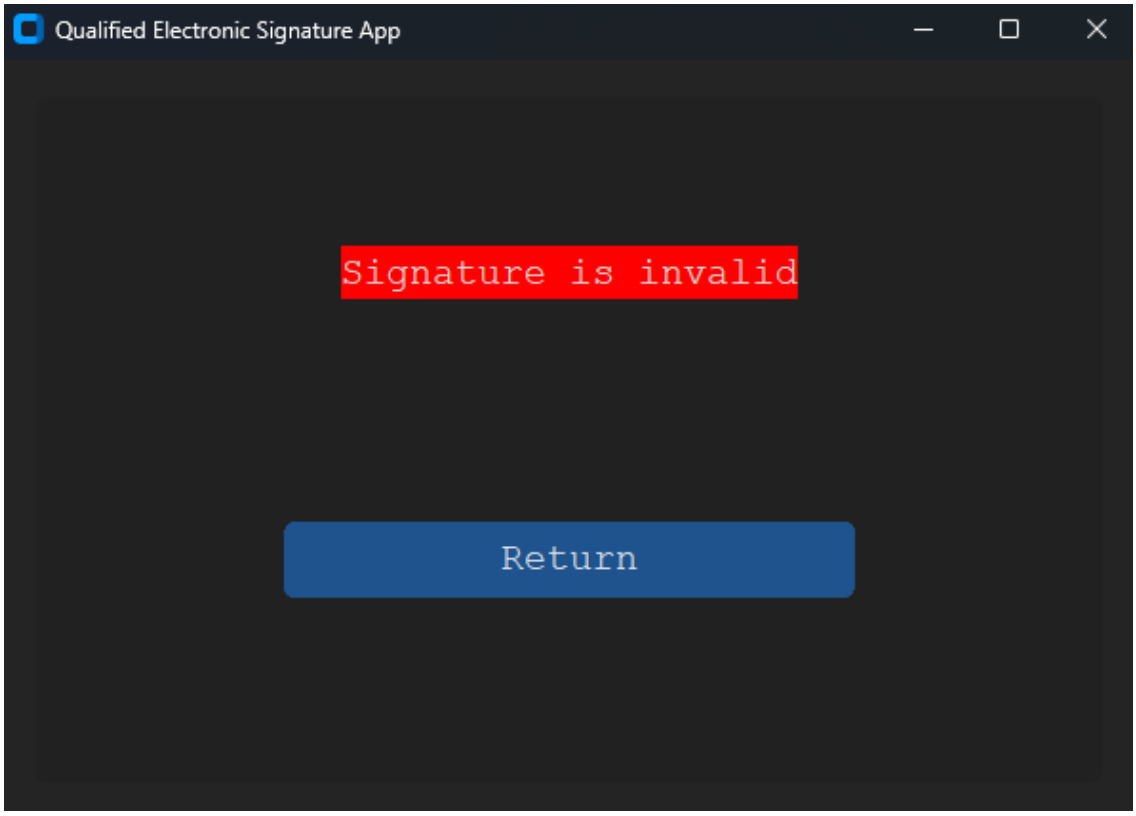


Verify the signature Page 1 – Chosen file to verification



Verify the signature Page 2 – Chosen valid public key and xml signature file

Verify the signature Page 3 – Successful validation



Verify the signature Page 3 – Unsuccessful validation

1.4 Results

The application successfully generates and verifies digital signatures in accordance with the XAdES standard. During testing, it efficiently processed *.html and *.txt files, performing the required signing and encryption tasks. RSA keys are securely generated, with the private key encrypted on a USB drive, ensuring high security and functionality.

1.5 Summary

This project provides a user-friendly and secure tool for digital signatures and file encryption, meeting all specified project requirements. It adheres to essential cryptographic standards and offers an effective solution for ensuring document security and integrity, demonstrating the application's operational success and reliability.

2. Literature

[1] PyCryptodome’s documentation, https://www.pycryptodome.org

[2] Lecture slides,https://enauczanie.pg.edu.pl/moodle/course/view.php?id=31195