

---

# **Security of Computer Systems**

## **Project Report**

Authors:  
Jan Barczewski 188679  
Radosław Gajewski 188687

Version: 1.1

---

## Versions

Version	Date	Description of changes
1.0	19.04.2024	Creation of the document
1.1	31.05.2024	Project – Final term

## 1. Project – control term

### 1.1 Description

Całość projektu realizowana jest za pomocą języka python. Do wygenerowania kluczy RSA oraz szyfrowania została wykorzystana biblioteka *cryptography*. Aplikacja okienkowa została zrealizowana za pomocą biblioteki *tkinter*.

Na stan obecny powstała aplikacja konsolowa do generowania pary kluczy RSA. Klucz prywatny jest dodatkowo szyfrowany algorytmem AES za pomocą pinu wprowadzanego ręcznie przez użytkownika oraz zapisywany na odpowiednio nazwanym pendrive. Aplikacja prosi użytkownika o uprzednie odpowiednie nazwanie pendrive'a w celu poprawnego odnalezienia urządzenia.

Rozpoczęliśmy również pracę nad drugą aplikacją, w wersji okienkowej. Aplikacja wykrywa już podłączonego pendrive'a z kluczem i za jego pomocą pozwala na podpisanie dokumentu po podaniu poprawnego hasła. Nie zostały jeszcze narzucone ograniczenia, jeśli chodzi o typ dokumentu.

### 1.2 Results

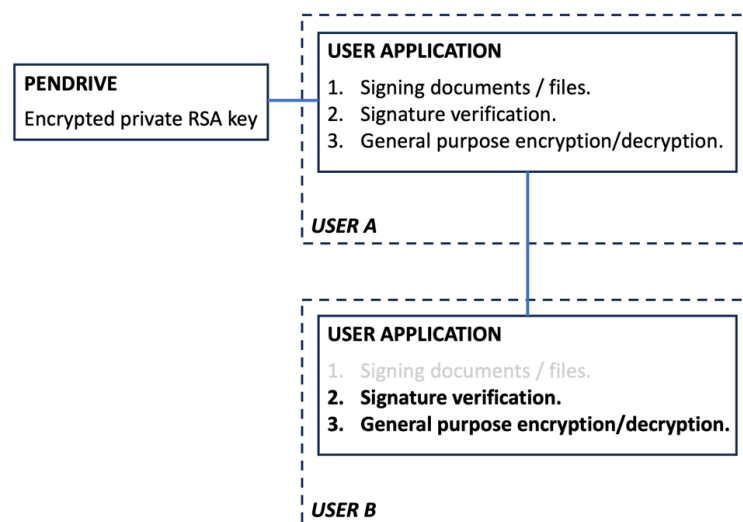


Fig. 1 – Block diagram.

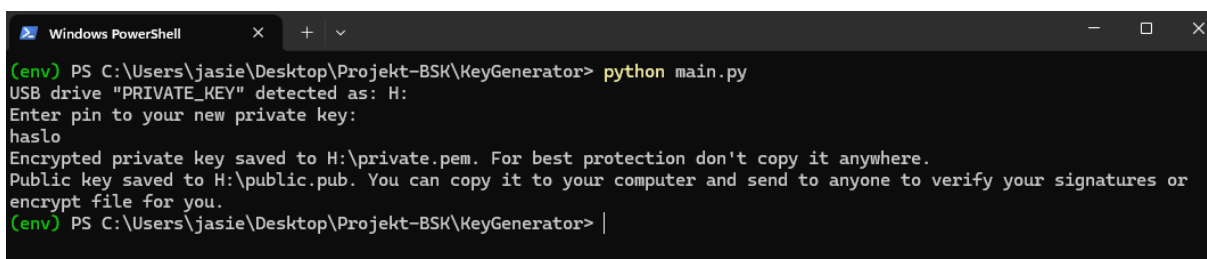
Działanie rozpoczynamy od wygenerowania pary kluczy RSA za pomocą aplikacji konsolowej. Aby skorzystać z aplikacji, należy wcześniej podłączyć pendrive nazwany „PRIVATE\_KEY”. Aplikacja poprosi użytkownika o wprowadzenie pinu do zaszyfrowania klucza prywatnego a następnie zapisze go na podłączonym urządzeniu.

---

Druga aplikacja umożliwia wykrycie podłączonego pendrive'a z wygenerowanym kluczem RSA. W celu podpisania dokumentu należy wybrać z eksploratora plików odpowiedni dokument oraz podać poprawne hasło. Po wykonaniu tych czynności aplikacja wygeneruje plik .xml zawierający:

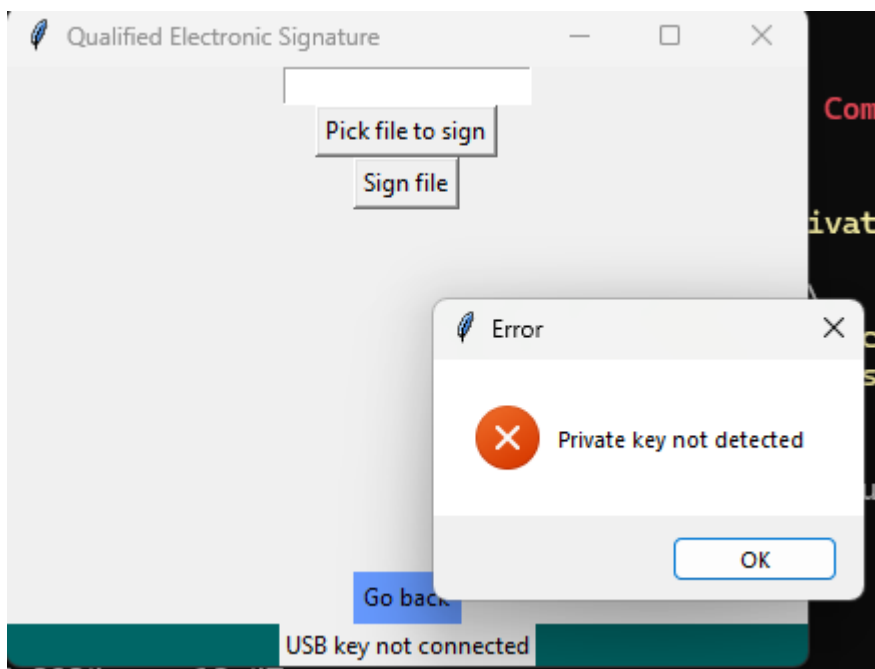
- Informacje ogólne o dokumencie
- Informacje o podpisującym użytkowniku
- Hash dokumentu zaszyfrowany kluczem RSA
- Datę i godzinę podpisu

Poniższe zdjęcia przedstawiają rezultaty dotychczasowej pracy:

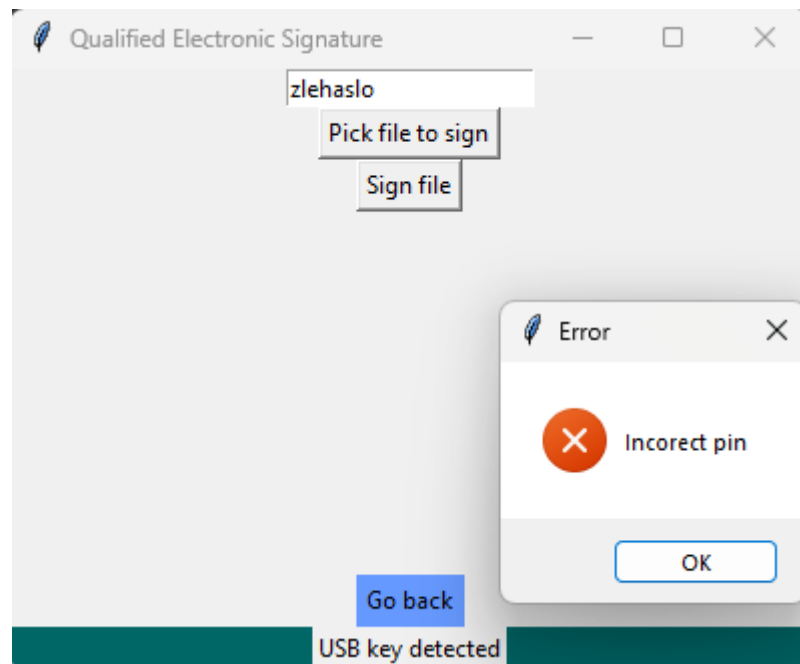


```
Windows PowerShell
(env) PS C:\Users\jasie\Desktop\Projekt-BSK\KeyGenerator> python main.py
USB drive "PRIVATE_KEY" detected as: H:
Enter pin to your new private key:
haslo
Encrypted private key saved to H:\private.pem. For best protection don't copy it anywhere.
Public key saved to H:\public.pub. You can copy it to your computer and send to anyone to verify your signatures or
encrypt file for you.
(env) PS C:\Users\jasie\Desktop\Projekt-BSK\KeyGenerator> |
```

Zdjęcie 1: działanie aplikacji generującej parę kluczy RSA



Zdjęcie 2: Druga aplikacja – nie wykryto pendrive'a



Zdjęcie 3: Druga aplikacja – Podano złe hasło

```
</> txtFile.txt.xml X
C: > Users > jasio > Desktop > </> txtFile.txt.xml
1  <Signature>
2  <DocumentInfo>
3      <Name>txtFile.txt</Name>
4      <Size>3210</Size>
5      <Extension>.txt</Extension>
6      <DateModified>2024-03-18T20:10:51</DateModified>
7  </DocumentInfo>
8  <SigningUser>
9      <Name>jasio</Name>
10 </SigningUser>
11 <Signature>nyX1lfk3MVyq91ljBwPqRVW5EZpI2mJiP4c6ifQjKS2DHQWq5IGxD
12 <Timestamp>
13     <LocalTime>2024-04-19T20:20:22</LocalTime>
14 </Timestamp>
15 </Signature>
```

Zdjęcie 4: Wygenerowany plik .xml

### 1.3 Summary

Aplikacja do generowania kluczy RSA została w pełni ukończona.

Druga aplikacja na chwilę obecną zawiera podstawową funkcjonalność podpisywania dokumentów, którą należy udoskonalić (między innymi o wprowadzenie ograniczeń do rozszerzeń plików). Należy również uzupełnić ją o brakujące funkcjonalności (sprawdzanie podpisu, szyfrowanie/deszyfrowanie pliku) oraz udoskonalić interfejs aplikacji.

---

## 2. Project – Final term

### 2.1 Description

Całość projektu realizowana jest za pomocą języka python. Do wygenerowania kluczy RSA oraz szyfrowania została wykorzystana biblioteka *cryptography*. Aplikacja okienkowa została zrealizowana za pomocą biblioteki *tkinter*.

W ramach projektu powstała aplikacja konsolowa do generowania pary kluczy RSA. Klucz prywatny jest dodatkowo szyfrowany algorytmem AES za pomocą pinu wprowadzanego ręcznie przez użytkownika oraz zapisywany na odpowiednio nazwanym pendrive. Aplikacja prosi użytkownika o uprzednie odpowiednie nazwanie pendrive'a w celu poprawnego odnalezienia urządzenia.

Druga aplikacja wykrywa podłączony pendrive'a z kluczem i za jego pomocą pozwala na podpisanie dokumentu po podaniu poprawnego hasła. Zostały narzucone ograniczenia, jeśli chodzi o typ dokumentu – txt i pdf. Dodatkowo zaimplementowano funkcjonalność weryfikacji takiego podpisu wskazując odpowiedni klucz publiczny. Możliwe jest również szyfrowanie i deszyfrowanie małych (~1kB) plików o rozszerzeniu txt, również za pomocą kluczy RSA.

### 2.2 Code Description

**Funkcja podpisująca:**

```

def sign_file(key_path, pin, file_path):
    pin_hash = hashlib.sha256(pin.encode())

    key_file = open(key_path, "r")
    key = key_file.read().encode()
    key_file.close()

    try:
        private_key = load_pem_private_key(key, pin_hash.digest())
    except ValueError:
        raise ValueError("Incorrect pin")

    file_to_sign = open(file_path, "rb")
    file_content = file_to_sign.read()
    file_to_sign.close()

    file_hash = hashlib.sha256(file_content).digest()

    signature = private_key.sign(
        file_hash,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        utils.Prehashed(hashes.SHA256())
    )

```

Zdjęcie 5: Kod funkcji podpisującej dokument

Funkcja podpisująca zaczyna się od wygenerowania hasha pinu w celu odszyfrowania klucza prywatnego, za co odpowiada funkcja `load_pem_private_key()`. Wcześniej należy tylko wczytać klucz z pliku do pamięci. W przypadku podania złego pinu zwracany jest odpowiedni wyjątek. Następnie ładowany jest plik, który ma być podpisany, oraz generowany jest jego hash. Za wygenerowanie podpisu odpowiada funkcja `sign` obiektu `private_key` z biblioteki `cryptography`, która szyfruje hash dokumentu kluczem prywatnym. Jako argument podaje się również jaki padding ma być zastosowany.

---

```
signature_xml = ET.Element("Signature")

document_info = ET.SubElement(signature_xml, "DocumentInfo")
file_name = ET.SubElement(document_info, "Name")
file_name.text = os.path.basename(file_path)
size = ET.SubElement(document_info, "Size")
size.text = str(os.path.getsize(file_path))
extension = ET.SubElement(document_info, "Extension")
extension.text = os.path.splitext(file_path)[1]
date_modified = ET.SubElement(document_info, "DateModified")
date_modified.text = datetime.fromtimestamp(os.path.getmtime(file_path)).strftime("%Y-%m-%dT%H:%M:%S")

signing_user = ET.SubElement(signature_xml, "SigningUser")
name = ET.SubElement(signing_user, "Name")
name.text = os.getlogin()

signature_node = ET.SubElement(signature_xml, "Signature")
signature_node.text = base64.b64encode(signature).decode()

timestamp = ET.SubElement(signature_xml, "Timestamp")
local_time = ET.SubElement(timestamp, "LocalTime")
local_time.text = datetime.now().strftime("%Y-%m-%dT%H:%M:%S")

tree = ET.ElementTree(signature_xml)
ET.indent(tree, space="\t", level=0)
tree.write(file_path+".xml")
```

Zdjęcie 6: Kod funkcji podpisującej dokument cd.

Następnie generowane jest drzewo dokumentu xml za pomocą biblioteki xml.etree, do którego zapisywane są wszystkie dane podpisu cyfrowego. Następnie podpis ten jest zapisywany w tej samej lokalizacji co oryginalny dokument.

## Weryfikacja podpisu:

```
def verify_signature(signature_path, public_key_path):
    tree = ET.parse(signature_path)
    root = tree.getroot()
    signature = base64.b64decode(root.find('Signature').text)
    file_name = root.find("DocumentInfo").find("Name").text
    file_path = str(Path(signature_path).parent) + "\\\" + file_name

    file_to_verify = open(file_path, "rb")
    file_content = file_to_verify.read()
    file_to_verify.close()
    file_hash = hashlib.sha256(file_content).digest()

    key_file = open(public_key_path, "r")
    key = key_file.read().encode()
    key_file.close()
    public_key = load_pem_public_key(key)

    public_key.verify(
        signature,
        file_hash,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        utils.Prehashed(hashes.SHA256())
    )
```

Zdjęcie 7: Kod funkcji weryfikującej podpis

W celu weryfikacji podpisu, z pliku xml z podpisem ładowany jest sam podpis oraz nazwa dokumentu podpisywanego. Następnie dokument ten jest również ładowany do pamięci i generowany jest jego hash w celu weryfikacji. Klucz publiczny też jest wczytywany, służy do tego funkcja `load_pem_public_key` z biblioteki `cryptography`. Ostatnim krokiem jest wywołanie funkcji `verify` na obiekcie klucza, która odszyfrowuje podany podpis i porównuje go z haszem dokumentu w celu sprawdzenia zgodności. Wymaga ona podania paddingu jaki został użyty w procesie podpisywania. W przypadku niezgodności obu hashy zwracany jest odpowiedni wyjątek.

## Zaszyfrowanie pliku:



---

```
def encrypt_file(file_path, public_key_path):
    file_to_encrypt = open(file_path, "rb")
    file_content = file_to_encrypt.read()
    file_to_encrypt.close()

    key_file = open(public_key_path, "r")
    key = key_file.read().encode()
    key_file.close()
    public_key = load_pem_public_key(key)

    cyphertext = public_key.encrypt(
        file_content,
        padding.OAEP(
            mgf=padding.MGF1(hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    encrypted_file = open(file_path+".encrypted", "wb")
    encrypted_file.write(cyphertext)
    encrypted_file.close()
```

Zdjęcie 8: Kod funkcji szyfrującej plik

W celu zaszyfrowania pliku należy załadować go do pamięci, tak samo jak klucz publiczny, który posłuży do szyfrowania. Następnie na obiekcie klucza wywoływana jest metoda encrypt, która zwraca szyfrogram, aby następnie można było zapisać go na dysku. Funkcja ta wymaga podania paddingu, który zostanie użyty w procesie szyfrowania.

### Deszyfrowanie pliku:

```
def decrypt_file(file_path, private_key_path, pin):
    pin_hash = hashlib.sha256(pin.encode())

    key_file = open(private_key_path, "r")
    key = key_file.read().encode()
    key_file.close()

    try:
        private_key = load_pem_private_key(key, pin_hash.digest())
    except ValueError:
        raise ValueError("Incorrect pin")

    file_to_decrypt = open(file_path, "rb")
    file_content = file_to_decrypt.read()
    file_to_decrypt.close()

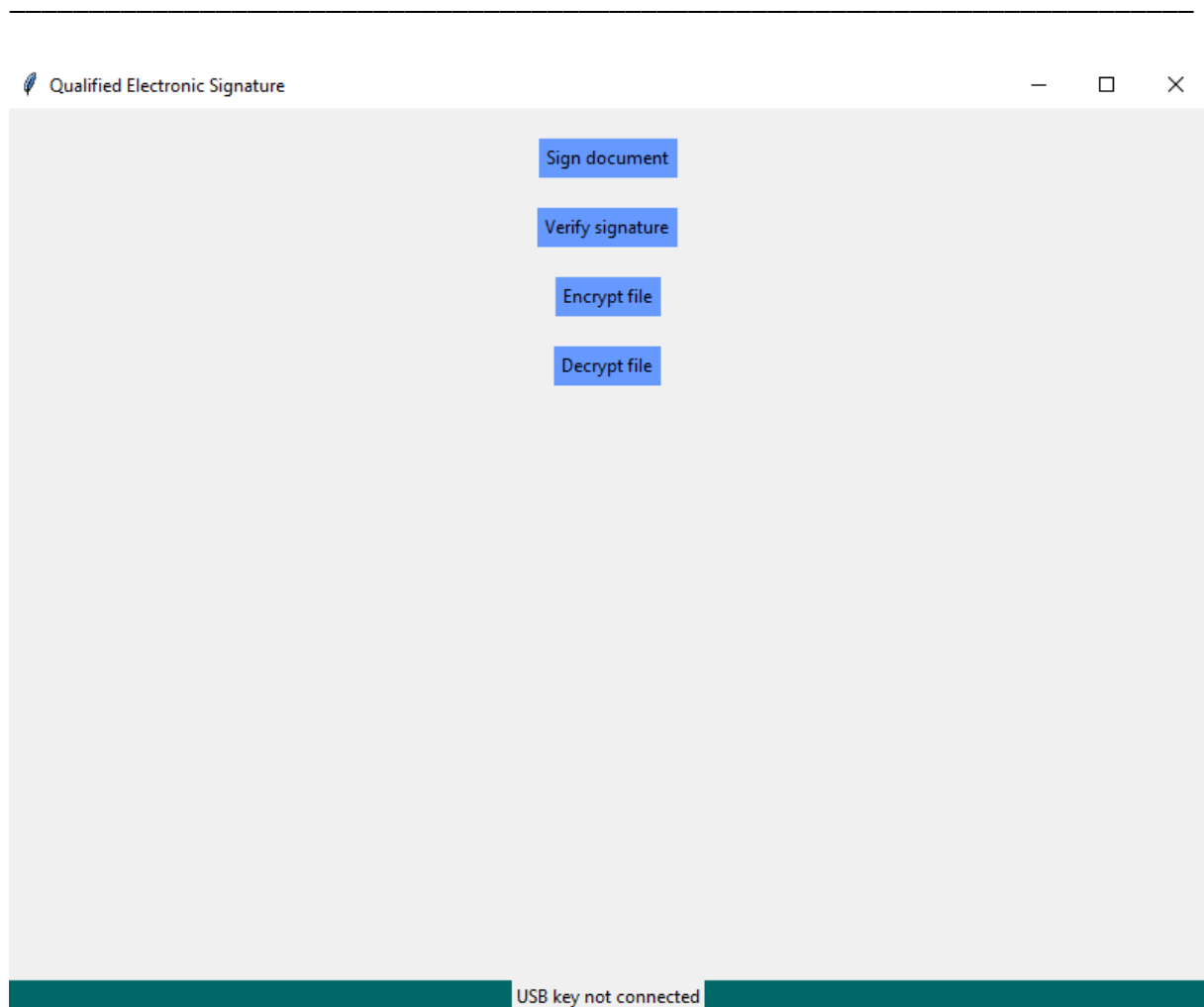
    plaintext = private_key.decrypt(
        file_content,
        padding.OAEP(
            mgf=padding.MGF1(hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    decrypted_file = open(str(file_path).removesuffix(".encrypted"), "wb")
    decrypted_file.write(plaintext)
    decrypted_file.close()
```

Zdjęcie 9: Kod funkcji deszyfrującej plik

Funkcja odszyfrowująca zaczyna się podobnie jak podpisywanie - od wygenerowania hasha pinu w celu odszyfrowania klucza prywatnego, za co odpowiada funkcja `load_pem_private_key()`. Wcześniej należy tylko wczytać klucz z pliku do pamięci. W przypadku podania złego pinu zwracany jest odpowiedni wyjątek. Następnie ładowany jest plik, który ma być odszyfrowany. Za tą operację odpowiada funkcja `decrypt` obiektu `private_key` z biblioteki `cryptography`. Jako argument należy podać padding, taki sam jaki był użyty w procesie szyfrowania. Ostatnim krokiem jest zapisanie zwróconych przez funkcję bajtów do pliku.

## 2.3 Results

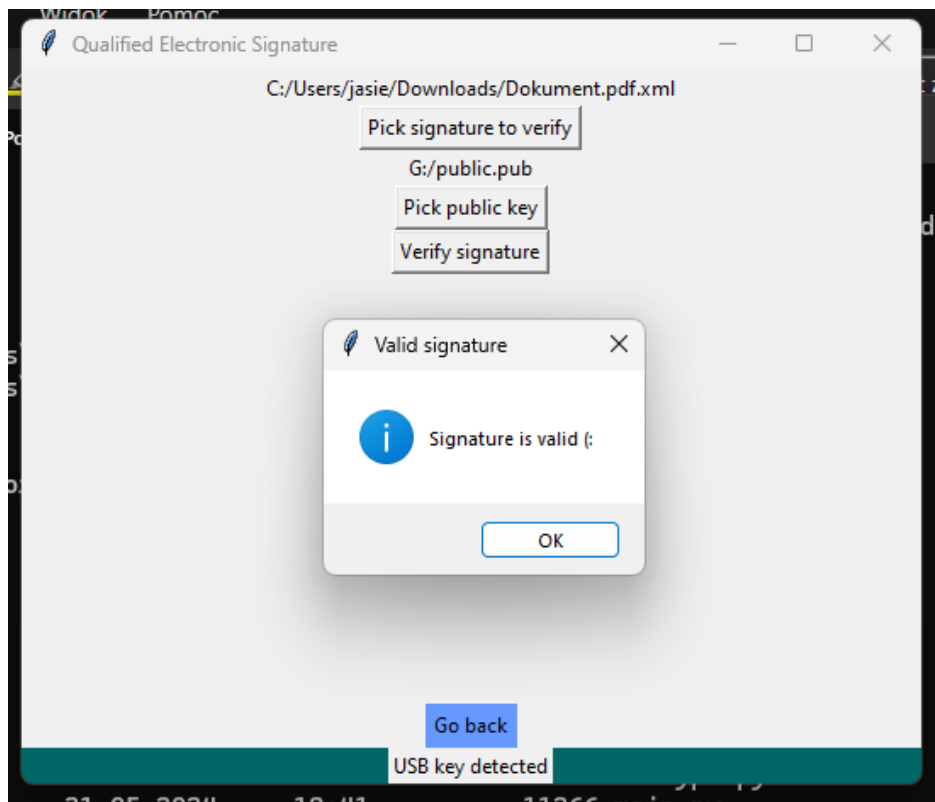


Zdjęcie 10: Menu główne aplikacji

Powyższe zdjęcie przedstawia menu główne aplikacji. Funkcjonalność podpisywania dokumentu i osobna aplikacja generująca klucze zostały opisane w rezultatach etapu kontrolnego. Poniżej przedstawione zostaną pozostałe funkcjonalności aplikacji.

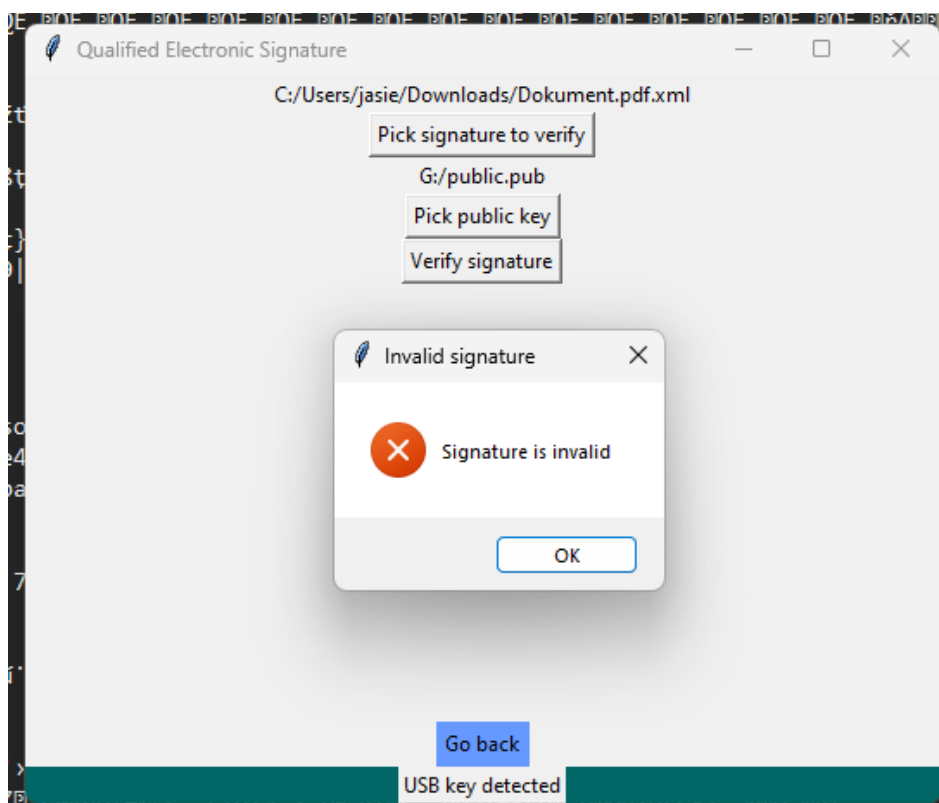
### **Weryfikacja podpisu:**

W celu zweryfikowania podpisu, należy wskazać plik .xml zawierający ten podpis oraz klucz publiczny, który chcemy sprawdzić jako autora. Ważne jest, aby plik, którego podpis jest sprawdzany był w tej samej lokalizacji co plik z podpisem. Jeśli podpis jest ważny, czyli został podpisany przez klucz prywatny wygenerowany w parze z podanym kluczem publicznym oraz plik nie został później zmodyfikowany, aplikacja pokaże poniższy komunikat:



Zdjęcie 11: Podpis ważny

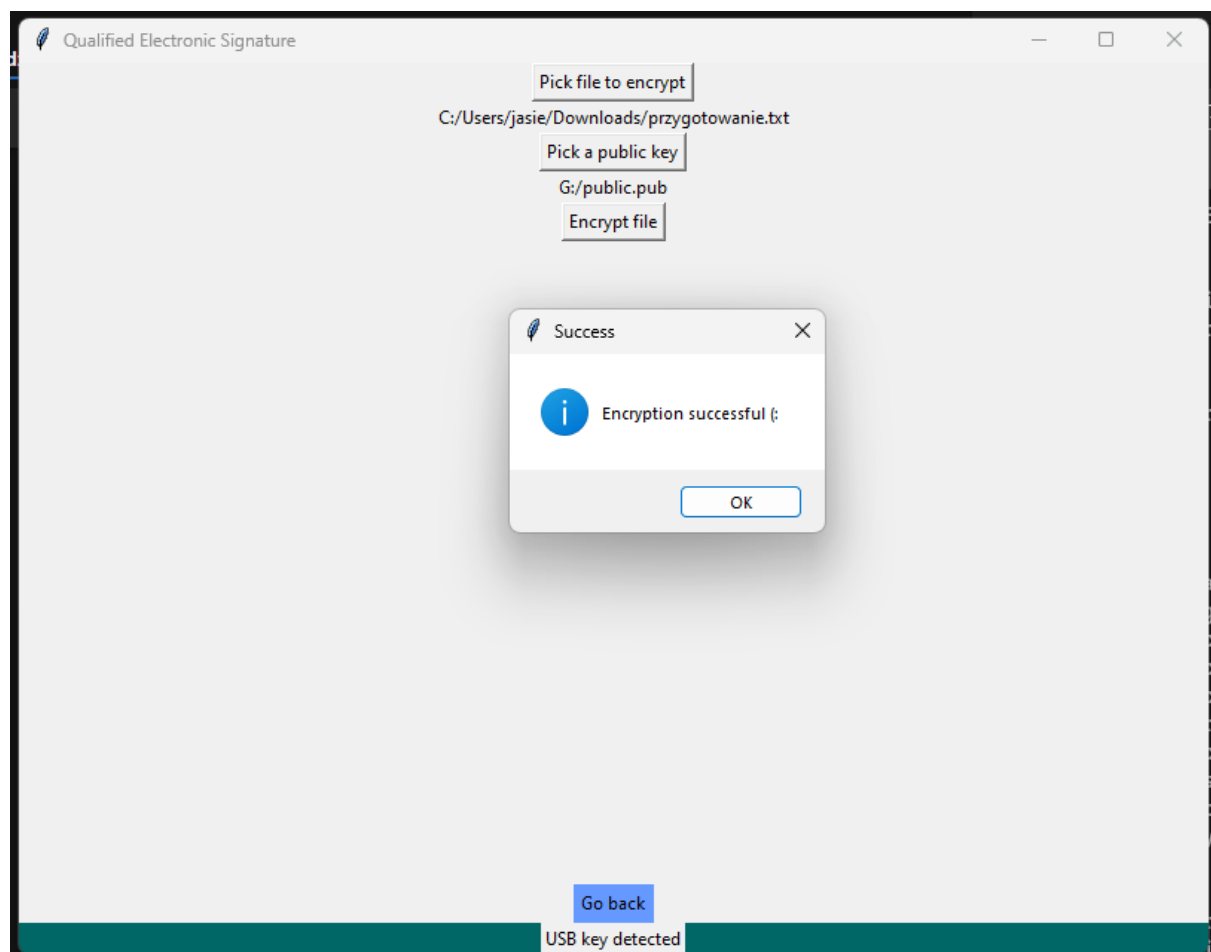
W przeciwnym wypadku aplikacja poinformuje nas o nieważności podpisu:



Zdjęcie 12: Podpis nieważny

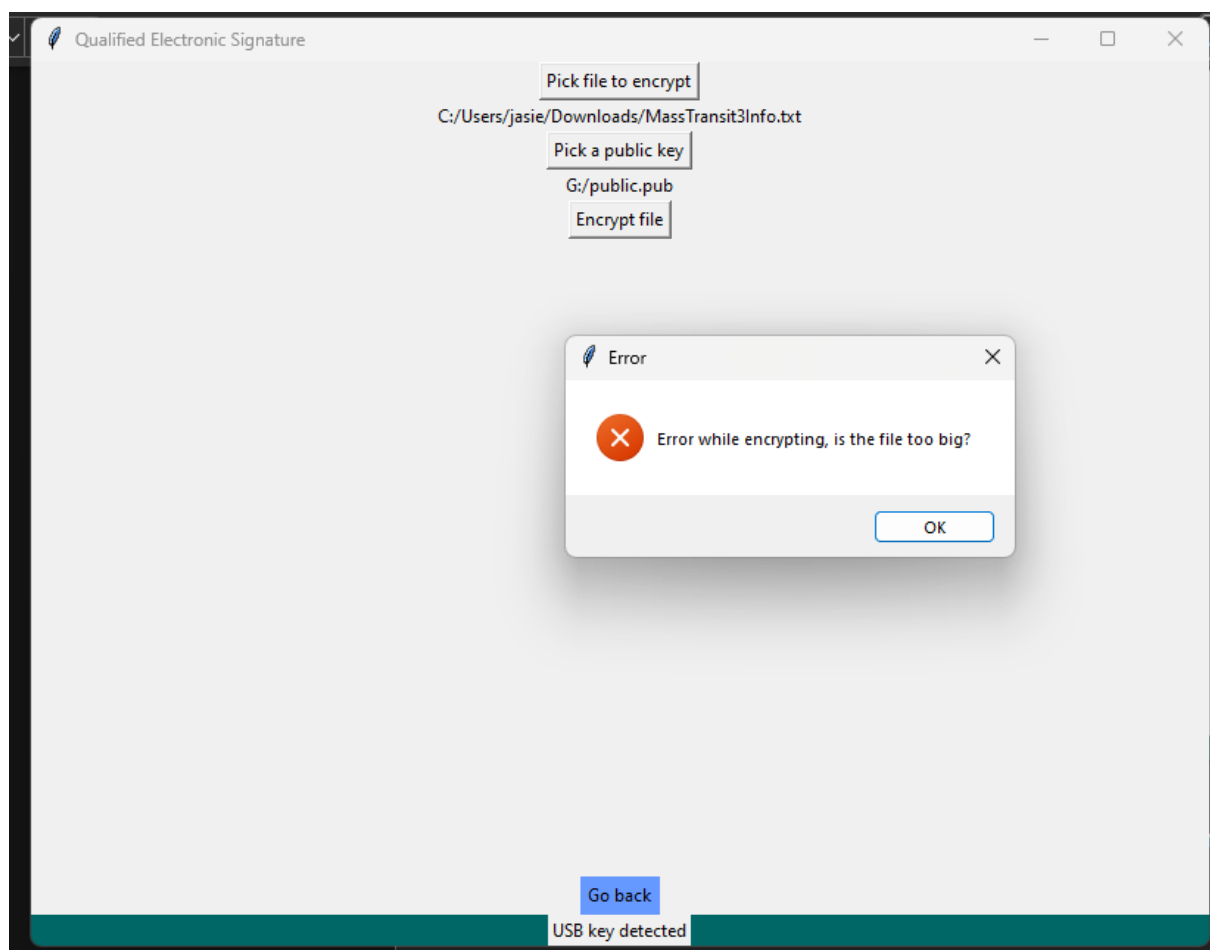
### Szyfrowanie pliku:

Aby zaszyfrować plik, aplikacja poprosi nas o wskazanie pliku do podpisania (musi być to plik niewielkich rozmiarów w formacie txt – ok 1kB) oraz klucza publicznego. Jeżeli szyfrowanie się powiedzie, w tej samej lokalizacji co oryginalny plik pojawi się plik zaszyfrowany z dodatkiem “.encrypted” w nazwie oraz aplikacja wyświetli następujący komunikat:



Zdjęcie 13: Szyfrowanie zakończone sukcesem

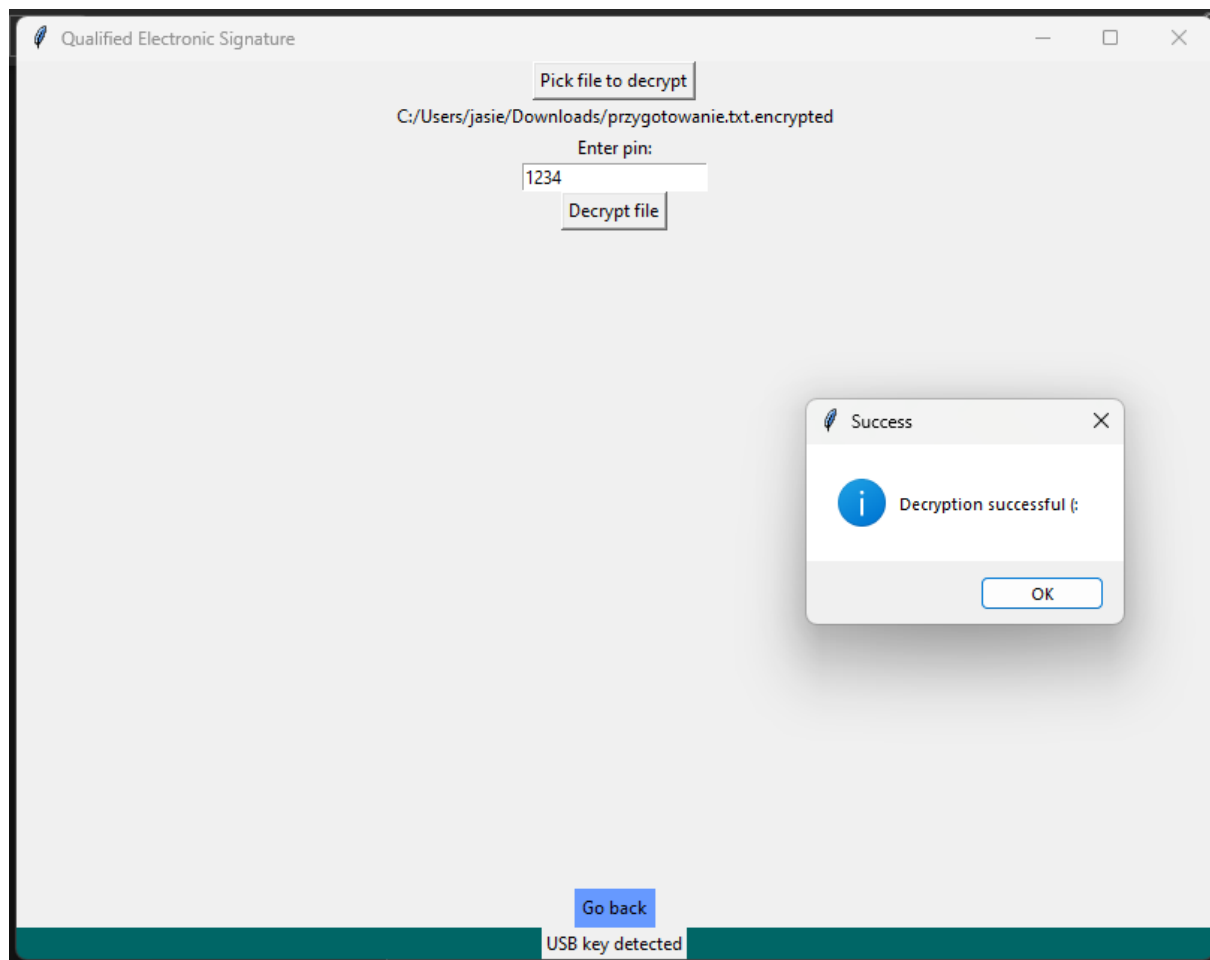
**Jeżeli podany zostanie zbyt duży plik, aplikacja poinformuje użytkownika o błędzie:**



Zdjęcie 14: Szyfrowanie zakończone niepowodzeniem

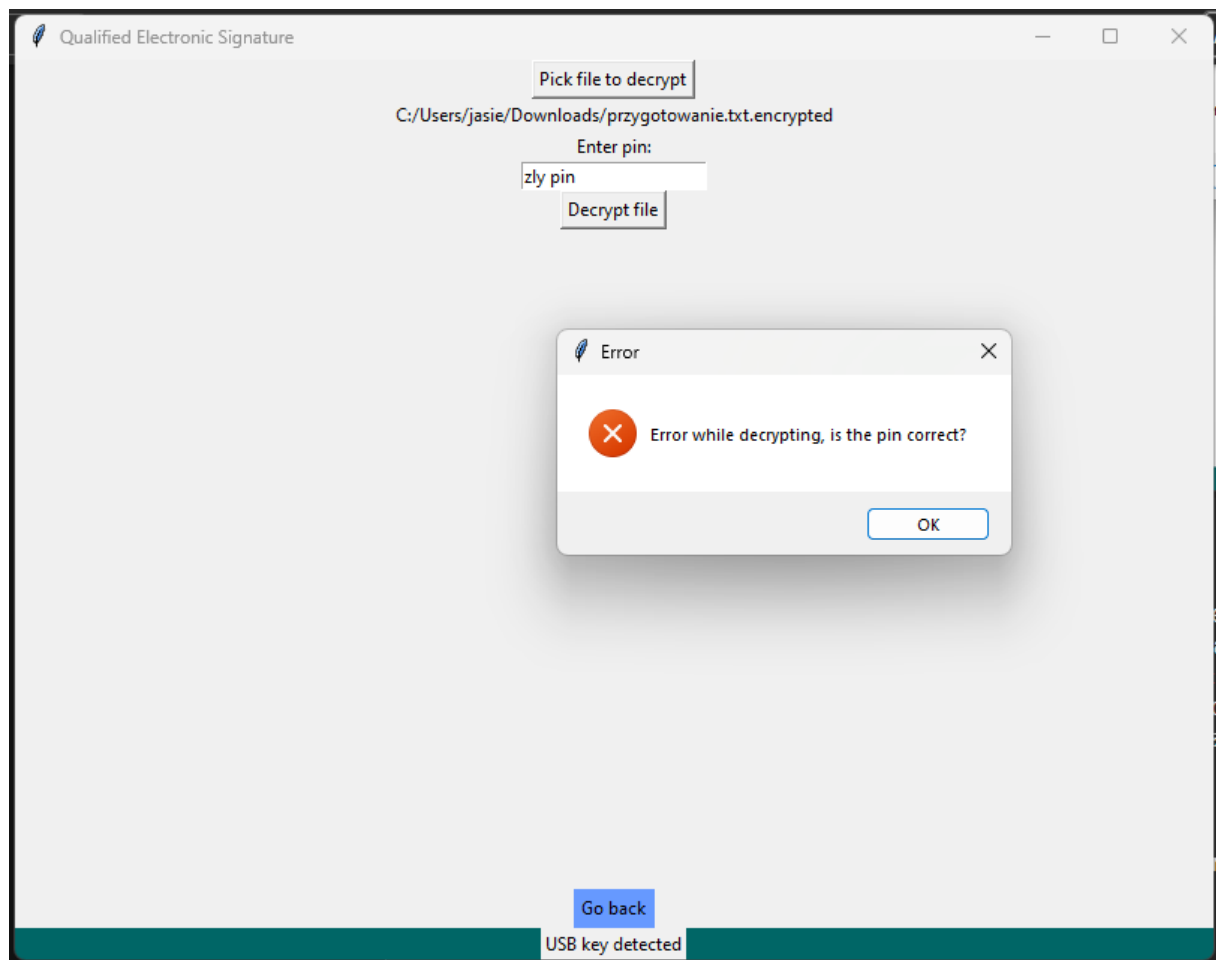
### **Deszyfrowanie pliku:**

Aby odszyfrować plik, najpierw należy wskazać poprawnie zaszyfrowany plik kluczem publicznym. Klucz prywatny aplikacja znajdzie sama, szukając go w tej samej lokalizacji co klucz publiczny. Należy również wpisać poprawne hasło do klucza prywatnego. Jeśli deszyfrowanie się powiedzie, plik pojawi się w tej samej lokalizacji co zaszyfrowany (końcówka “.encrypted” zostanie usunięta) oraz aplikacja wyświetli następujący komunikat:



Zdjęcie 15: Deszyfrowanie zakończone sukcesem

**Jeżeli podane zostanie hasło do klucza, aplikacja poinformuje użytkownika o błędzie:**



Zdjęcie 16: Deszyfrowanie zakończone niepowodzeniem

## 2.4 Summary

Obydwie aplikacje zostały w pełni zrealizowane i wyposażone we wszystkie funkcjonalności wymagane w kryteriach projektowych.

## 3. Literature

- [1] Dokumentacja tkinter <https://docs.python.org/3/library/tk.html>
- [2] Dokumentacja biblioteki cryptography <https://cryptography.io/en/latest/>