,

Bezpieczeństwo Systemów Komputerowych

Link do Github,
Adam Jakubowski 193352, Hubert Wajda 193511

Wersja 1.0

Spis treści

1. Opis projektu

Głównym celem projektu było opracowanie aplikacji umożliwiającej emulację kwalifikowanego podpisu elektronicznego zgodnego ze standardem PAdES (PDF Advanced Electronic Signature). Na potrzeby projektu stworzyliśmy dwie aplikacje:

- **Aplikacja pomocnicza** – generuje parę kluczy RSA oraz szyfruje klucz prywatny algorytmem AES, przy użyciu kodu PIN pobranego od użytkownika.
- **Aplikacja do podpisywania dokumentów** – realizujące podpisywanie dokumentów PDF oraz weryfikację poprawności podpisu z wykorzystaniem klucza publicznego.

2. Aplikacja pomocnicza

2.1. Opis ogólny

Aplikacja pomocnicza (Auxiliary App) jest narzędziem umożliwiającym generowanie kluczy RSA oraz ich zapis na wybranym urządzeniu USB. Głównym celem aplikacji jest zapewnienie bezpiecznego mechanizmu tworzenia i przechowywania kluczy RSA z wykorzystaniem algorytmu AES do ich szyfrowania. Aplikacja została zbudowana przy użyciu biblioteki Tkinter dla interfejsu użytkownika oraz pakietów kryptograficznych Crypto i cryptography dla operacji szyfrowania.

2.2. Struktura aplikacji

Aplikacja pomocnicza składa się z trzech głównych modułów:

- **main.py** – punkt wejścia do aplikacji.
- **gui.py** - odpowiada za interfejs użytkownika oraz interakcję z systemem plików.
- **util.py** - zawiera funkcje kryptograficzne do generowania kluczy i ich szyfrowania.

2.3. Funkcjonalność aplikacji pomocniczej

Aplikacja pomocnicza umożliwia użytkownikowi skorzystanie z następujących funkcji:

- Wprowadzenie 4-cyfrowego PINu, który będzie wykorzystywany do zabezpieczenia klucza prywatnego.
- Generowanie klucza RSA (4096-bitowego) oraz jego zapis na wybranym nośniku USB:
  - Klucz prywatny (zaszyfrowany przy użyciu AES z wykorzystaniem PINu)
  - Klucz publiczny jest zapisywany w formacie PEM.

2.4. Opis kluczowych Funkcjonalności

2.4.1. Generowanie klucza RSA

Funkcja generuje klucz prywatny RSA o długości 4096 bitów wykorzystując do tego bibliotekę cryptography.

```
private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=4096
    )
```

2.4.2. Serializacja klucza prywatnego i publicznego

Klucz prywatny oraz publiczny jest serializowany do formatu PEM, który jest standardowym formatem zapisu kluczy w kryptografii.

```
# Serialize the public key
    pem_public_key = private_key.public_key().public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
```

```
    # Serialize the private key
    pem_private_key = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )
```

2.4.3. Szyfrowanie klucza prywatnego

Klucz prywatny jest szyfrowany przy użyciu algorytmu AES z wykorzystaniem klucza (PINu) pochodzącego z hasła użytkownika. W tym celu skorzystaliśmy z biblioteki Crypto. Szczegółowy opis algorytmu:

- **Generowanie klucza AES** - Wprowadzony przez użytkownika PIN (4-cyfrowy) jest przekształcany na klucz AES poprzez funkcję haszującą SHA-256.
- **Tworzenie szyfratora** - Szyfrator jest tworzony w trybie ECB, który dzieli tekst na bloki o stałym rozmiarze (16 bajtów) i szyfruje każdy blok osobno. Tryb ECB jest prosty w implementacji, ale mniej bezpieczny niż inne tryby (np. CBC) ze względu na brak losowości.
- **Dodanie wypełnienia (padding)** - AES wymaga, aby dane były wielokrotnością rozmiaru bloku (16 bajtów). Funkcja pad() dodaje odpowiednią ilość danych wypełniających.
- **Szyfrowanie danych** - Funkcja encrypt() szyfruje przygotowane dane, zwracając zaszyfrowane bajty.
- **Kodowanie Base64** - Zaszyfrowane bajty są kodowane do formatu Base64, który jest wygodny do zapisu w pliku tekstowym.

```
aes_key = SHA256.new(pin.encode()).digest()
    cipher = AES.new(aes_key, AES.MODE_ECB)

    padded_private_key = pad(private_key, AES.block_size)

    ct_bytes = cipher.encrypt(padded_private_key)
    ct_base64 = base64.b64encode(ct_bytes).decode('utf-8')

    return ct_base64
```

3. Główna aplikacja

Aplikacja umożliwia odczyt klucza prywatnego z nosnika usb. Odszyfrowanie klucza prywatnego, a nastepnie podpisanie dokumentu PDF. Do odczytania klucza prywatnego wykorzystaliśmy bibliotekę **CryptoJS**.

```
async function encode_key(_event, pin) {
  return await load_data_from_pendrive().then((encrypted_key_base64) => {
    if (encrypted_key_base64 == null) {
      return { state: "error", message: "Key has not been found!", data: null };
    } else {
      var hash_pin = sha256(pin);
      const encrypted_key = CryptoJS.enc.Base64.parse(encrypted_key_base64);
      try {
        const decrypted = CryptoJS.AES.decrypt(
          { ciphertext: encrypted_key },
          hash_pin,
          {
            mode: CryptoJS.mode.ECB,
            padding: CryptoJS.pad.Pkcs7,
          }
```

```
        );
        const decryptedPem = decrypted.toString(CryptoJS.enc.Utf8);
        console.log("DECRYPTED PEM (UTF-8):", decryptedPem);
        private_rsa_key = decryptedPem;
        return { state: "success", message: "Key is loaded", data: null };
      } catch (err) {
        console.log(err);
        return { state: "error", message: "Invalid PIN", data: null };
      }
    }
  });
}
```

Weryfikacja podpisu z użyciem biblioteki **crypto**:

```
const public_rsa_key = await load_public_key();
const fileBuffer = Buffer.from(file);

const fileData = fileBuffer.subarray(0, -SIGNATURE_LENGTH);
const signature = fileBuffer.subarray(-SIGNATURE_LENGTH);

const verify = crypto.createVerify("SHA256");
verify.update(fileData);
verify.end();
```

Podpisywanie pliku PDF z użyciem biblioteki **crypto**:

```
const fileBuffer = Buffer.from(file);
const sign = crypto.createSign("SHA256");
sign.update(fileBuffer);
sign.end();

const signature = sign.sign(private_rsa_key);
fs.writeFileSync(
  path.join(__dirname, "../signed_.pdf"),
  Buffer.concat([fileBuffer, signature])
);
```

4. Wykorzystane technologie

4.1. Język programowania

- Aplikacja pomocnicza została napisana w języku Python, ze względu na bogatą ofertę bibliotek kryptograficznych oraz prostotę implementacji algorytmów kryptograficznych.
- Aplikacja do podpisywania dokumentów została napisana w języku JavaScript z wykorzystaniem frameworka Electron, który umożliwia tworzenie aplikacji desktopowych przy użyciu technologii webowych (HTML, CSS, JavaScript). Dzięki temu możliwe jest łatwe tworzenie interfejsu użytkownika oraz integracja z systemem operacyjnym.

4.2. Biblioteki kryptograficzne

- Do generowania kluczy RSA oraz szyfrowania klucza prywatnego wykorzystaliśmy bibliotekę cryptography, która zapewnia interfejs do biblioteki OpenSSL. Do szyfrowania klucza prywatnego z wykorzystaniem algorytmu AES skorzystaliśmy z biblioteki Crypto.
- Do podpisywania dokumentów PDF oraz weryfikacji podpisu wykorzystaliśmy bibliotekę crypto, która jest standardową biblioteką kryptograficzną dla języka JavaScript.

4.3. Interfejs użytkownika

- Do stworzenia interfejsu użytkownika wykorzystaliśmy bibliotekę Tkinter, która jest standardowym interfejsem graficznym dla języka Python.
- Interfejs głównej aplikacji został stworzony przy użyciu biblioteki React.

4.4. Obsługa nośnika USB

- Do obsługi nośnika USB wykorzystaliśmy bibliotekę psutil służąca do zarządzania procesami oraz dostępem do informacji o urządzeniach systemowych, w tym nośnikach USB.
- Do odczytu i zapisu danych na nośniku USB wykorzystaliśmy drivelist.

4.5. System kontroli wersji

Do zarządzania kodem źródłowym wykorzystaliśmy system kontroli wersji Git oraz platformę GitHub do przechowywania kodu źródłowego.

4.6. Dokumentacja

Do stworzenia dokumentacji skorzystaliśmy z Doxygen, który umożliwia generowanie dokumentacji z komentarzy w kodzie źródłowym.

5. Interfejs graficzny aplikacji pomocniczej

5.1. Główne okno aplikacji

Po uruchomieniu aplikacji użytkownik zostaje przywitany przez główne okno aplikacji, które zawiera pola do wprowadzenia PINu oraz przyciski do generowania klucza oraz zapisu klucza na nośniku USB.

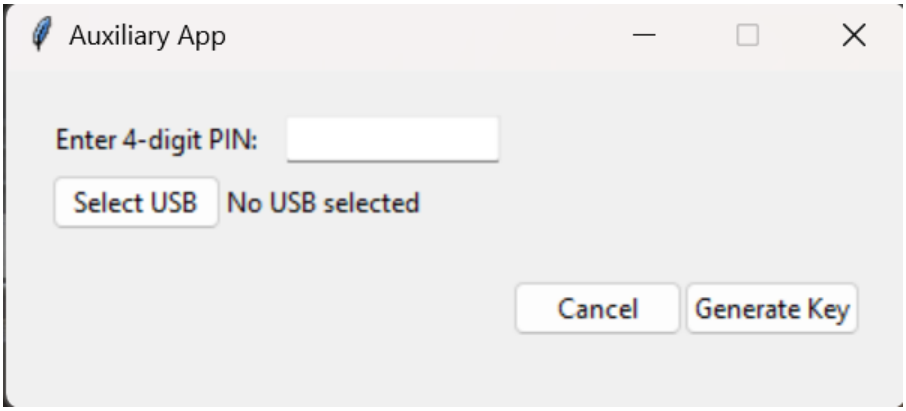

Figure 1: Główne okno aplikacji

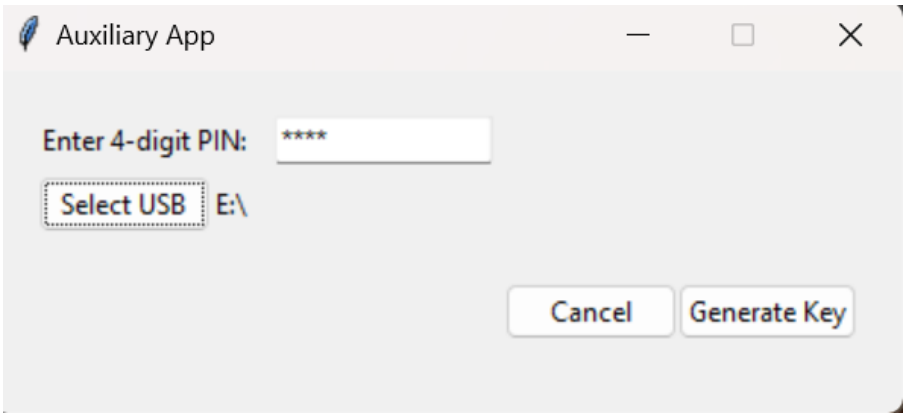

Figure 2: Główne okno aplikacji z wybranymi danymi

5.2. Okno wyboru nośnika USB

Po naciśnięciu przycisku „Select USB” użytkownik zostaje poproszony o wybranie nośnika USB, na który zostanie zapisany klucz prywatny.

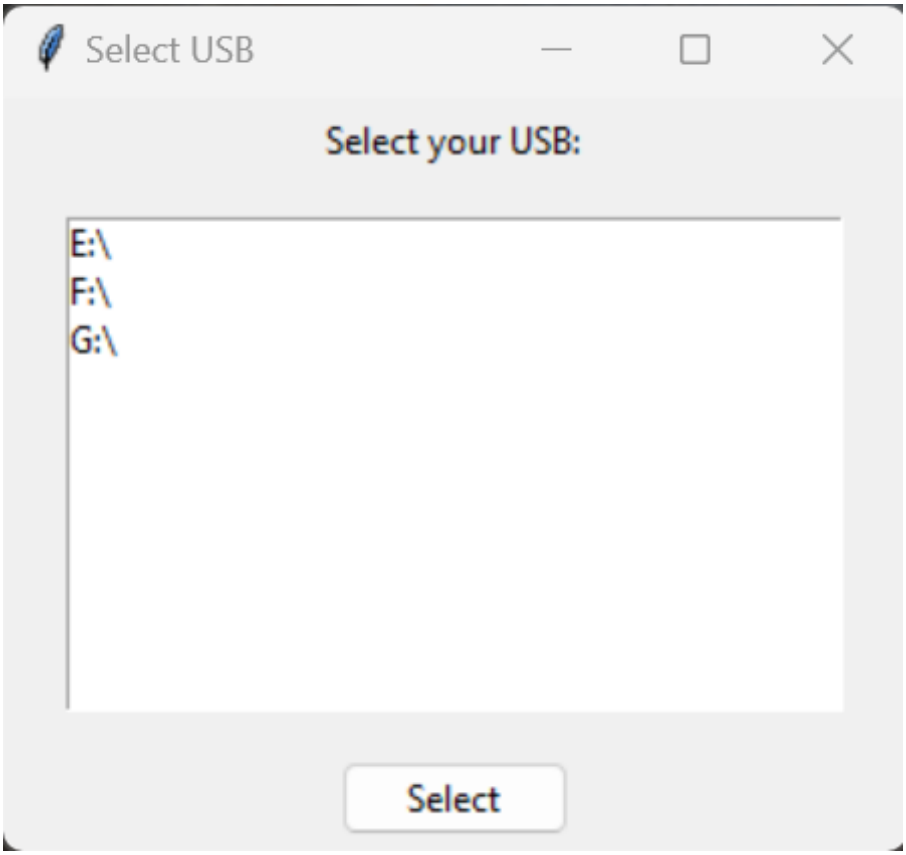

Figure 3: Okno wyboru nośnika USB

5.3. Okno informacyjne

5.3.1. Okno informaujące o sukcesie

Po poprawnym zapisaniu klucza prywatnego na nośniku USB użytkownik zostaje poinformowany o sukcesie operacji.

,

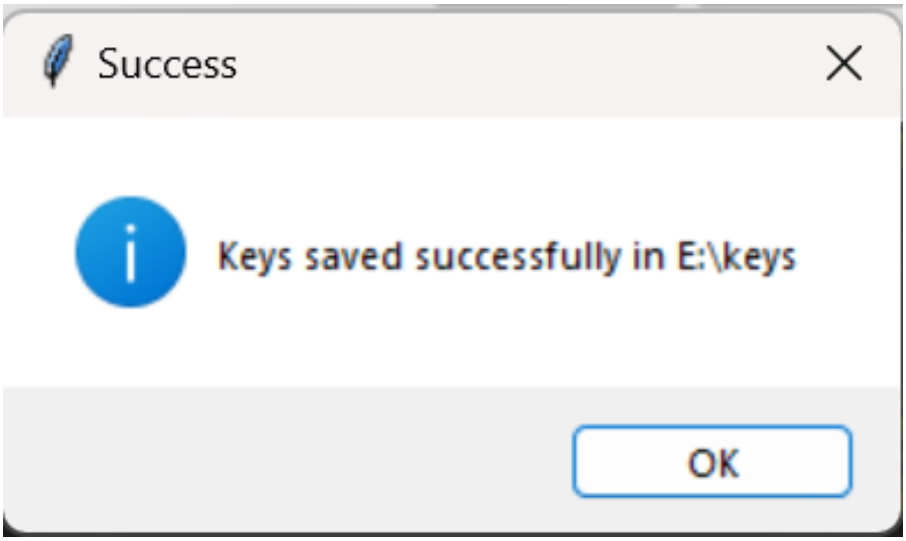

Figure 4: Okno informujące o sukcesie

5.3.2. Okno informujące o błędzie

W przypadku błędu podczas zapisu klucza prywatnego użytkownik zostaje poinformowany o błędzie.

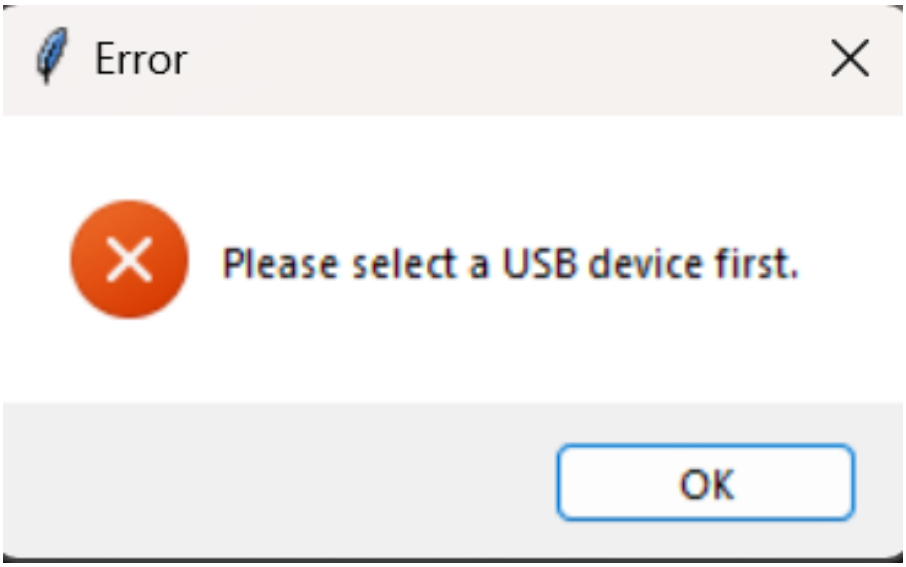

Figure 5: Okno informujące o błędzie

6. Interfejs graficzny aplikacji głównej

6.1. Podpisywanie dokumentów

W celu podpisania dokumentu, użytkownik najpierw musi wprowadzić poprawny PIN. Aplikacja informuje w przypadku błędnego pinu lub braku nośnika USB.

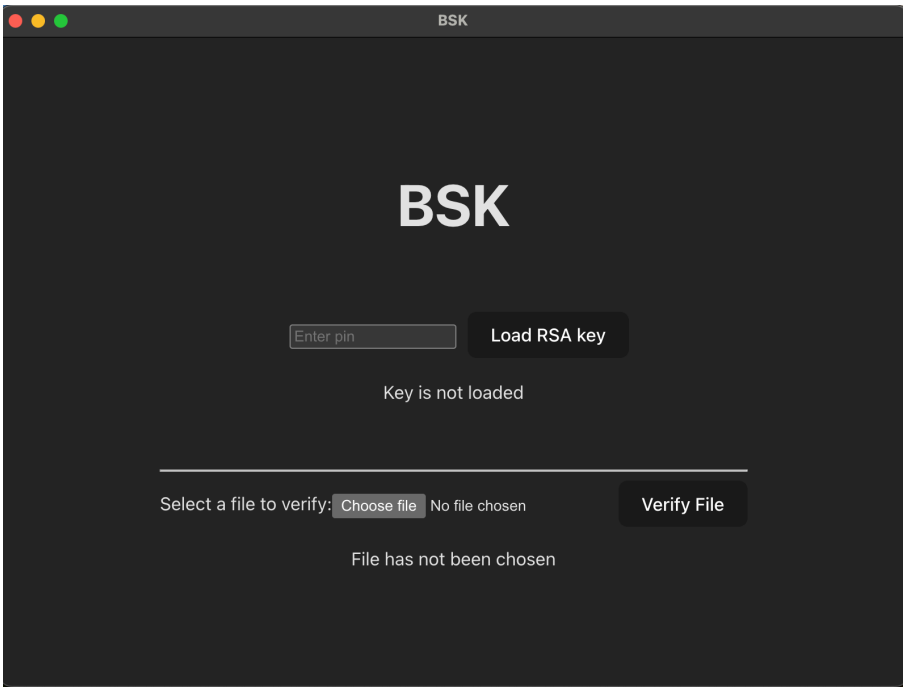

Figure 6: Główne okno aplikacji

Po poprawnym wprowadzeniu PINu użytkownik może wybrać plik PDF, który chce podpisać. Następnie aplikacja generuje podpis elektroniczny i zapisuje go w pliku PDF w głównym folderze projektu. Użytkownik zostaje poinformowany o sukcesie operacji.

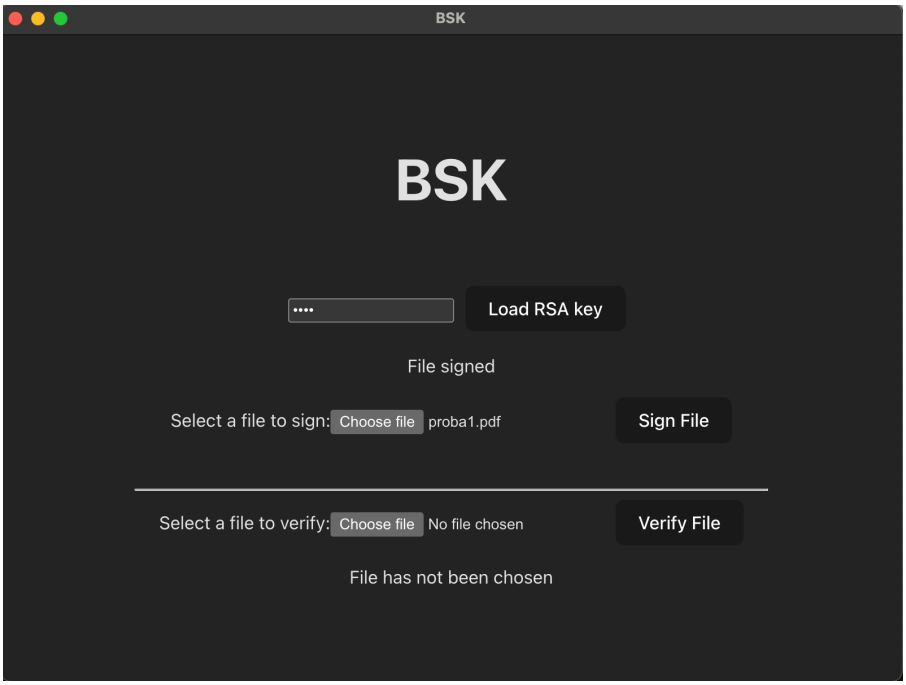

Figure 7: Wprowadzenie poprawnego PINU

6.2. Weryfikacja podpisu

Aplikacja umożliwia również weryfikację podpisu elektronicznego. Użytkownik musi wybrać plik PDF, który chce zweryfikować. Aplikacja informuje użytkownika o poprawności lub błędzie podpisu.

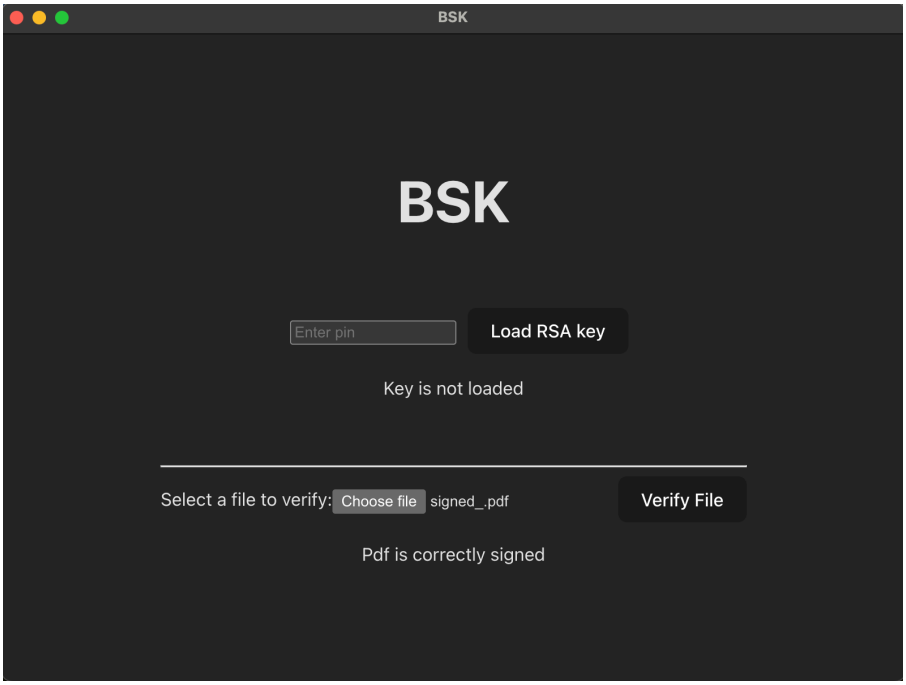

Figure 8: Weryfikacja podpisu

,

7. Dokumentacja

7.1. Opis

Dokumentacja została dodana przy wykorzystaniu Doxygen. Jest to narzędzie, które umożliwia generowanie dokumentacji z komentarzy w kodzie źródłowym.

7.2. Generowanie dokumentacji

Aby wygenerować dokumentację należy wpisać komendę: doxygen -g Doxyfile. Następnie należy zmodyfikować plik Doxyfile, ustawiając odpowiednie parametry. Aby wygenerować dokumentację należy wpisać komendę: doxygen Doxyfile.

7.3. Przykładowe wykorzystanie

Krótki opis funkcji służącej do szyfrowania klucza prywatnego może wyglądać następująco:

```
"""!
    @brief Szyfruje klucz prywatny za pomocą AES-256 (PIN jako klucz)
    @param private_key Klucz prywatny w formacie PEM
    @param pin 4-cyfrowy PIN użytkownika
    @return Klucz prywatny zaszyfrowany w formacie Base64
    """
```

@brief - opisuje funkcję
@param - opisuje parametry funkcji
@return - opisuje wartość zwracaną przez funkcję
@note - dodatkowe informacje

8. Podsumowanie

W ramach projektu udało nam się zaimplementować aplikację pomocniczą umożliwiającą generowanie kluczy RSA oraz szyfrowanie klucza prywatnego z wykorzystaniem algorytmu AES. Aplikacja została zbudowana w języku Python przy użyciu bibliotek kryptograficznych cryptography i Crypto. Dzięki zastosowaniu interfejsu Tkinter użytkownik może w prosty sposób generować klucze oraz zapisywać je na nośniku USB.

9. Link do Github

https://github.com/WajHub/BSK/tree/main

10. Literatura

- https://nodejs.org/api/crypto.html
- https://www.electronjs.org/docs/latest
- Generowanie kluczy RSA
- Szyfrowanie za pomocą AES-256
- Dokumentacja Tkinter
- Dokumentacja cryptography
- Dokumentacja Crypto
- Dokumentacja psutil