Security of Computer Systems

Project Report

Authors:
Adrian Zdankowski 193480
Arkadiusz Flisikowski 193496

Version: 1.0

Versions

Version	Date	Description of changes
1.0	2025-04-03	Creation of the document

1. Project – control and final term

1.1 Description

The aim of the project is to realise a software tool for emulating the qualified electronic signature in accordance with the PAdES (PDF Advanced Electronic Signature) format. The primary task of the project is to design and develop an application that will allow users to digitally sign PDF documents using RSA encryption. The private key is stored on a USB drive encrypted by the AES algorithm.

1.2 Results

1.2.1 Auxiliary application

- ➢ **RSA Key Generation**
  - Generation of RSA private and public keys.
  - Encryption of the private key using an 8-digit PIN provided by the user, secured with the AES algorithm.
- ➢ **Secure Storage of RSA Keys on a USB Drive**
  - Storing the encrypted RSA private key on a USB drive.
- ➢ **Graphical User Interface (GUI)**
  - Detection of the connected USB drive.
  - User input for the 8-digit PIN.
  - Selection of the path for saving the public key.
  - Generation of RSA keys through the user interface.
  - Displaying progress and status updates using message boxes.

1.2.2 Main application

- ➢ **Automatic Detection of the RSA Private Key**
  - Automatic retrieval of the RSA private key from a connected USB drive.
- ➢ **Document Signing In Accordance with PAdES**

- Hashing the PDF file based on its content.
- Decrypting the RSA private key using the AES algorithm and the user-provided PIN.
- Creating a digital signature using the generated hash and decrypted private key.
- Embedding the signature into the PDF file.

➢ **Signature Verification**

- Extracting the signature from the PDF file.
- Hashing the PDF file based on its content.
- Verifying the signature using the RSA public key.

➢ **Graphical User Interface (GUI)**

- User input for the 8-digit PIN.
- Selection of the path to the public key for signature verification.
- Selection of the PDF file to sign or verify.
- Signing the PDF file through the user interface.
- Verifying the PDF file through the user interface.
- Displaying progress and status updates using message boxes.

1.2.3 Documentation and Code Repository

- The project documentation was created using Doxygen to provide a well-structured and detailed reference.
- GitHub was used for version control and collaboration.
- Repository: https://github.com/AdrianZdankowski/pades-emulation-tool.git

1.2.4 Testing

- Verification of the GUI functionality in both applications.
- Generation of RSA private and public keys.

- Secure storage of the RSA private key on a USB drive.
- Execution of the PDF signing process.
- Validation of the digital signature verification process.

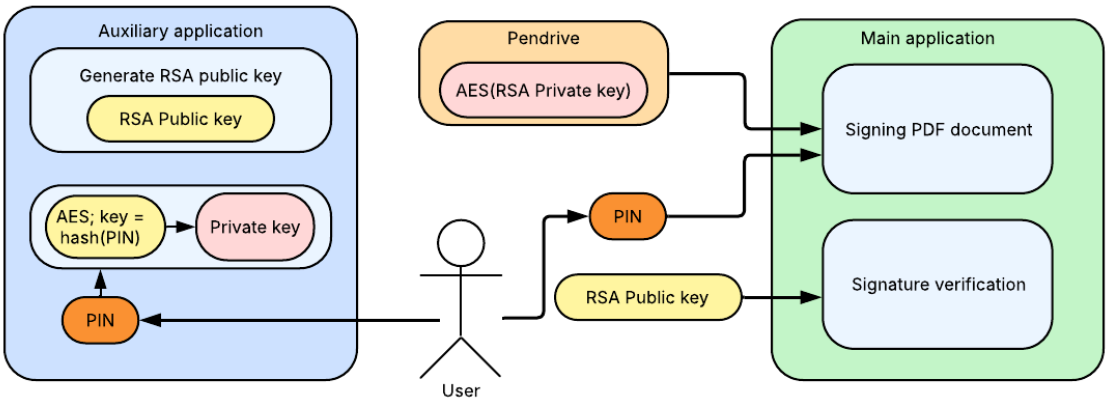

Fig.1 Block diagram of the project overview.

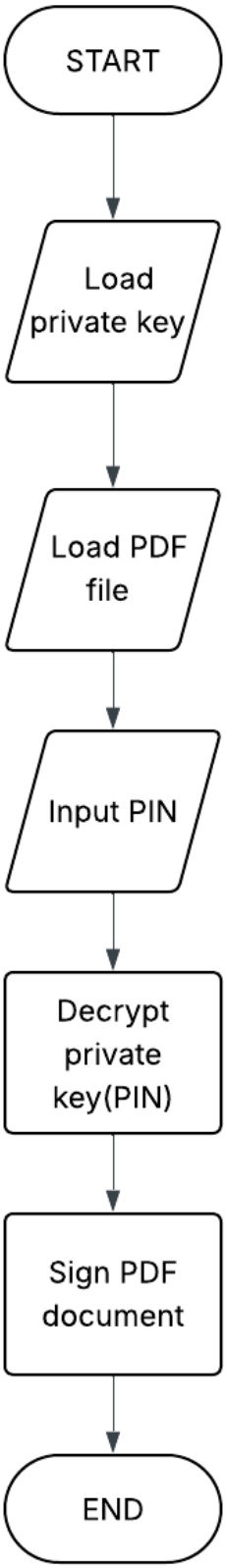

Fig 2. Flow diagram of the PDF document signing process.

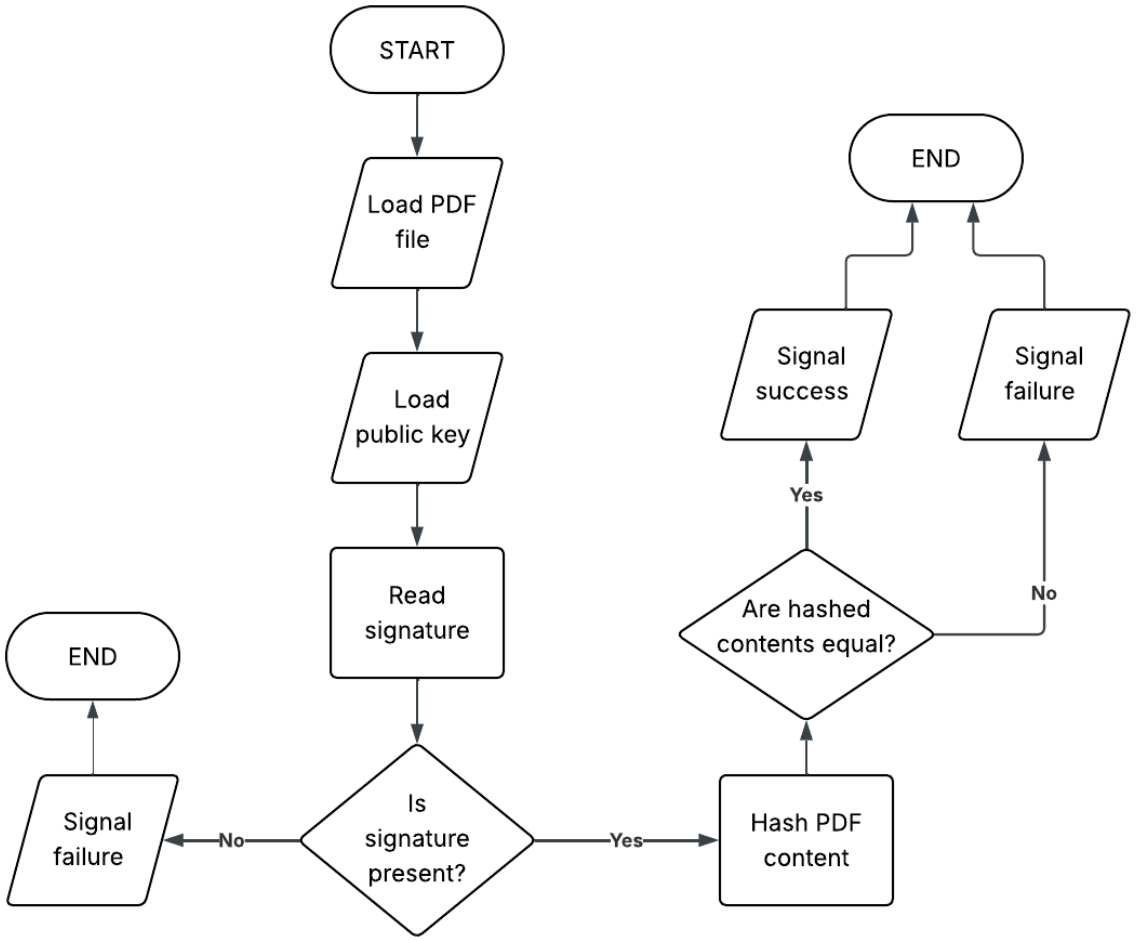

Fig. 3 Flow diagram of the signature verification process.

1.3 Code description

1.3.1 Auxiliary application - delivers functions for generating a pair of RSA (4096) keys. The private key is encrypted using an AES cipher with a 256 bit hash from an 8 digit PIN given by the user.

- **RSA Key Generation -** generates a private and public RSA key pair, encrypts the private key using a PIN, and saves both keys in specified locations.

```
def GenerateAndSaveKeys(gui, pin, pendrive, publicKeyPath, DELAY_IN_MS):
    try:
        gui.after(0, gui.setGenerationStatusLabelText, "Generating PIN hash...")
        pseudoRandomGenerator = get_random_bytes
        hashedPin = sha256(pin.encode()).digest()

        gui.after(DELAY_IN_MS, gui.setGenerationStatusLabelText, "Generating private and public key...")
        key = RSA.generate(4096, randfunc=pseudoRandomGenerator)
        privateKey = key.export_key()
        publicKey = key.public_key().export_key()

        gui.after(DELAY_IN_MS*2, gui.setGenerationStatusLabelText, "Encrypting private key...")
        cipher = AES.new(hashedPin, AES.MODE_CBC)
        iv = cipher.iv
        encryptedPrivateKey = iv + cipher.encrypt(pad(privateKey, AES.block_size))

        keyFolder = os.path.join(pendrive,"Key")
        os.makedirs(keyFolder,exist_ok=True)

        privateKeyPath = os.path.join(keyFolder, "encrypted_PK.bin")
        publicKeyPath = os.path.join(publicKeyPath, "public_key.pem")

        gui.after(DELAY_IN_MS*3, gui.setGenerationStatusLabelText, "Saving private key...")
        with open(privateKeyPath, "wb") as f:
            f.write(encryptedPrivateKey)

        gui.after(DELAY_IN_MS*4, gui.setGenerationStatusLabelText, "Saving public key...")
        with open(publicKeyPath, "wb") as f:
            f.write(publicKey)

        gui.after(DELAY_IN_MS*5, gui.setGenerationStatusLabelText, "Generation and saving succeeded!")
        return True
    except Exception as e:
        gui.after(0, gui.setGenerationStatusLabelText, e)
        return False
```

1.3.2 Main application - delivers functions for signing PDF documents using an RSA (4096) private key. Additionally, it enables users to verify PDF documents using an RSA (4096-bit) public key and the embedded digital signature. The digital signature is generated by hashing the contents of the PDF document pages.

- **Document Signing in accordance to PAdES -** signs a PDF document and embeds the digital signature in its metadata. The digital signature is generated by hashing the contents of the PDF document pages.

```python
def signPDF(gui, privateKey, pin, pdfPath):
    try:
        base, ext = os.path.splitext(pdfPath)
        signedPdfPath = f"{base}-signed.pdf"
        decryptedPrivateKey = RSA.import_key(decryptPrivateKey(pin, privateKey))

        gui.setLabelText("Opening the PDF file ...", INFO_ABOUT_SIGNING_DOCUMENT, "white")
        with open(pdfPath, "rb") as f:
            pdfReader = PdfReader(f)
            pdfWriter = PdfWriter()

            gui.setLabelText("Generating hash of the PDF file ...", INFO_ABOUT_SIGNING_DOCUMENT)
            pdfHash = createHash(pdfReader)

            gui.setLabelText("Creating a digital signature ...", INFO_ABOUT_SIGNING_DOCUMENT)
            signature = pkcs1_15.new(decryptedPrivateKey).sign(pdfHash)

            gui.setLabelText("Adding a signature to the PDF file ...", INFO_ABOUT_SIGNING_DOCUMENT)
            for page in pdfReader.pages:
                pdfWriter.add_page(page)

            pdfWriter.add_metadata({
                "/Signature": signature
            })

            gui.setLabelText("Saving the signed PDF file ...", INFO_ABOUT_SIGNING_DOCUMENT)
            with open(signedPdfPath, "wb") as signedPDF:
                pdfWriter.write(signedPDF)

            return True
    except Exception as e:
        gui.setLabelText(e, INFO_ABOUT_SIGNING_DOCUMENT)
        return False
```

- **Signature Verification -** Checks whether the digital signature in the PDF document is valid. The user's RSA (4096-bit) public key is required during this

process.

```
def verifyPdfSignatureFromMetadata(gui, pdfPath, publicKeyPath):
    try:
        gui.setLabelText("Reading the public key from file ...", INFO_ABOUT_VERIFYING_DOCUMENT)
        with open(publicKeyPath, "rb") as pub_file:
            public_key = RSA.import_key(pub_file.read())

        gui.setLabelText("Reading the PDF file ...", INFO_ABOUT_VERIFYING_DOCUMENT)
        with open(pdfPath, "rb") as f:
            pdfReader = PdfReader(f)
            metadata = pdfReader.metadata

            gui.setLabelText("Checking if signature exists in metadata ...", INFO_ABOUT_VERIFYING_DOCUMENT)
            if "/Signature" not in metadata:
                return NO_SIGNATURE

            signature = metadata["/Signature"]

            gui.setLabelText("Generating hash of the PDF file ...", INFO_ABOUT_VERIFYING_DOCUMENT)
            pdfHash = createHash(pdfReader)

            gui.setLabelText("Verifying the signature ...", INFO_ABOUT_VERIFYING_DOCUMENT)
            try:
                pkcs1_15.new(public_key).verify(pdfHash, signature)
                return OK
            except (ValueError, TypeError) as e:
                return SIGNATURE_VERIFICATION_VALUE_ERROR
    except Exception as e:
        gui.setLabelText(e, INFO_ABOUT_VERIFYING_DOCUMENT)
        return UNKNOWN_ERROR
```

- **Hashing the PDF File -** generates a hash for the digital signature by hashing the contents of the PDF document pages.

```
def createHash(pdfReader):
    hash = SHA256.new()
    for page in pdfReader.pages:
        hash.update(page.extract_text().encode())
    return hash
```

- **Decryption of the RSA Private Key -** decrypts an RSA (4096-bit) private key using the AES algorithm.

```
def decryptPrivateKey(pin, privateKey):
    iv = privateKey[:16]
    cipherText = privateKey[16:]

    hashedPin = sha256(pin.encode()).digest()

    cipher = AES.new(hashedPin, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(cipherText), AES.block_size)
```

1.3.3 Utils functions - delivers common functions used across the project.

- **Searching Connected Pendrives -** searches for paths of the connected pendrives.

```python
def searchForPendrive():
    pendrives = []
    for partition in psutil.disk_partitions(all=False):
        if 'removable' in partition.opts.lower():
            pendrives.append(partition.device)
    return pendrives
```

- **Searching and Reading the Private Key** - searches for a file containing the user's private key on mounted pendrives and reads its contents.

```python
def searchAndReadPrivateKey(pendrives, filePath):
    for pendrive in pendrives:

        fullFilePath = os.path.join(pendrive, filePath)
        if os.path.exists(fullFilePath):
            try:
                with open(fullFilePath, 'rb') as f:
                    privateKey = f.read()
                    return privateKey
            except Exception as e:
                print(f"Error during opening the file: {e}")
                return "Not found"

    return "Not found"
```

1.4 Summary

This project provides a secure solution for digitally signing and verifying PDF documents using the PAdES standard. It involves generating RSA key pairs, encrypting the private key with an 8-digit PIN, and securely storing the keys on a USB drive. The main application signs and verifies PDF files by decrypting the private key, creating a digital signature, and validating it with the public key. The user interacts with a simple GUI, which also provides progress updates via message boxes. The project uses Doxygen for documentation and GitHub for version control.

2. Literature

[1] CustomTkinter: https://customtkinter.tomschimansky.com/documentation/

[2] Online Doxygen: https://www.doxygen.nl/manual/lists.html

[3] PyCryptodome: https://pycryptodome.readthedocs.io/en/latest/index.html

[4] PyPDF2: https://pypdf2.readthedocs.io/en/3.x/

[5] hashlib: https://docs.python.org/3/library/hashlib.html

[6] RSA algorithm: https://en.wikipedia.org/wiki/RSA_cryptosystem

[7] AES algorithm: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard