Security of Computer Systems

Project Report

Authors:
Jakub, Falk, 193252
Lucjan, Gackowski, 193150

Version: 1.2

Versions

Version	Date	Description of changes
1.0	04.04.2025	Creation of the document
1.1	07.04.2025	Added github repository link in Description section and provided Performed Tests section
1.2	14.05.2025	Provided additional documentation concerning separated architecture

1. Project – control and final term

1.1.Description

The PAdES Signature Application is a comprehensive tool that enables users to create and verify PDF Advanced Electronic Signatures with strong cryptographic security. It implements a complete workflow for generating RSA key pairs, securely storing the private key on USB storage devices protected by PIN-based AES encryption, and signing PDF documents in compliance with industry standards. The application features an intuitive graphical user interface with clear visual indicators for system status, allowing users to easily manage the entire signature process from key generation to signature verification.

The application is separated into two distinct components (processes): an **Auxiliary App**, responsible for key generation and encrypted storage management, and a **Main (Signature) App**, which handles document signing and signature verification. This modular design enhances both security and maintainability by isolating cryptographic key handling from signature operations.

Project's GitHub repository link.

1.2.Results

- **Key Management**
  - Implemented 4096-bit RSA key generation
  - Private keys are encrypted with AES using PIN-based security
  - Keys are properly stored (private on USB, public locally)

- **Document Operations**
  - PDF signing with proper document hashing
  - Signature verification functionality
  - Clear visual feedback during operations

- **User Interface**
  - Multi-panel design (keygen, sign, verify)
  - Real-time USB detection and status display
  - Secure PIN entry with visual keypad

- **Application Architecture**
  - Application separated into two independent components:
    - **Auxiliary App** – handles RSA key generation and encrypted key storage
    - **Main (Signature) App** – responsible for signing documents and verifying signatures
  - Multi-threaded design for responsive UI
  - Modular code structure with clear separation of concerns
  - Comprehensive error handling

- **Documentation**
  - Complete Doxygen documentation
  - Automated documentation generation via GitHub Actions
  - Code properly documented with standard tags

1.3.Code description

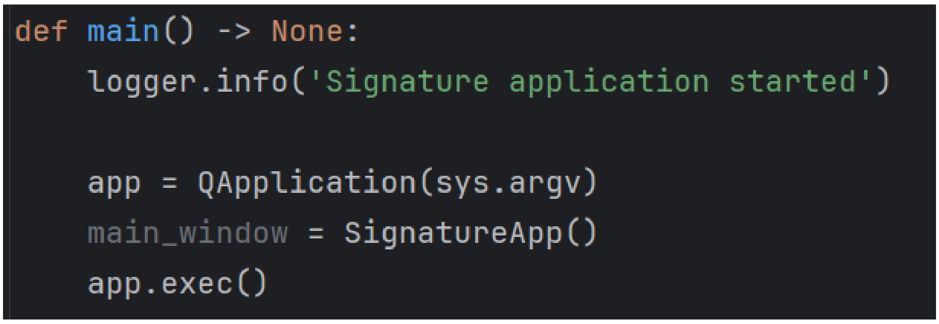
Architecture Overview

The PAdES Signature Application follows a modular architecture with clear separation of concerns. The codebase is organized into three primary components:

- **GUI Layer**: Contains the user interface implementation using PyQt6
- **Utility Layer**: Implements core cryptographic and file operations
- **Logger**: Handles application-wide logging

Key Components

- **Main (Signature) Application entrypoint (**main.py**) -** The entry point of the signature application initializes the logger and launches the window:



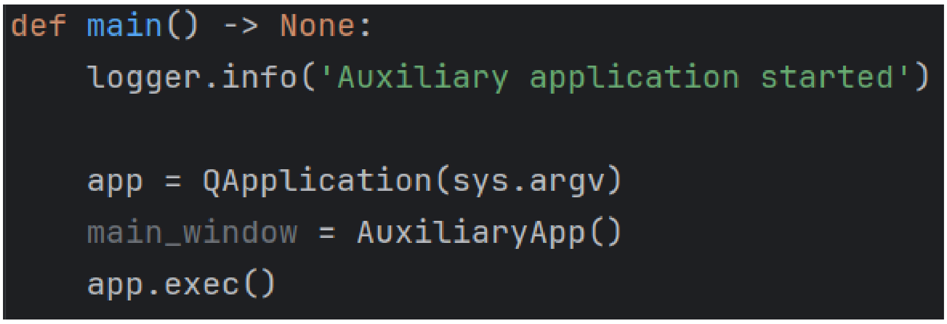
```
def main() -> None:
    logger.info('Signature application started')

    app = QApplication(sys.argv)
    main_window = SignatureApp()
    app.exec()
```

- **Signature Application (**SignatureApp.py**) -** The signature application window manages two key pages:

    - Document Signing Page
    - Signature Verification Page

- **Auxiliary application entrypoint (**main.py**) -** The entry point of the auxiliary application initializes the logger and launches the window:

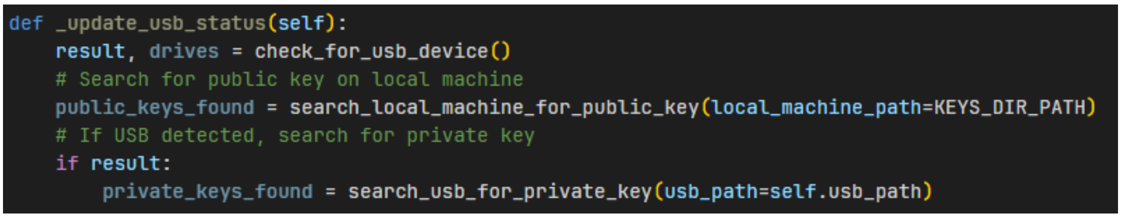


```
def main() -> None:
    logger.info('Auxiliary application started')

    app = QApplication(sys.argv)
    main_window = AuxiliaryApp()
    app.exec()
```

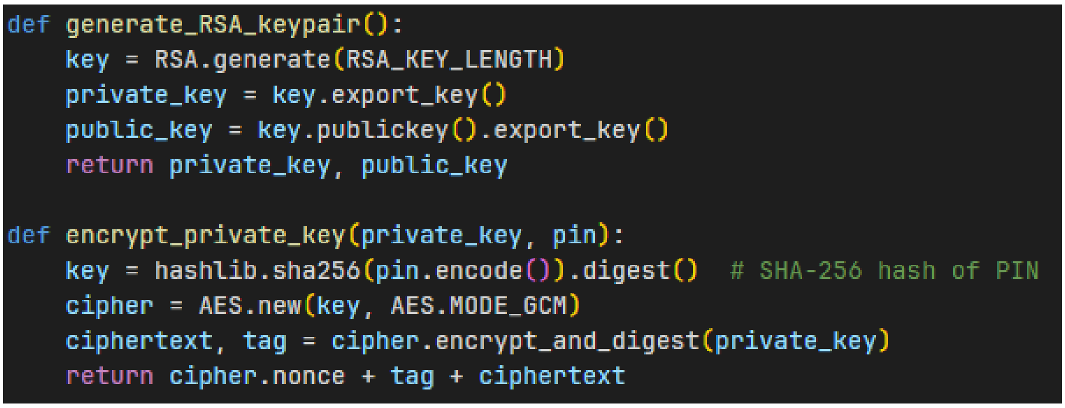
- **Auxiliary Application (**AuxiliaryApp.py**) -** The auxiliary application window manages one page:
    - Key Generation Page

These applications also handle USB detection, key discovery, and page navigation:



```
def _update_usb_status(self):
    result, drives = check_for_usb_device()
    # Search for public key on local machine
    public_keys_found = search_local_machine_for_public_key(local_machine_path=KEYS_DIR_PATH)
    # If USB detected, search for private key
    if result:
        private_keys_found = search_usb_for_private_key(usb_path=self.usb_path)
```

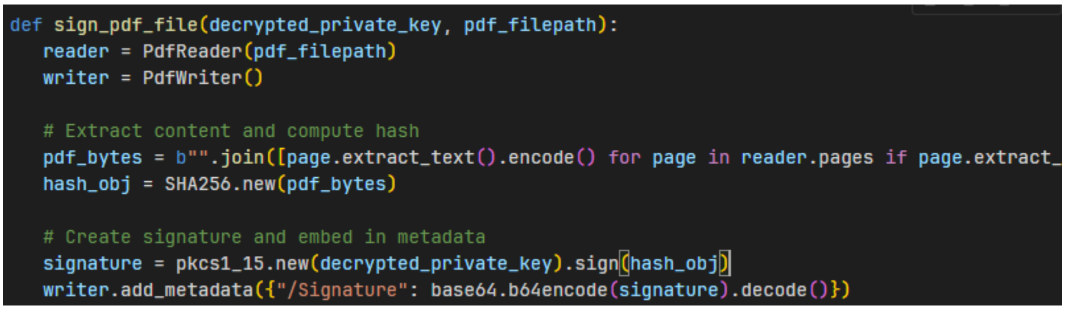
- **Cryptographic Operations (keygen.py, pdf_sign.py) -** The core security operations are implemented in specialized modules:



```
def generate_RSA_keypair():
    key = RSA.generate(RSA_KEY_LENGTH)
    private_key = key.export_key()
    public_key = key.publickey().export_key()
    return private_key, public_key

def encrypt_private_key(private_key, pin):
    key = hashlib.sha256(pin.encode()).digest()  # SHA-256 hash of PIN
    cipher = AES.new(key, AES.MODE_GCM)
    ciphertext, tag = cipher.encrypt_and_digest(private_key)
    return cipher.nonce + tag + ciphertext
```

- **PDF Operations** (pdf_sign.py) **-** The PDF signing and verification uses PyPDF2 to handle the PDF files, with custom signature embedding and extraction:

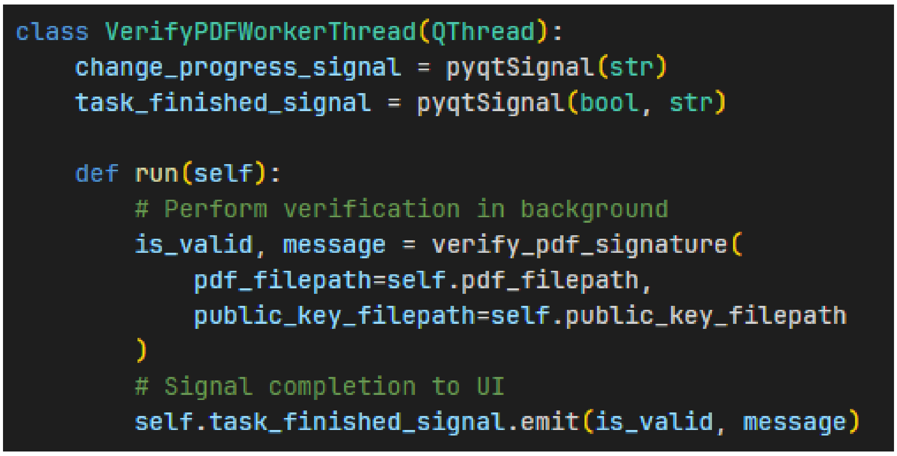


```
def sign_pdf_file(decrypted_private_key, pdf_filepath):
    reader = PdfReader(pdf_filepath)
    writer = PdfWriter()

    # Extract content and compute hash
    pdf_bytes = b"".join([page.extract_text().encode() for page in reader.pages if page.extract_
    hash_obj = SHA256.new(pdf_bytes)

    # Create signature and embed in metadata
    signature = pkcs1_15.new(decrypted_private_key).sign(hash_obj)
    writer.add_metadata({"/Signature": base64.b64encode(signature).decode()})
```

- **Worker Threads (**RSAWorkerThread.py, PDFWorkerThread.py, VerifyPDFWorkerThread.py**) -** Long-running operations are executed in background threads to keep the UI responsive:



```
class VerifyPDFWorkerThread(QThread):
    change_progress_signal = pyqtSignal(str)
    task_finished_signal = pyqtSignal(bool, str)

    def run(self):
        # Perform verification in background
        is_valid, message = verify_pdf_signature(
            pdf_filepath=self.pdf_filepath,
            public_key_filepath=self.public_key_filepath
        )
        # Signal completion to UI
        self.task_finished_signal.emit(is_valid, message)
```

1.4. Performed Tests Summary

- Key generation with RSA 4096-bit keys
- Private key encryption with user-defined PIN
- Attempting private key encryption with missing or invalid PIN
- Signing a PDF document using a valid private key with correct PIN provided
- Attempting signing a PDF document using a valid private key with incorrect PIN provided
- Attempting signing a PDF document using invalid or missing private key
- Attempting signing a PDF document which already has been signed
- Verifying a signed PDF with the correct public key
- Attempting verifying a signed PDF with the incorrect public key
- Attempting verifying a missing/corrupted PDF document
- Detecting changes in a tampered signed document
- Handling incorrect PIN entry during signing
- USB detection and key file recognition
- PIN input using an on-screen virtual keypad
- Maintaining UI responsiveness during cryptographic operations

1.5.Summary

The PAdES Signature Application successfully implements all required functionality for creating and verifying PDF Advanced Electronic Signatures. The application features robust key management with 4096-bit RSA key pairs, PIN-protected private keys on USB devices, and local public key storage. Document operations include PDF signing and verification with proper security measures. The implementation uses a user-friendly interface with clear status indicators and multi-threaded architecture for responsive operation. The application is fully documented with Doxygen comments and includes GitHub Actions for automatic documentation generation.

2. Literature

[1] Doxygen docs, https://www.doxygen.nl/manual/index.html

[2] Wikipedia Digital Signature, https://en.wikipedia.org/wiki/Digital_signature

[3] Python docs, https://docs.python.org/3/

[4] GitHub Actions docs, https://docs.github.com/en/actions