Tool for Emulating the PAdES Qualified Electronic Signature

Security of Computer Systems

Project Report

Authors:
Bartłomiej, Krawisz, 193319
Stanisław, Nieradko, 193044

Version: 1.0

Versions

Version	Date	Description of changes
1.0	26.03.2025	Creation of the document

1. Project

1.1 Project's description

The project's goal was to create tool to sign PDF documents to detect any attempts in changing its content. The project was supposed to fully emulate the process, including the hardware toll needed for person identification utilising USB drive for that purpose.

1.2 Results

The project was completed using python in a form of three separate applications: generate.py, sign.py and verify.py. Each application is responsible for different part of the project's requirements, utilizing common modules for configuration, cryptographic operations and integrations with operating system. The application's is based on the concept diagram below:

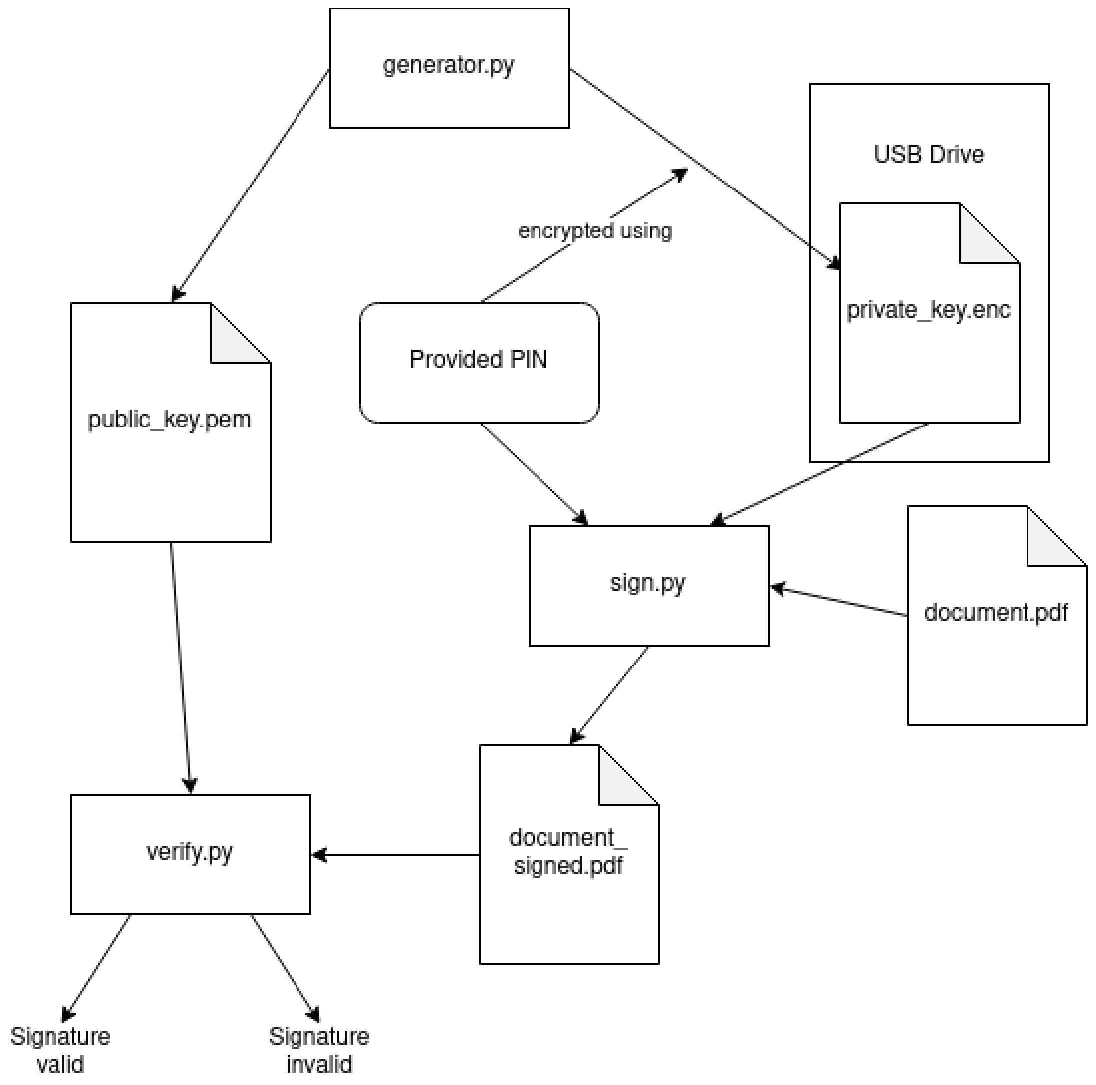


Fig. 1 – Application's concept diagram.

Application is created on and intended to run on Linux based operating systems, due to the way in which it looks up any new USB drivers connected to the system:

```
## @package utils.usb_utils
# USB drive detection utility functions
#
# This module provides utility functions to detect USB drives with specific files.

import getpass
import os
import threading
import time
from typing import Callable


## Detect a USB drive with a specific file
# @param filename Name of the file to detect
# @return Path of the USB drive if found, otherwise None
def detect_usb_drive_with_file(filename: str):
    base_paths = ["/media/", f"/run/media/{getpass.getuser()}/"]

    for base_path in base_paths:
        if os.path.exists(base_path):
            # List directories in the base path
            for subdir in os.listdir(base_path):
                file_path = os.path.join(base_path, subdir, filename)
                if os.path.exists(file_path):
                    return file_path
    return None

## USB drive detection daemon
# @param callback Callback function to call when USB drive is detected or removed
# @param filename Name of the file to detect
# @param interval Interval in seconds to check for USB drive
# @return Thread object of the daemon, call start() to start the daemon
def usb_detection_daemon(callback: Callable[[str|None], None], filename: str, interval: float = 1):
    def daemon():
        previous_usb_path = None
        callback(None)  # initial call
        while True:
            usb_path = detect_usb_drive_with_file(filename)
            if usb_path == previous_usb_path:
                time.sleep(interval)
                continue
            previous_usb_path = usb_path
            if usb_path:
                callback(usb_path)
            else:
                callback(None)
        time.sleep(interval)

    t = threading.Thread(target=daemon)
    t.daemon = True
    return t
```

List. 1 – usb_utils.py – module responsible for looking up usb drives with encrypted private key

Other parts of the application are designed to work independently of the underlying OS (logic is written in Python and GUI is created in tkinter).

Application was created by both creators over VSCode's LiveShare and later published to Bartłomiej Krawisz's GitHub account.

1.3 Application's Usage Flow

To begin using / testing the application run generator.py application and insert the USB drive that will be used to store application's secret. The application should display prompt for entering PIN for the secret:

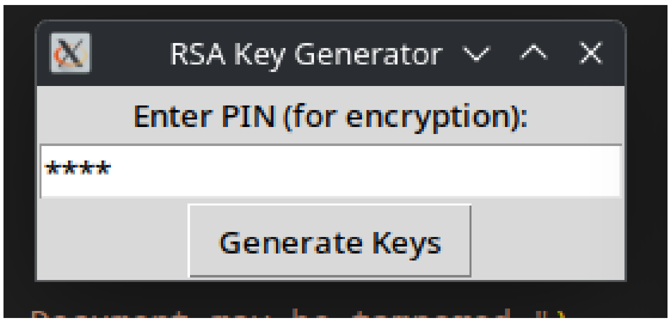


Fig. 2 – generator.py PIN prompt window

After providing the PIN, click Generate Keys button and select the USB drive in a file manager window:

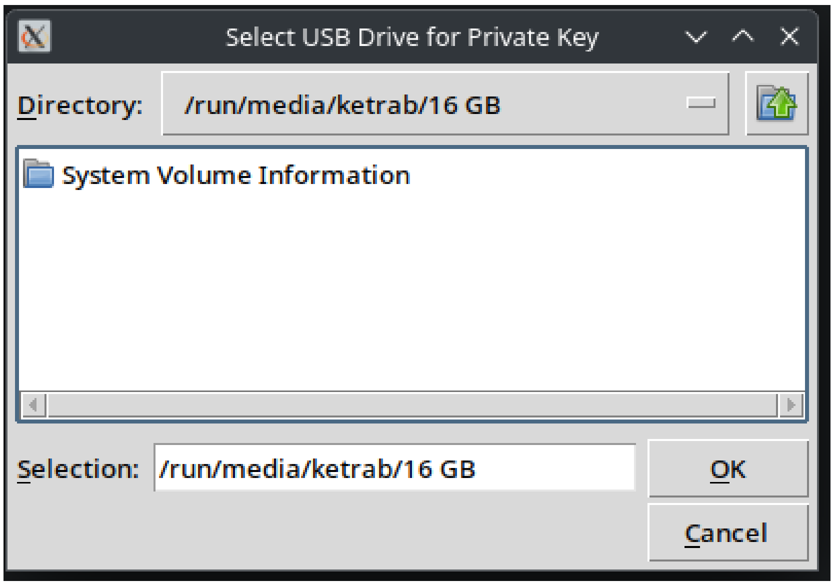
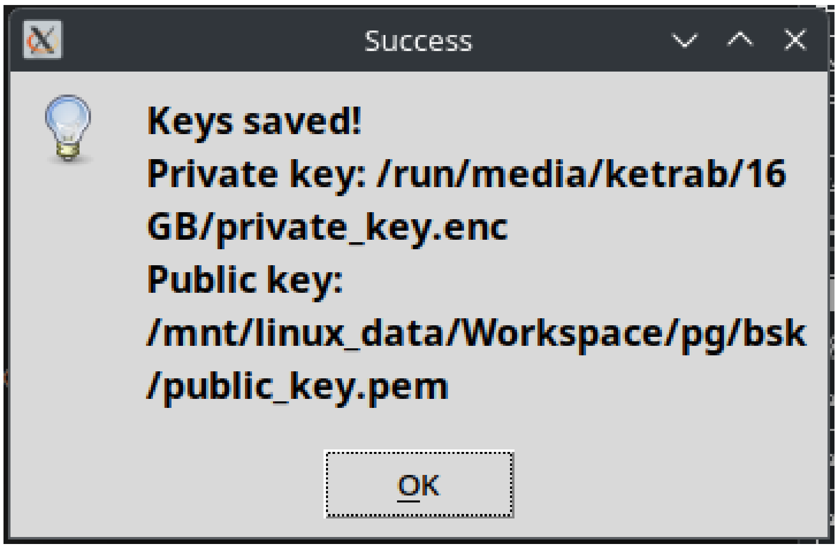

Select USB Drive for Private Key
Directory: /run/media/ketrab/16 GB
System Volume Information
Selection: /run/media/ketrab/16 GB
OK
Cancel

Fig. 3 – File manager window used to select USB drive

When OK button is clicked the application saves the secret to selected location and displays confirmation window with both public and private keys locations. (Public key is stored in the same directory in which program was run from).



```
Keys saved!
Private key: /run/media/ketrab/16
GB/private_key.enc
Public key:
/mnt/linux_data/Workspace/pg/bsk
/public_key.pem
```

Fig. 4 – Confirmation window

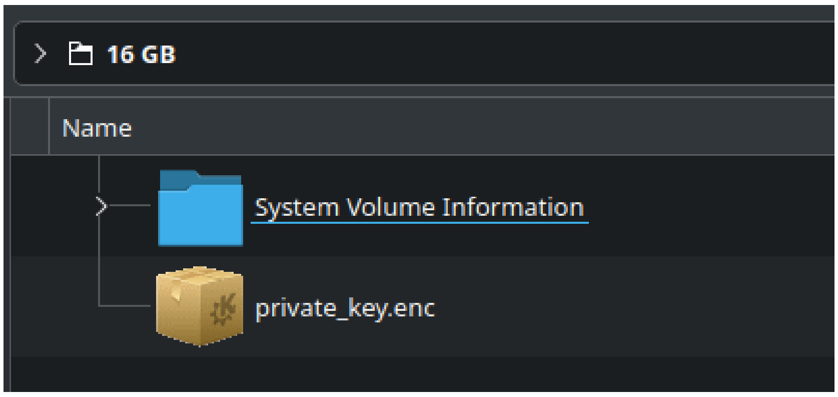


Fig. 5 – USB drive's contents after running generate.py

After preparing the USB drive you can eject it for further usage.

To sign the document, you can use sign.py. When run, the application will display window prompting for PIN and will display whether USB drive is detected or not.

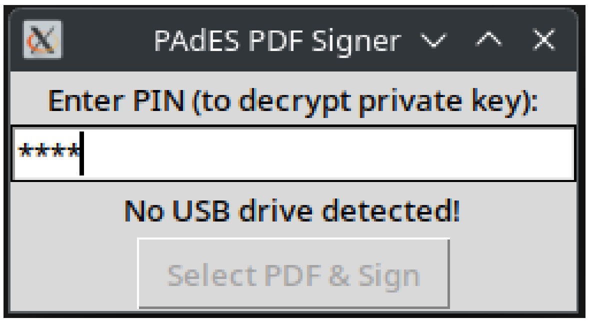


Fig. 6 – Window without any USB drive detected

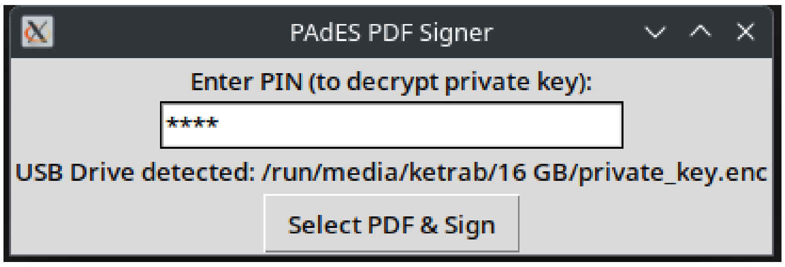

PAdES PDF Signer
Enter PIN (to decrypt private key):
****
USB Drive detected: /run/media/ketrab/16 GB/private_key.enc
Select PDF & Sign

Fig. 7 – Window with USB drive with secret detected

When USB drive is detected, and correct PIN is provided the user can click the button below and select the PDF document to sign. When correct PDF document is selected and button Open is clicked the application will display the message window informing us about the successful PDF document signing and the location of signed PDF (should be next to original PDF file with _signed suffix).

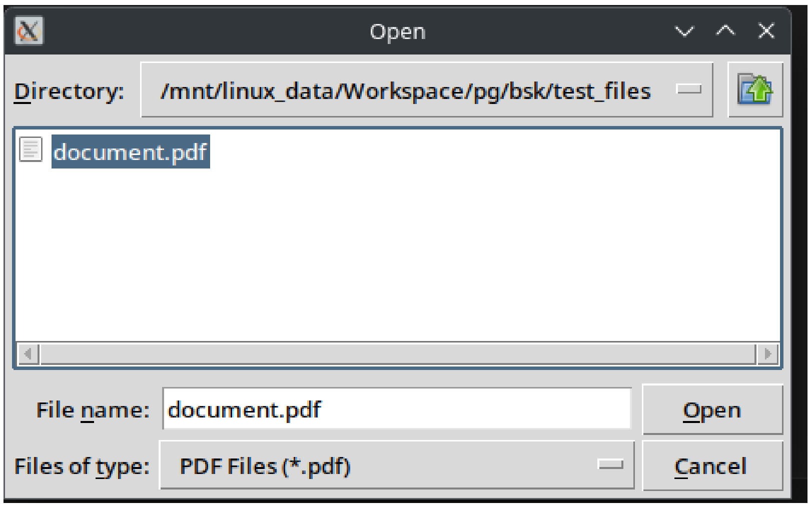
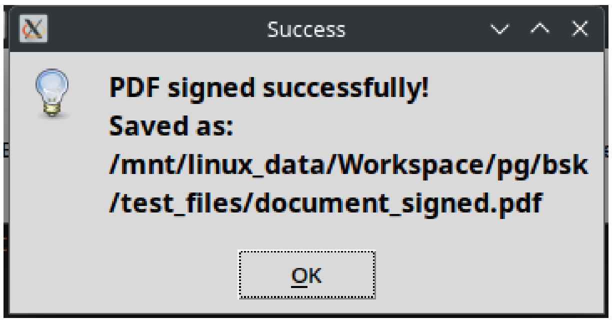

Open
Directory: /mnt/linux_data/Workspace/pg/bsk/test_files
document.pdf
File name: document.pdf
Open
Files of type: PDF Files (*.pdf)
Cancel

Fig. 8 – File Manager's window used to select the PDF document to sign



Success

PDF signed successfully!
Saved as:
/mnt/linux_data/Workspace/pg/bsk
/test_files/document_signed.pdf

Fig. 9 – Success message after successfully signing the PDF document

To test whether the signed PDF document has been tampered with, run verify.py, select public key alongside PDF to check and message will appear with requested information.

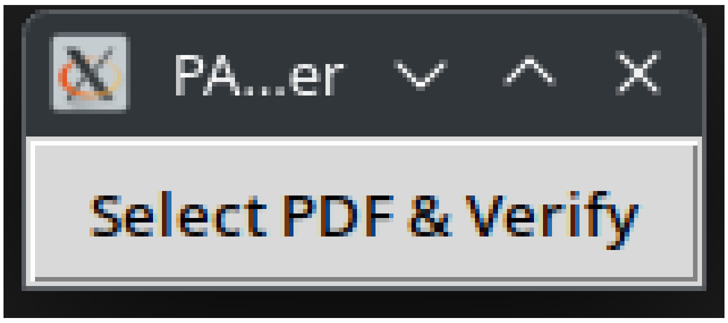
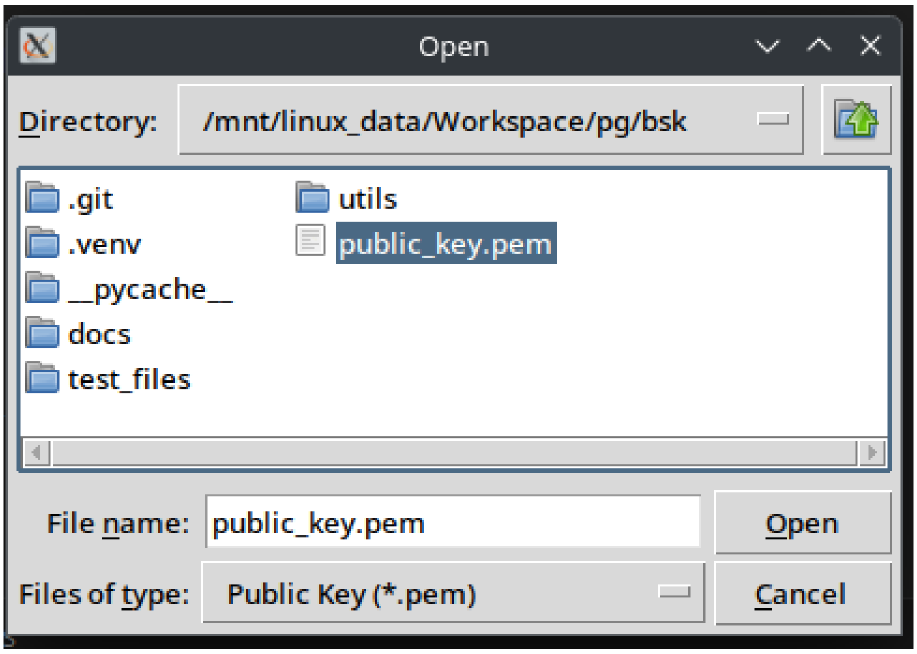


Fig. 10 – verify.py window



Open

Directory: /mnt/linux_data/Workspace/pg/bsk

.git
.venv
__pycache__
docs
test_files
utils
public_key.pem

File name: public_key.pem

Files of type: Public Key (*.pem)

Fig. 11 – File Manager's window with public key selection

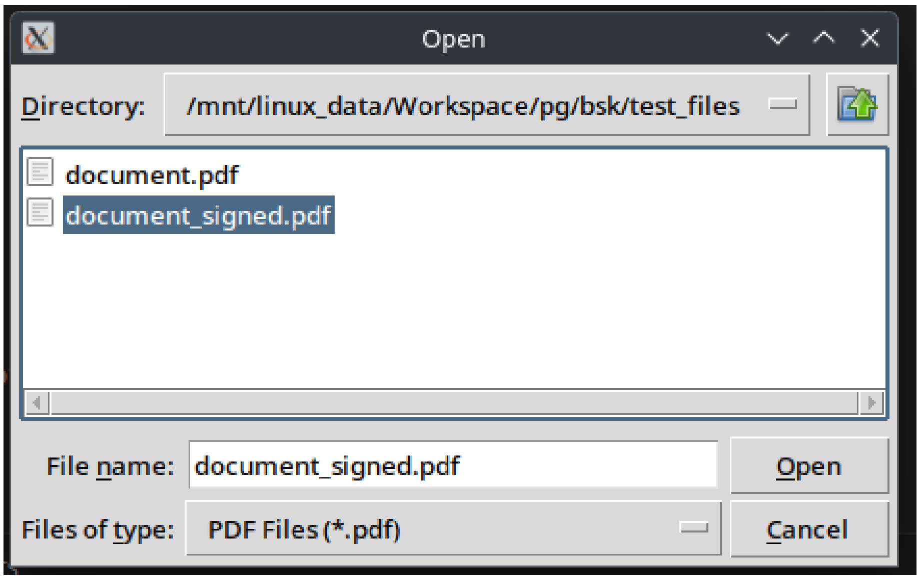
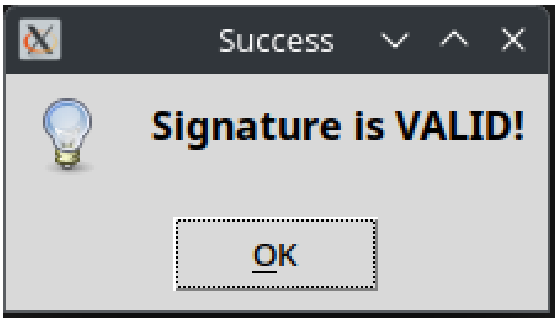
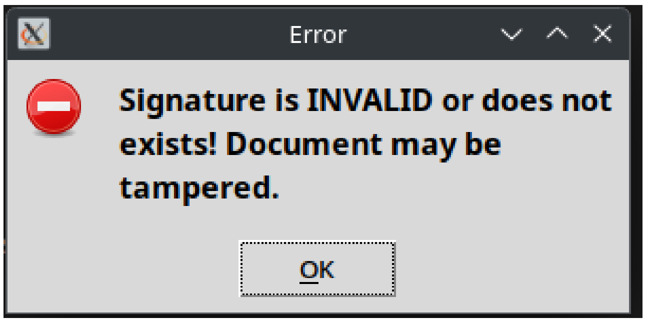


Fig. 12 – File Manager's window with PDF document selection


Success

Signature is VALID!

Fig. 13 – Success window informing us of temperance-free PDF document


Error

Signature is INVALID or does not exists! Document may be tampered.

Fig. 14 – Failure window informing us about PDF document temperance

1.4 Important application's parts

To sign the PDF documents, application:

- Generates RSA keys pair
- Encrypts the private key using provided PIN (it will be required to sign the documents)
- When signing the document: the PDF's content hash is created, encrypted using private key and stored at the end of the signed document

To verify the PDF document, application:

- Reads the public key and uses it to decrypt the stored hash in the PDF document
- Hashes the PDF document without the signature and compares them. When difference is detected it means that the document has been tampered with.

```
## @package utils.crypt_utils
# Cryptography utility functions
#
# This module provides utility functions for RSA key generation and encryption/decryption of private keys.
#
# It uses the PyCryptodome library for cryptographic operations.

from Crypto.PublicKey import RSA
from Crypto.Cipher import AES
from Crypto.Hash import SHA256

## Length of the RSA key
RSA_KEY_LENGTH = 4096


## Generate RSA key pair
# @return Tuple of private key and public key in bytes
def generate_rsa_keys():
    key = RSA.generate(RSA_KEY_LENGTH)
    private_key = key.export_key()
    public_key = key.publickey().export_key()
    return private_key, public_key

## Encrypt the private key using a PIN
# @param private_key Private key in bytes
# @param pin PIN to encrypt the private key
# @return Encrypted private key in bytes
def encrypt_private_key(private_key: bytes, pin: str):
    hash_obj = SHA256.new(pin.encode('utf-8'))
    aes_key = hash_obj.digest()

    cipher = AES.new(aes_key, AES.MODE_EAX)
    ciphertext, tag = cipher.encrypt_and_digest(private_key)

    return cipher.nonce + tag + ciphertext

## Decrypt the private key using a PIN
# @param encrypted_key Encrypted private key in bytes
# @param pin PIN to decrypt the private key
# @return Decrypted private key as RSA key object
def decrypt_private_key(encrypted_key: bytes, pin: str):
    hash_obj = SHA256.new(pin.encode('utf-8'))
    aes_key = hash_obj.digest()

    nonce, tag, ciphertext = encrypted_key[:16], encrypted_key[16:32], encrypted_key[32:]
    cipher = AES.new(aes_key, AES.MODE_EAX, nonce=nonce)

    try:
        private_key = cipher.decrypt_and_verify(ciphertext, tag)
        return RSA.import_key(private_key)
    except ValueError:
        return None
```

List. 2 – crypt_utils.py – module responsible for all cryptographic operations

```
## @package utils.pdf_signing_utils
# PDF signing utility functions
#
# This module provides utility functions for signing and verifying PDF files using RSA digital signatures.

from Crypto.PublicKey import RSA
import hashlib

## Length of the RSA signature in bytes
SIGNATURE_LENGTH = 512

## Length of the hash in bytes
HASH_LENGTH = 256


## Sign a PDF file using an RSA private key
# @param pdf_path Path of the PDF file to sign
# @param private_key RSA private key object
# @param signed_pdf_path Path to save the signed PDF file
# @return Path of the signed PDF file
def sign_pdf(pdf_path: str, private_key: RSA.RsaKey, signed_pdf_path: str | None = None):
    if signed_pdf_path is None:
        signed_pdf_path = pdf_path.replace('.pdf', '_signed.pdf')

    with open(pdf_path, 'rb') as f:
        pdf_data = f.read()
    pdf_hash = hashlib.sha256(pdf_data).digest()

    signature = pow(int.from_bytes(pdf_hash, byteorder='big'), private_key.d, private_key.n)
    signature_bytes = signature.to_bytes(SIGNATURE_LENGTH, byteorder='big')

    with open(signed_pdf_path, 'wb') as f:
        f.write(pdf_data + signature_bytes)

    return signed_pdf_path

## Verify the signature of a signed PDF file using an RSA public key
# @param pdf_path Path of the signed PDF file
# @param public_key RSA public key object
# @return True if the signature is valid, False otherwise
def verify_signature(pdf_path: str, public_key: RSA.RsaKey):
    with open(pdf_path, 'rb') as f:
        content = f.read()
    pdf_data, signature = content[:-SIGNATURE_LENGTH], content[-SIGNATURE_LENGTH:]
    pdf_hash = hashlib.sha256(pdf_data).digest()

    decrypted_hash = pow(int.from_bytes(signature, byteorder='big'), public_key.e, public_key.n)

    try:
        decrypted_hash_bytes = decrypted_hash.to_bytes(HASH_LENGTH//8, byteorder='big')
    except OverflowError:
        return False

    return pdf_hash == decrypted_hash_bytes
```

List. 3 – pdf_signing_utils.py – module responsible for signing PDF files

Applications frontend is written using tkinter. As an example, the generate.py is attached below:

```
## @package generate
# RSA key generation user GUI application
#
# This script generates RSA key pair and saves the private key on a USB drive and public key on the local machine.

import os
import tkinter as tk
from tkinter import filedialog, messagebox

from consts import PRIVATE_KEY_FILENAME, PUBLIC_KEY_FILENAME
from utils.crypt_utils import generate_rsa_keys, encrypt_private_key


## Main application class for RSA key generation
class GenerateApp:
    ## Constructor
    def __init__(self):
        # GUI setup
        self.__root = tk.Tk()
        self.__root.title("RSA Key Generator")

        tk.Label(self.__root, text="Enter PIN (for encryption):").pack()
        self.__pin_entry = tk.Entry(self.__root, show="*", width=30)
        self.__pin_entry.pack()

        self.__generate_button = tk.Button(self.__root, text="Generate Keys", command=self.__save_keys)
        self.__generate_button.pack()

    ## Start the application
    def start(self):
        self.__root.mainloop()

    def __save_keys(self):
        pin = self.__pin_entry.get()
        if not pin:
            messagebox.showerror("Error", "PIN cannot be empty!")
            return

        usb_path = filedialog.askdirectory(title="Select USB Drive for Private Key")
        if not usb_path:
            return

        private_key, public_key = generate_rsa_keys()  # Generate keys
        encrypted_private_key = encrypt_private_key(private_key, pin)

        # Save encrypted private key on USB drive
        private_key_path = os.path.join(usb_path, PRIVATE_KEY_FILENAME)
        with open(private_key_path, "wb") as f:
            f.write(encrypted_private_key)

        # Save public key on local machine
        public_key_path = os.path.join(os.getcwd(), PUBLIC_KEY_FILENAME)
        with open(public_key_path, "wb") as f:
            f.write(public_key)

        messagebox.showinfo("Success", f"Keys saved!\nPrivate key: {private_key_path}\nPublic key: {public_key_path}")

        # Clear the PIN entry
        self.__pin_entry.delete(0, tk.END)


if __name__ == "__main__":
    app = GenerateApp()
    app.start()
```

List. 4 – generate.py – application's frontend

1.5 Summary

Application has been successfully presented and verified by the teacher. Application's source code is available on GitHub https://github.com/ketrab2003/pades-emulation alongside the documentation generated using Doxygen https://ketrab2003.github.io/pades-emulation/.

2. Bibliography

- https://en.wikipedia.org/wiki/PAdES
- Project's Instruction