# Security of Computer Systems

## Project Report

Authors:
Jakub, Bronowski, 193208
Piotr, Trybisz, 193557

Version: 2.0

# Versions

| Version | Date | Description of changes |
|---|---|---|
| 1.0 | 08.04.2025 | Creation of the document. |
| 1.1 | 08.04.2025 | Added pictures. |
| 1.2 | 09.04.2025 | Filling parts necessary to pass control term. |
| 2.0 | 10.06.2025 | Filling parts necessary to pass submission term. |

# 1. Project – control term

## *1.1 Description*

The main aim of this project is to create an app which will allow user to sign and verify the PDF document signage. Also, app has to allow user to generate a pair of RSA keys to make all operations possible.

Link to Github: https://github.com/JakubBron/bsk

## *1.2 Results*

App was created in Python with usage of PyQT5 UI library. App is also available via TUI, which was created earlier in the process.

App consists of 3 functional parts:

1. Key management (fig. 1)

Here, user can set paths to private and public keys. Colours of the text next to textboxes are signalising if keys are present in given places. Also, here user sets paths where new keys will be stored.

When user set paths, user is asked to enter the password (PIN, can contains letters) to unlock access to files or to create new set of keys. If password is incorrect, user is asked again and files are not accessible in the app. Worth mentioning is a fact that if user creates new pair of keys, user has to remember PIN number. It is impossible to set new password to already existing keys, so that's why in this case PIN popup window is showed just once. Field to enter the password is placed in popup window (fig. 3). Process of key generation takes some time, that is why popup windows informing about executed processes are displayed (fig. 4 and fig. 5).

2. Document signing (fig. 1)

Here user can enter path to file user wants to sign and where to store signed file. It is possible to overwrite not signed file. Operation status is shown to user in popup window (fig. 6). Signature is added to the end of the PDF file, past last *EOF* tag. Signature contains encrypted SHA-256 checksum of original PDF document. Only *.pdf* documents can be signed.

3. Document verifying (fig. 2)

Here user can verify if given document is signed and not modified after. User has to specify path to PDF file. If document wasn't changed after signing, positive result is shown to user in popup window (fig. 7). If PDF verification failed, popup window is also displayed, but with appropriate text (fig. 8). Only *.pdf* documents can be verified. Verification consist of splitting signed file to its content and signature. Encrypted SHA-256 checksum is calculated and checked if matches file signature.

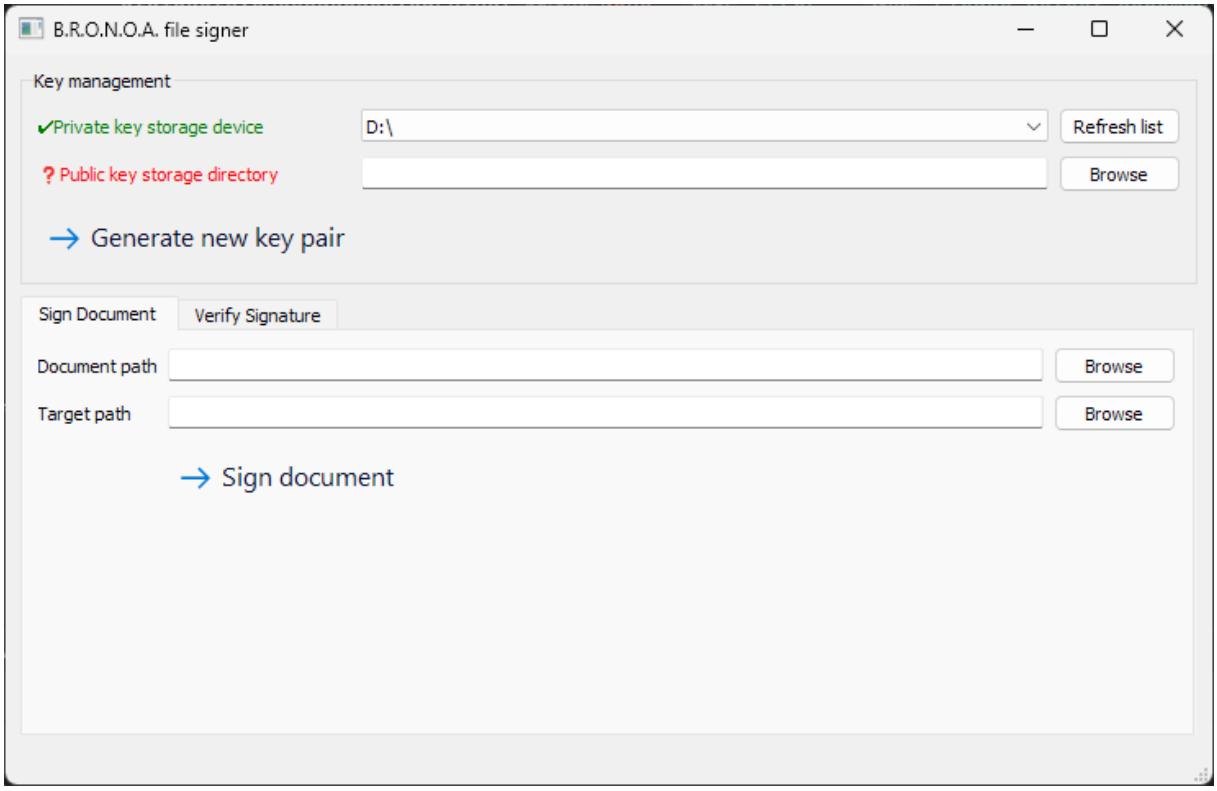


*Fig. 1 – App main screen.*

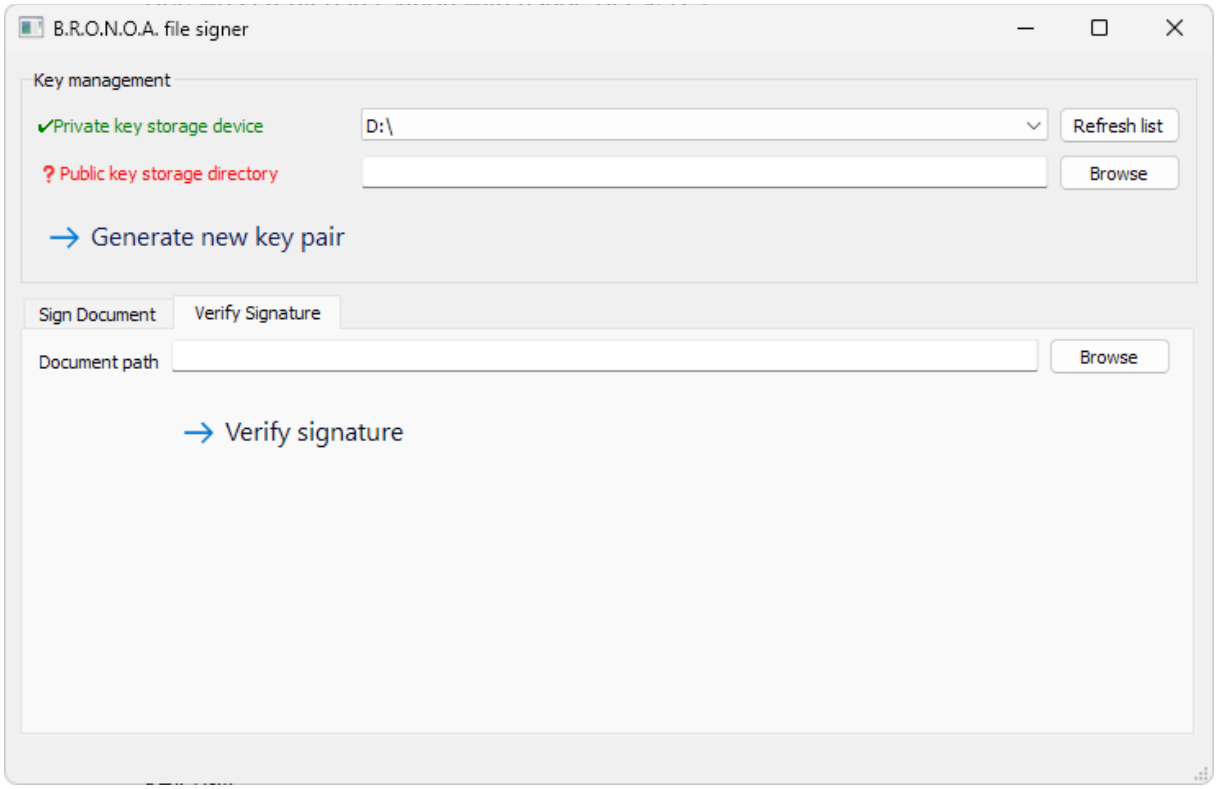


*Fig. 2 – Signature verifying screen.*

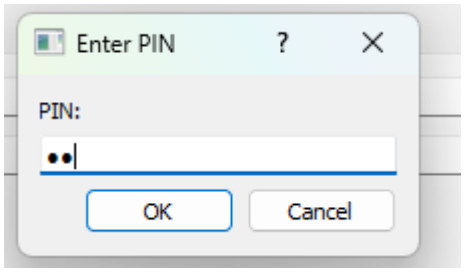


*Fig. 3 – PIN window.*

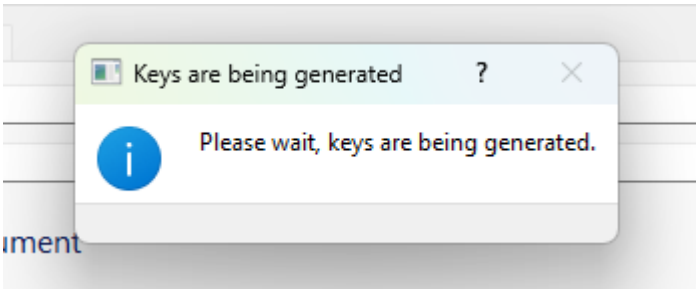


*Fig. 4 – Key generation popup window.*

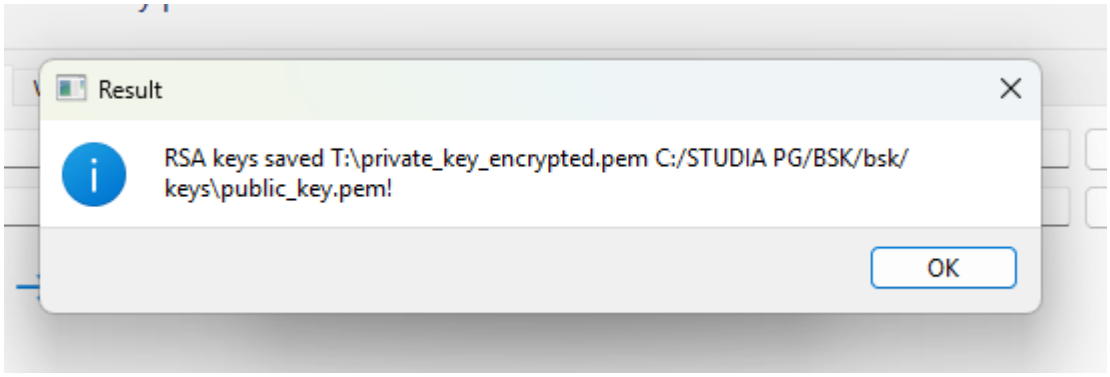


*Fig. 5 – Success of key generation.*

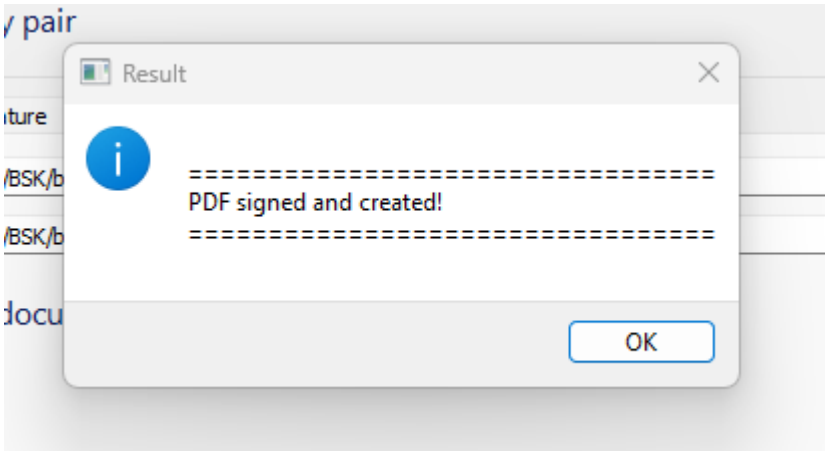


*Fig. 6 – Success of creating signed PDF.*

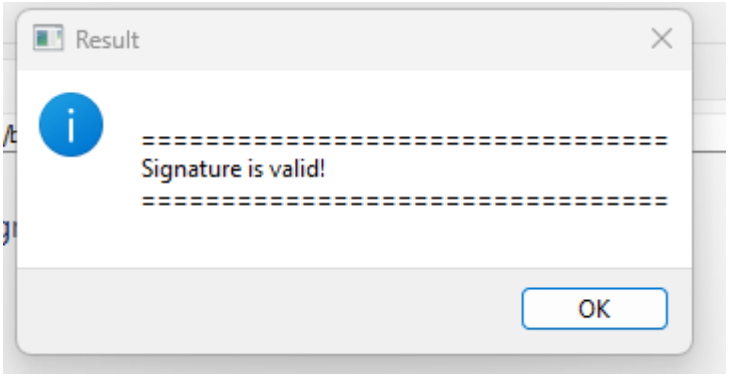


*Fig. 7 – Success of PDF verification.*

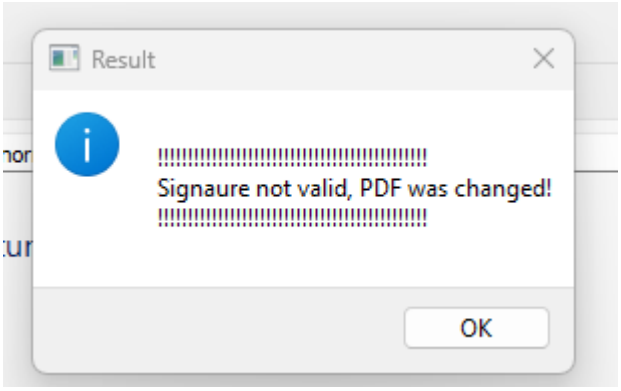


*Fig. 8 – Failure of PDF verification.*

### *1.3 Summary*

Application works in expected way. All critical methods are implemented using *try… except* mechanism to handle errors during its execution. App was tested manually.

# 2. Project – Final term

## *2.1 Description*

Fully working application was coded, tested manually and passed code review. General code documentation was created with Doxygen. It's accessible in the project Github repository (https://github.com/JakubBron/bsk). Documentation can be generated with the doxygen cli tool.

## *2.2 Code Description*

Pendrive handler – preparing device and setting PIN.

```
def __init__(self, drive, public_key_path=None, pin=None):
"""!
@brief Constructor for Pendrive class.
@param drive Path to the mounted pendrive.
@param public_key_path Directory where the public key is or will be stored.
@param pin PIN used to derive AES key.
"""
if not os.path.exists(drive):
    raise Exception(f"Drive {drive} is not plugged in or is damaged!")
self.pin = pin
self.aes_key = self.generate_AES_key(pin) if pin else None
self.drive = drive
self.private_key_path = os.path.join(drive, FILENAMES.PRIVATE_KEY_ENCRYPTED)
self.public_key_path = os.path.join(public_key_path, FILENAMES.PUBLIC_KEY)
```

*List. 1a – Initialization of pendrive handler.*

```
def set_pin(self, new_pin: int) -> None:
"""!
@brief Sets the PIN and regenerates AES key.
@param new_pin The new PIN value.
"""
self.pin = new_pin
self.aes_key = self.generate_AES_key(new_pin)

def get_pin(self) -> int:
"""!
@brief Returns the current PIN.
@return PIN value.
"""
return self.pin
```

*List. 1b – PIN setter and getter.*

Pendrive handler – AES key generation and getter to AES key

```
def generate_AES_key(self, pin: int) -> str:
"""!
@brief Generates a 256-bit AES key from a PIN.
@param pin Integer PIN.
@return AES key derived via SHA-256.
"""
return hashlib.sha256(str(pin).encode()).digest()

def get_AES_key(self) -> str:
"""!
@brief Gets the AES key.
@return Current AES key.
"""
return self.aes_key
```

*List. 2 – AES key generation and getter to AES key*.

Pendrive handler – AES encryption and decryption.

```
def encrypt_AES(self, message: str) -> str:
"""!
@brief Encrypts a message using AES.
@param message Plaintext message.
@return Base64-encoded ciphertext.
"""
message = pad(message.encode(), AES.block_size)
cipher = AES.new(self.aes_key, AES.MODE_CBC, MODES.AES_IV)
return base64.b64encode(cipher.encrypt(message)).decode()


def decrypt_AES(self, encrypted: str) -> str:
"""!
@brief Decrypts a Base64-encoded AES-encrypted message.
@param encrypted Base64-encoded ciphertext.
@return Decrypted plaintext.
"""
encrypted = base64.b64decode(encrypted)
cipher = AES.new(self.aes_key, AES.MODE_CBC, MODES.AES_IV)
return unpad(cipher.decrypt(encrypted), AES.block_size).decode()
```

*List. 3 – AES encryption and decryption*.

Pendrive handler – RSA handler (generating, getting keys).

```
def generate_RSA_key(self, encrypt=True) -> (str, str):
"""!
@brief Generates RSA public/private key pair.
@param encrypt Whether to AES-encrypt the private key.
@return Tuple containing public key and (optionally encrypted) private key.
"""
key = RSA.generate(LENGTHS.RSA_LENGTH)
public_key = key.publickey().export_key().decode()
private_key = key.export_key().decode()
private_key_encrypted = private_key
if encrypt == True:
    private_key_encrypted = self.encrypt_AES(private_key)

return (public_key, private_key_encrypted)
```

*List. 4a – RSA keys generator.*

```
def save_RSA_keys(self) -> str:
"""!
@brief Saves the generated RSA keys to the pendrive and public key path.
@return Message indicating result.
"""
public_key, private_key_encrypted = self.generate_RSA_key()
if not os.path.exists(self.drive):
    return f"Drive {self.drive} is not plugged in or is dagamed!"

try:
    with open(self.private_key_path, "w") as f:
    f.write(private_key_encrypted)
except Exception as e:
    return f"Unable to save {self.private_key_path} file! {e}"

try:
    with open(self.public_key_path, 'w') as f:
    f.write(public_key)
except Exception as e:
    return f"Unable to save {self.public_key_path} file! {e}"

return f"RSA keys saved {self.private_key_path} {self.public_key_path}!"
```

*List. 4b – saving RSA keys.*

```
def get_RSA_public_key(self) -> str:
"""!
@brief Reads and returns the stored public key.
@return Public key or error message.
"""
try:
    with open(self.public_key_path, "r") as f:
    return f.read()
except Exception as e:
    return f"UNABLE TO READ {self.public_key_path} file! {e}"

def get_RSA_private_key(self) -> str:
"""!
@brief Reads and decrypts the private key.
@return Private key or error message.
"""
```

```
try:
    with open(self.private_key_path, "r") as f:
    content = f.read()
    return self.decrypt_AES(content)
except Exception as e:
    return f"UNABLE TO READ {FILENAMES.PRIVATE_KEY_ENCRYPTED} file! {e}"

def get_RSA_private_key_encrypted(self) -> str:
"""!
@brief Reads the encrypted private key file.
@return Encrypted private key or error message.
"""
try:
    with open(self.drive+'\\'+FILENAMES.PRIVATE_KEY_ENCRYPTED, "r") as f:
    return f.read()
except Exception as e:
    return f"UNABLE TO READ {FILENAMES.PRIVATE_KEY_ENCRYPTED} file! {e}"

def check_if_RSA_keys_exist(self) -> bool:
"""!
@brief Checks if the public key exists on the drive.
@return True if it exists, False otherwise.
"""
return os.path.isfile(self.public_key_path)
```

*List. 4c – getters of RSA keys and RSA public key existence checker.*

PDF signing.

```
def __init__(self, private_key, path: str, target_path: str) -> None:
    """!
    @brief Constructor for PDF_Signer.
    @param private_key PEM-encoded RSA private key.
    @param path Path to the original PDF file.
    @param target_path Path where the signed PDF will be saved.
    """
    self.private_key = RSA.import_key(private_key)
    self.path_to_pdf = path
    self.path_to_signed_pdf = target_path

def create_signature(self, pdf_hash):
    """!
    @brief Creates a numerical RSA signature from the PDF hash.
    @param pdf_hash SHA-256 hash of the PDF content.
    @return RSA signature as an integer.
    """
    signature = pow(int.from_bytes(pdf_hash, byteorder='big'), self.private_key.d, self.private_key.n)
    return signature

def create_binary_signature(self, pdf_hash):
    """!
    @brief Converts numerical RSA signature into binary format.
    @param pdf_hash SHA-256 hash of the PDF content.
    @return Binary representation of the RSA signature.
    """
    signature = self.create_signature(pdf_hash)
    return signature.to_bytes(LENGTHS.SIGNATURE_LENGTH, byteorder='big')
```

*List. 5a – Class realising PDF signing procedure (creating signature in given format).*

```
def sign_pdf(self):
    """!
    @brief Signs a PDF file and appends the signature.
    @return Status message indicating success or failure.
    """
    pdf = None
    pdf_hash = None
    try:
        with open(self.path_to_pdf, 'rb') as f:
        pdf = f.read()
        pdf_hash = hashlib.sha256(pdf).digest()

    except Exception as e:
        msg = f"Unable to read unisgned PDF from {self.path_to_pdf} file! {e}"
        return msg

    signature_bin = self.create_binary_signature(pdf_hash)

    try:
        with open(self.path_to_signed_pdf, 'wb') as f:
        f.write(pdf+b"\nSIGNATURE\n"+signature_bin)

    except Exception as e:
        msg = f"Unable to create signed PDF to {self.path_to_signed_pdf} file! {e}"
        return msg

    return "\n=============\nPDF signed and created!\n============\n"
```

*List. 5b – Class realising PDF signing procedure (appending to file).*

PDF signature verifying.

```
class PDF_Verifier:
    """!
    @class PDF_Verifier
    @brief Verifies the signature of a signed PDF using the public key.
    """
    public_key = None
    path_to_signed_pdf = None

def __init__(self, public_key, path: str) -> None:
    """!
    @brief Constructor for PDF_Verifier.
    @param public_key PEM-encoded RSA public key.
    @param path Path to the signed PDF file.
    """
    self.public_key = RSA.import_key(public_key)
    self.path_to_signed_pdf = path

def validate_signature(self):
    """!
    @brief Verifies that the signature is valid and the file is unaltered.
    @return Status message indicating whether the signature is valid.
    """
```

```
    try:
        with open(self.path_to_signed_pdf, 'rb') as f:
        content = f.read()
        pdf_content, pdf_signature = content[:-LENGTHS.SIGNATURE_LENGTH-len("\nSIGNATURE\n")], content[ LENGTHS.SIGNATURE_LENGTH:]
        print(pdf_signature)
    except Exception as e:
        return f"Unable to read signed PDF. Exception: {e}"

    pdf_content_hash = hashlib.sha256(pdf_content).digest()
    signed_hash = pow(int.from_bytes(pdf_signature, byteorder='big'), self.public_key.e, self.public_key.n)

    try:
        decrypted_hash_bytes = signed_hash.to_bytes(LENGTHS.SHA_LENGTH//8, byteorder='big')
    except OverflowError:
        return "\n!!!!!!!!!!!!!!!!!!!\nSignaure not valid, PDF was changed!\n!!!!!!!!!!!!!!!!\n"

    if pdf_content_hash == decrypted_hash_bytes:
        return "\n==========\nSignature is valid!\n======================\n"
    else:
        return "\n!!!!!!!!!!!!!!!\nSignaure not valid, PDF was changed!\n!!!!!!!!!!!!!!!!!!!\n"
```

*List. 6 – Class PDF_verifier with signature verification method.*

### *2.3 Description*

The main aim of this project is to create an app which will allow user to sign and verify the PDF document signage. Also, app has to allow user to generate a pair of RSA keys to make all operations possible.

### *2.4 Results*

Same as in *1.2 Results*. In second project iteration only documentation was generated. No changes in functionalities, GUI or code has not changed (except comments).

### *2.5 Summary*

The project successfully achieved its primary objectives by developing a robust application that enables users to sign and verify PDF documents, as well as generate RSA key pairs for these operations. The application was implemented in Python, utilizing the PyQT5 library for the graphical user interface (GUI), and also offers a text-based user interface (TUI) created earlier in the development process. Code documentation was generated via Doxygen.

## 3. Link to GitHub repository

Link to GitHub: https://github.com/JakubBron/bsk

# 4. Literature

[1] https://en.wikipedia.org/wiki/PAdES [accessed on 08.04.2025]

[2] Project instruction.

[3] Doxygen documentation https://www.doxygen.nl/ [accessed on 10.06.2025]

[4] PyQt5 documentation https://www.riverbankcomputing.com/static/Docs/PyQt5/ [accessed on 10.06.2025]