Ransomware Simulator
Cryptography Course Project Report

⚠️ Disclaimer:
This project is strictly for educational purposes only. It is designed to demonstrate how ransomware works in a controlled, offline laboratory environment. Do NOT use this project on real systems or real data.

Note: This project is designed to be approximately 70–80% similar to real ransomware behavior

I. Introduction / Background

Ransomware is one of the most common types of cyber attacks today. It works by encrypting a victim's files and then demanding money (a "ransom") to unlock them. Because of how powerful and damaging ransomware can be, it is important for cybersecurity students to understand how it works-especially the cryptography behind it. However, studying real ransomware is dangerous because it can accidentally infect the computer. To avoid it, this project create a safe and controlled Ransomware Simulator. It behaves like real ransomware in terms of encryption and workflow, but it does not spread, does not harm the system, and always allows recovery.

Project Goal

The goal is just to help learn the cryptographic techniques used in ransomware without any danger by build a simple educational simulator that:

- Encrypts chosen files or folders
- Uses real cryptographic algorithms (like AES or RSA)
- Generates a ransom note for demonstration
- Allows safe decryption using a private key
- Teaches how real ransomware handles keys, encryption, and file recovery

Motivation

By building and testing this project, I hope I can:

- Understand how encryption affects real files
- Learn how symmetric and asymmetric keys work together
- See how difficult to recover files without the key
- Gain hands-on experience with secure key generation
- Understand why strong defenses and backups are important

Cryptographic concepts Involved

1. Symmetric Encryption
   • One key is used for both encryption and decryption
   • Very fast
   • Used by ransomware to encrypt large files
2. Asymmetric Encryption
   • Two keys: public key (encrypts) and private key (decrypts)
   • Used to protect the symmetric key
   • Makes sure only the attacker can decrypt files
3. Hybrid Encryption
   Real ransomware combines both:
   • Generate a random symmetric key to encrypt files
   • Encrypt that symmetric key with the attacker's public key
4. Hash Functions (SHA-256)
   • Used to verify file integrity
   • Used to generate IDs or signatures in ransom notes

II. System Design / Architecture

The simulator is divided into 2 logical domains: the Attacker Side and the Victim Side.

This separation reflects real ransomware behavior, where the attacker controls the private key and the victim only has access to the public key.

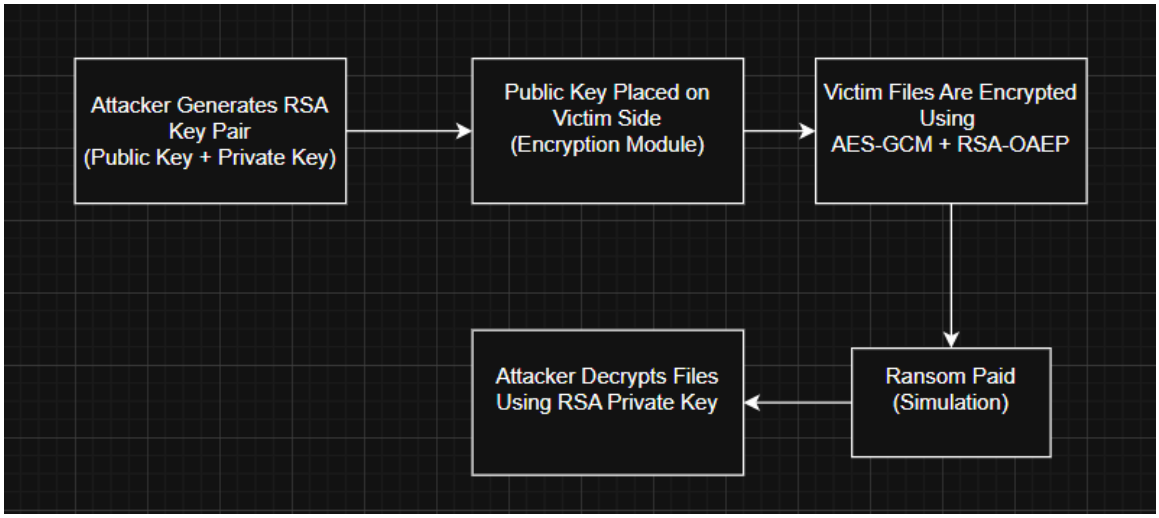
Attacker Side Responsibilities:

- Generate RSA key pair
- Securely store the private key
- Decrypt files after ransom payment

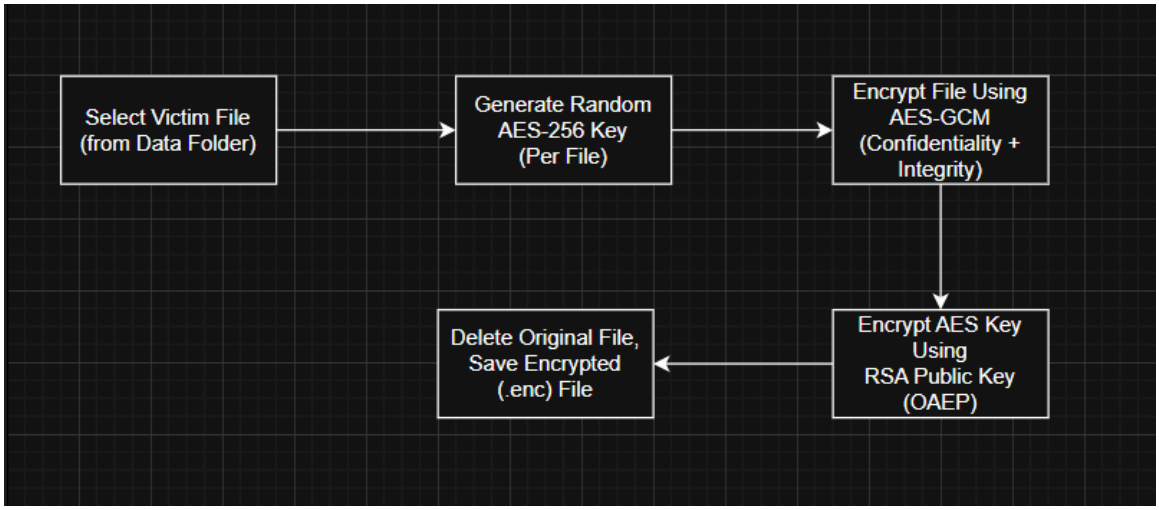
Victim Side Responsibilities:

- Store victim data files
- Backup files before encryption
- Encrypt files using the attacker’s public key
- Unable to decrypt files without attacker intervention

Overall Ransomware Simulator Workflow:



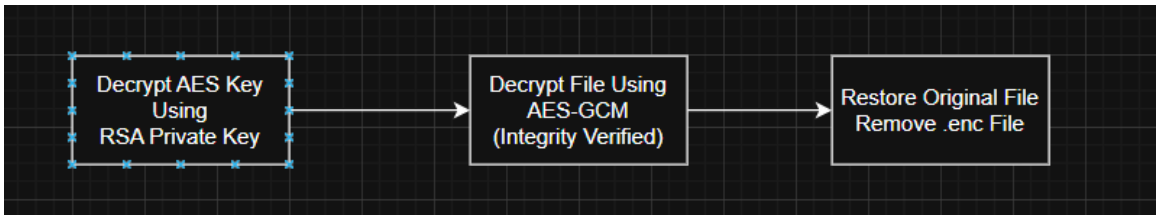
File Encryption Process (Hybrid Encryption):



Each encrypted file follows this structure:

```
[12 bytes]  Nonce (AES-GCM)
[4 bytes ]  Length of RSA-encrypted AES key
[N bytes ]  RSA-encrypted AES key
[M bytes ]  AES-GCM ciphertext + authentication tag
```

File Decryption Process (After Ransom Payment):



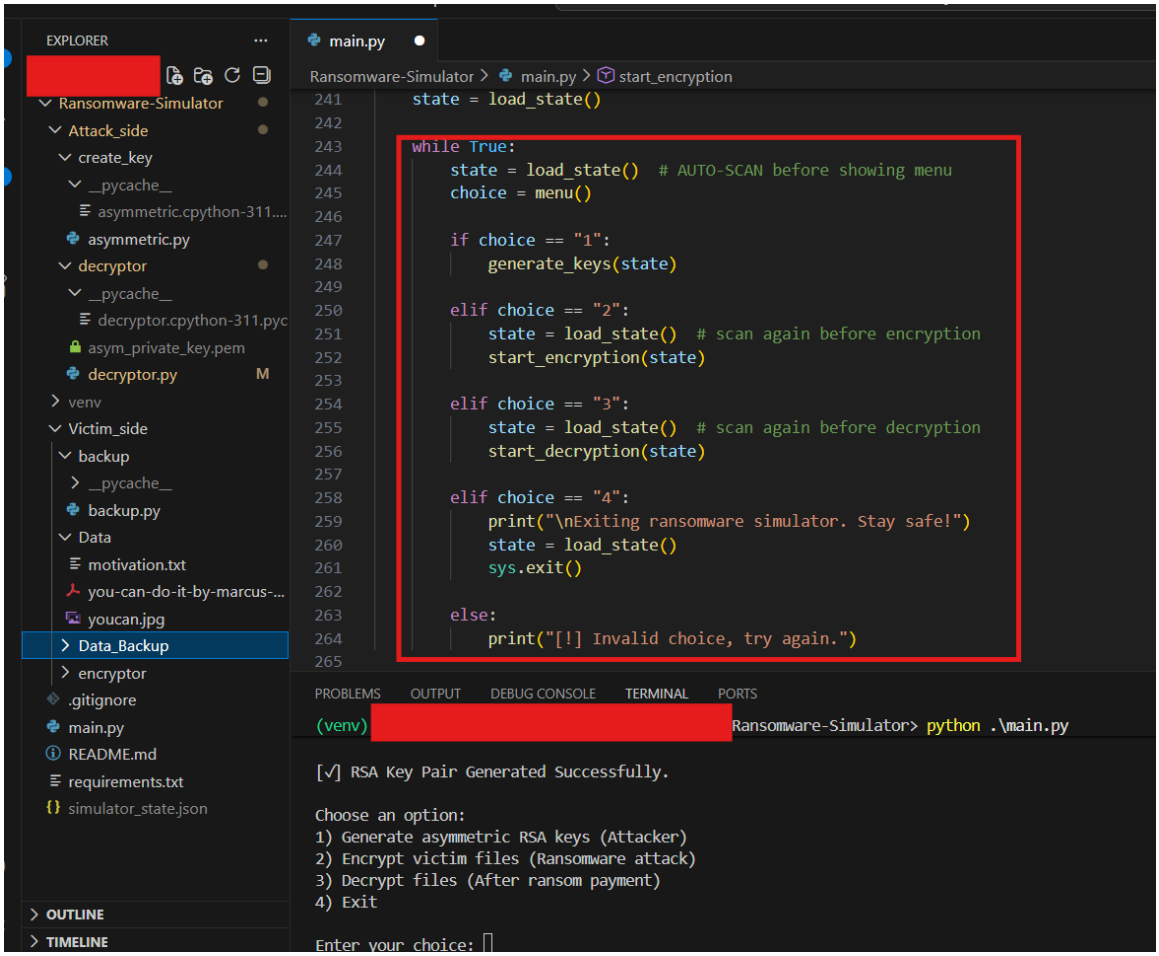
III. Implementation Details

This section describes the internal implementation of the Ransomware Simulator, focusing on the key components, cryptographic methods, program logic, and safety mechanisms used to realistically model ransomware behavior in a controlled educational environment.

The project is implemented in Python and follows a modular architecture, separating attacker-side and victim-side operations to closely reflect real-world ransomware workflows.

1. Overall Architecture and Design

A central controller (main.py) manages program flow, user interaction, automatic file-state detection, and safety checks to prevent irreversible data loss.



2. Cryptographic Libraries Used

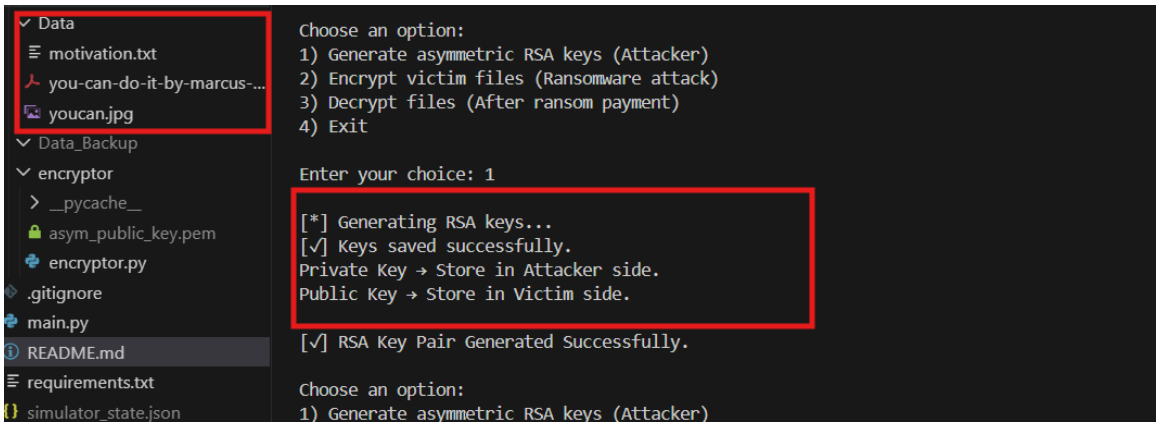
The project uses the Python **cryptography** library to implement secure cryptographic primitives such as RSA-OAEP, AES-GCM, padding, and SHA-256 hashing. Additional standard libraries such as os, json, and shutil are used for file system operations and state persistence.

Library	Purpose
cryptography.hazmat.primitives.asymmetric.rsa	RSA key generation
cryptography.hazmat.primitives.ciphers.aead.AESGCM	AES-256-GCM encryption
cryptography.hazmat.primitives.asymmetric.padding	RSA-OAEP padding
hashes.SHA256	Hash function for OAEP
os, shutil	File system operations
json	Simulator state persistence

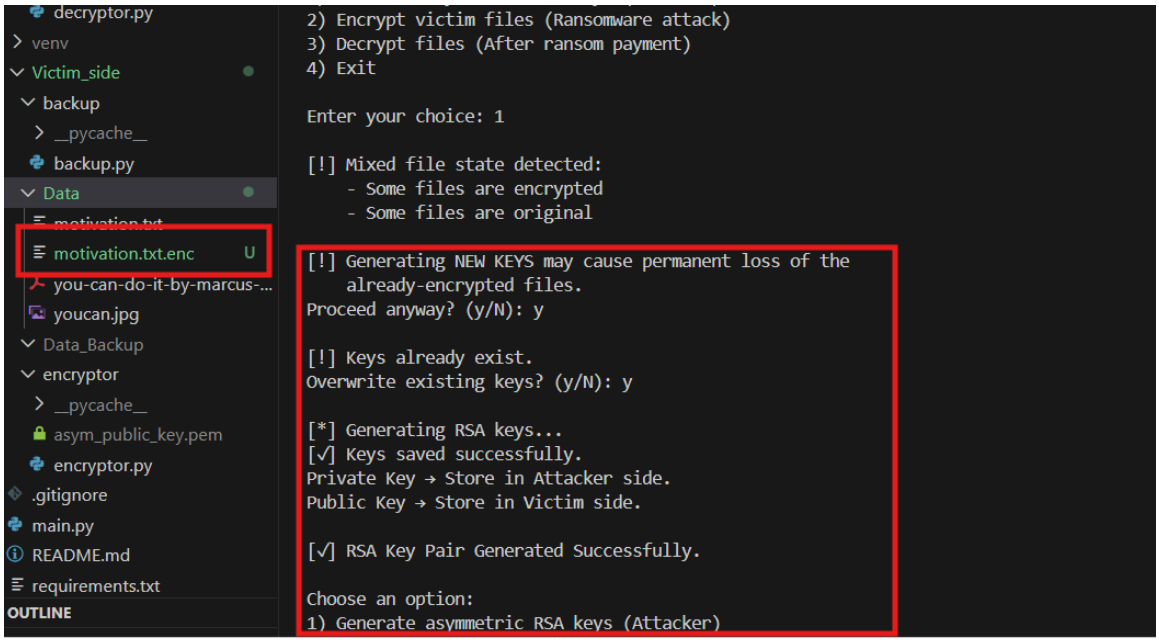
3. RSA Key Generation (Attacker Side)

A 2048-bit RSA key pair is generated using a public exponent of 65537. The private key is stored on the attacker side, while the public key is distributed to the victim side. The simulator blocks key regeneration when encrypted files exist to prevent permanent data loss.

If no encrypted files:



When have encrypted files:

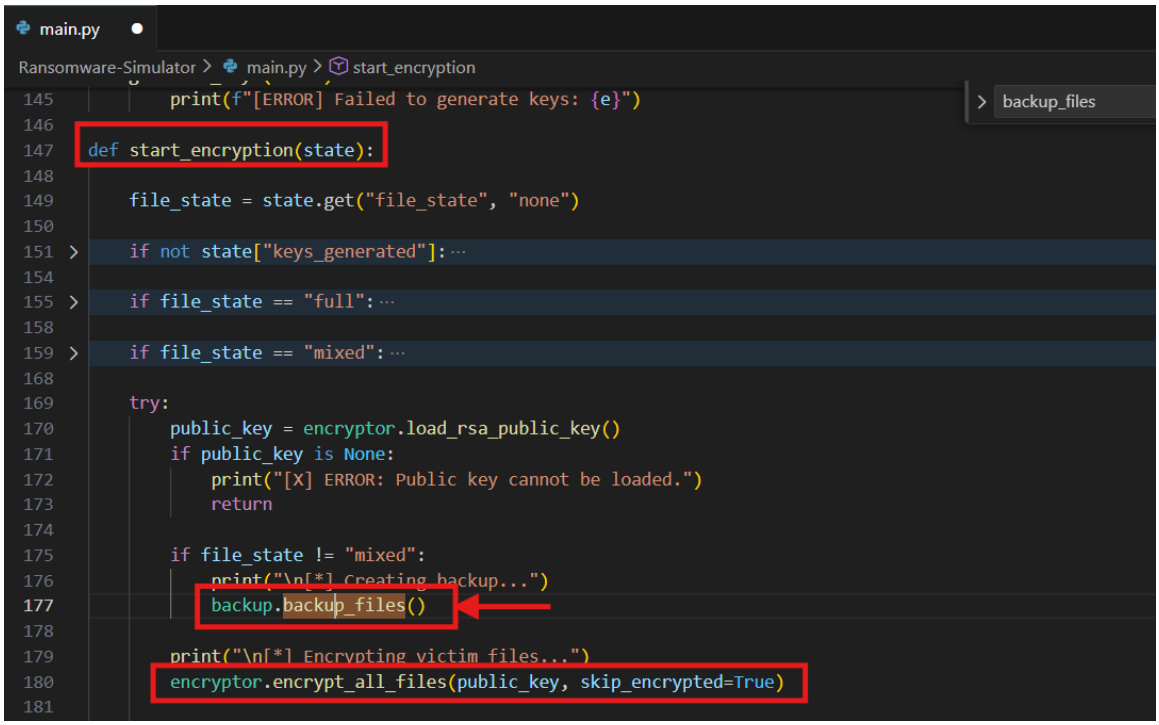


4. Backup Mechanism (Victim Side)

Before encryption, all plaintext files are copied to a backup directory. Encrypted files are excluded from backup. This ensures safe experimentation while demonstrating ransomware behavior.

In real ransomware, attacker never backup victim's data before encrypt but victims can backup their data to other place for before data destroy.

Backup process is function that call before encrypting start:



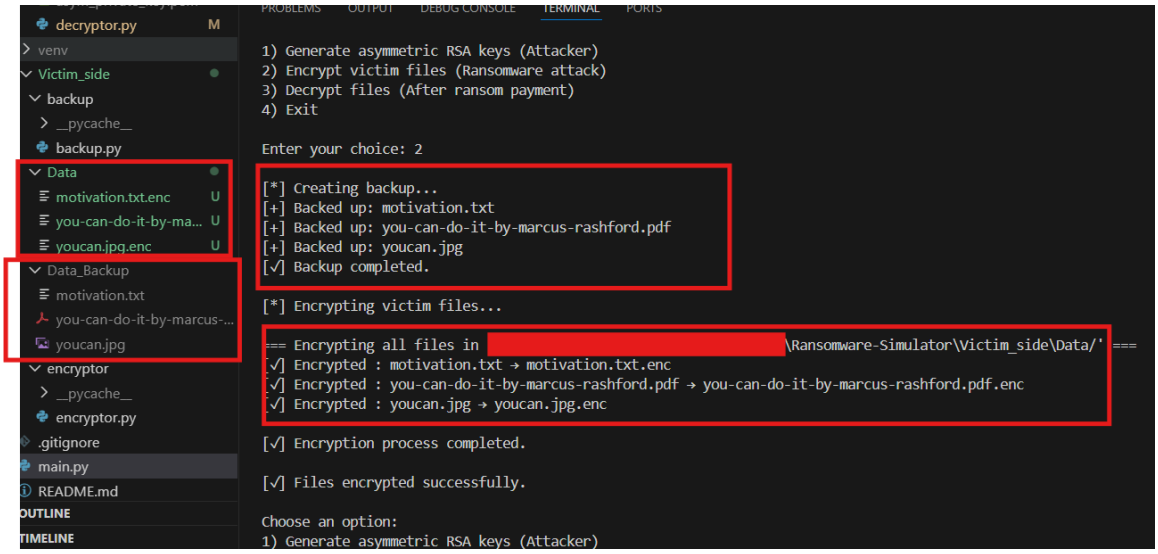
Path of function

- backup_files() : Ransomware-Simulator\Victim_side\backup\backup.py
- encrypt_all_files() : Ransomware-Simulator\Victim_side\encryptor\encryptor.py

5. Hybrid Encryption Process

Each file is encrypted using AES-256-GCM with a randomly generated key. The AES key is then encrypted using RSA-OAEP with SHA-256. The encrypted file format contains a nonce, encrypted AES key length, RSA-encrypted AES key, and ciphertext.

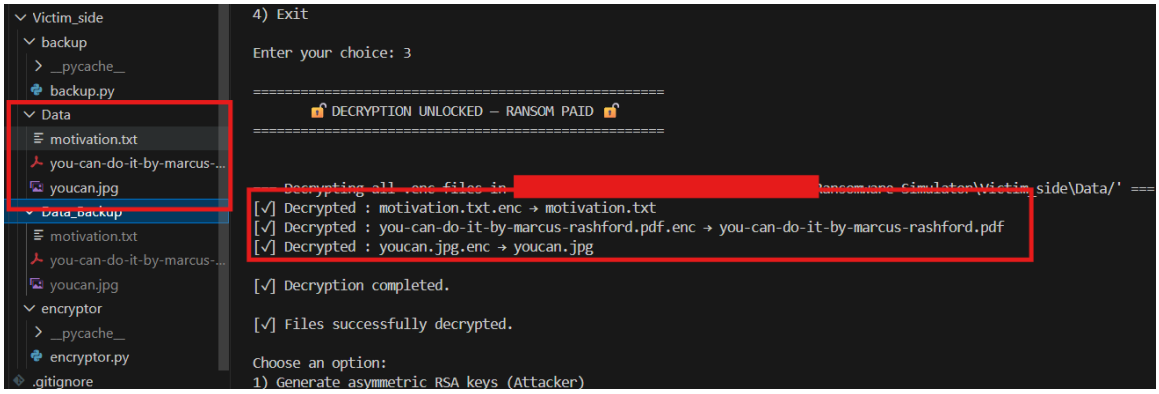
After encrypting the AES key, the RSA public key is securely deleted (in my project). The original plain text file is also securely deleted.



6. Decryption Process

Decryption requires the attacker's private RSA key. The AES key is first recovered using RSA, then the original file is restored using AES-GCM. Encrypted files are removed after successful decryption.

In my project, decrypting the AES key and decrypting the file using AES-GCM run direct on attacker side and the it will send decrypted data to victim_side's Data folder.



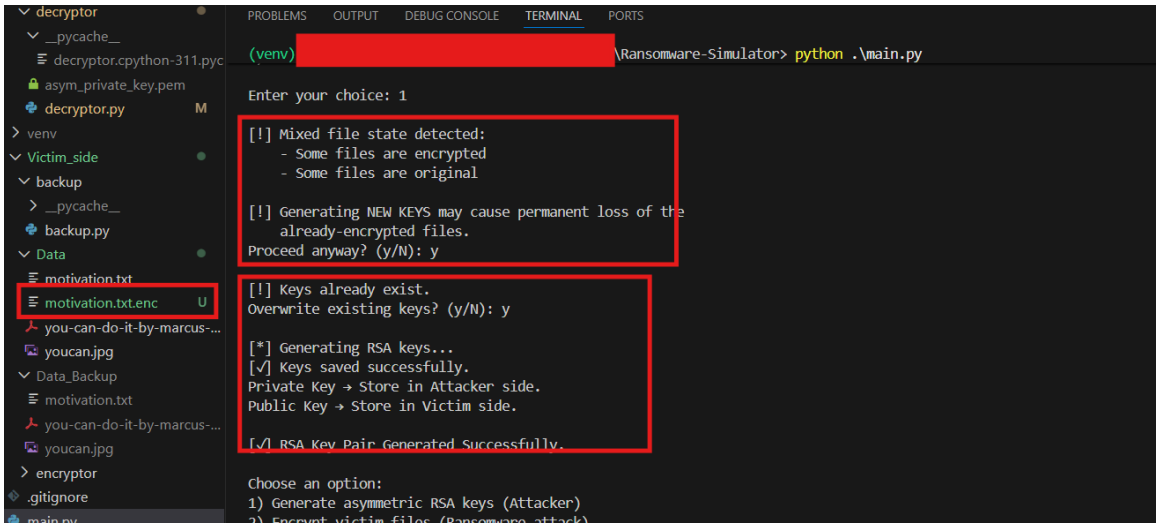
But in real ransomware, the victim can recover their data back by:

- Restore data from backup
- Use Free Decryption tools (If Available)
- Use data recovery software
- Rebuild system from scratch
- Engage professional incident response teams

- Paid ransom (not recommend, mostly attacker don't send your decrypted data back)

7. File-State Detection System

The simulator detects three file states: NONE, MIXED, and FULL. This system prevents double encryption, unsafe key overwrites



8. State Persistence

The simulator state is stored in a JSON file and automatically validated against the file system at runtime.

Path: Ransomware-Simulator\simulator_state.json

Point 7: File-State Detection System is load state from this point.

9. Version Control and Git Safety Configuration (.gitignore)

To ensure that the ransomware simulator remains safe, ethical, and suitable for academic use, a carefully designed .gitignore file is implemented. This configuration prevents sensitive files that are being created or that could be potentially harmful from being committed to a version control system like GitHub.

Proper version control hygiene is especially important for ransomware-related projects, as accidental publication of cryptographic keys or victim data could lead to misuse or misunderstanding.

9.1 Ignored Cryptographic Materials

The encryption key must not be sent to the data store. In real ransomware, the confidentiality of the private key is crucial. Ignoring the .pem file enforces proper key management and prevents accidental disclosure.

Line:

```
**/*.pem
```

All PEM-formatted files are excluded from version control. These files include:

- RSA private keys (asym_private_key.pem)
- RSA public keys (asym_public_key.pem)

9.2 Ignored Victim and Backup Data

Backup files are automatically created, may contain copies of the victim's data, so they are environment-specific and not part of the source code.

Line:

```
Victim_side/Data_Backup/
```

This will cause the backup directory to be excluded from Git tracking. Ignoring this directory prevents sensitive or unnecessary data from being uploaded and keeps the repository clean.

9.3 Ignored Simulator State Files

Line:

```
simulator_state.json
```

It is a dynamically changing state file, not source code, so if committed it can cause inconsistent behavior across different machines. By ignoring this file, each user starts with a clean and correct simulation state.

9.4 Ignored Virtual Environments

All Python virtual environments have been excluded by line:

```
venv/
.env
.venv
env/
```

9.5 Ignored Python Build and Cache Artifacts

The **.gitignore** includes a comprehensive Python template that excludes:

- Bytecode caches (__pycache__/, *.pyc)
- Build and distribution directories (build/, dist/, *.egg-info)
- Test and coverage artifacts (.pytest_cache/, .coverage)
- IDE-specific files (Spyder, PyCharm, Rope)
- Jupyter output and build files

These files do not contribute to the application logic and will unnecessarily clutter the repository.

10. Virtual Environment

To ensure dependency isolation, reproducibility, and system security, the Ransomware Simulator is designed to run in a Python virtual environment. The virtual environment creates a separate Python runtime that contains only the libraries required for the project, preventing conflicts with system-wide packages and other Python projects.

Using a virtual environment is especially important for security-focused projects like this simulator because it:

• Prevents accidental installation of cryptographic libraries into the global system

• Ensures consistent behavior across different machines and operating systems

• Reduces the risk of dependency version conflicts

• Makes the project safe to test and easy to clean

In this project, the virtual environment is typically created using the venv module that comes with Python. All required third-party libraries, including cryptographic packages, are installed only in this isolated environment.

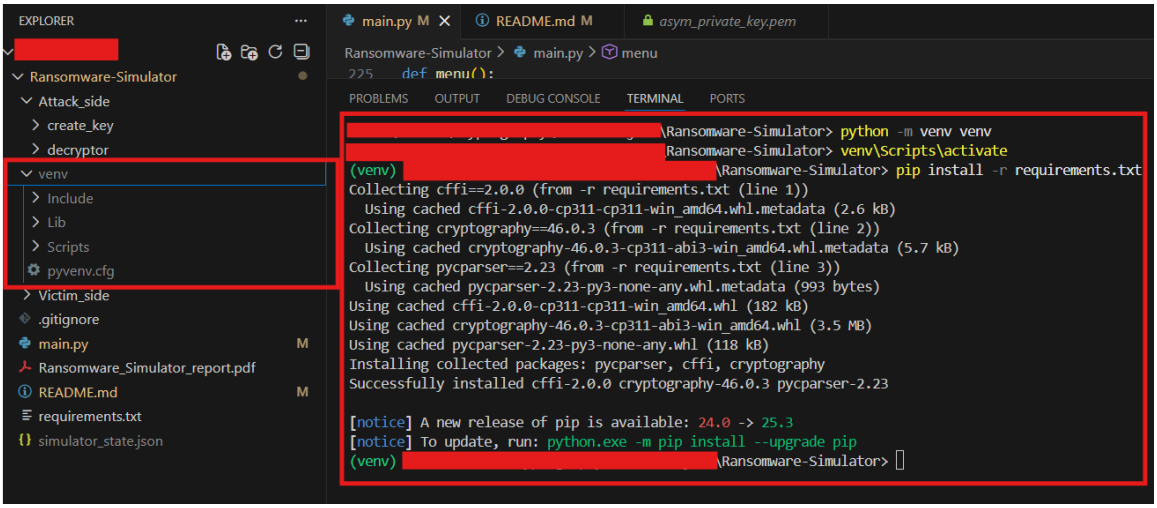
The virtual environment directory (for example, venv/, .venv/, or env/) is explicitly excluded from version control using the .gitignore configuration. This ensures that:

• Platform-specific binaries are not committed

• The database size remains small

• Each user creates their own clean environment

This approach follows industry best practices for Python development and reinforces the educational goal of demonstrating secure and professional software engineering practices along with cryptographic concepts.



IV. Usage Guide

This section explains how to build, configure, and run the Ransomware Simulator step by step.

1. System Requirements

To run this simulator, the system must meet the following requirements:

- Python version 3.9 or later
- Operating System: Windows, Linux, or macOS
- Internet connection is NOT required after setup

2. Building and Running the Program

Step 1: Clone the Repository

Run:

```
git clone https://github.com/Thin-Panha/Ransomware-Simulator.git
```

```
cd Ransomware-Simulator
```

Step 2: Create a Virtual Environment (Recommended)

Run :

```
python -m venv venv
```

To install Virtual Environment.

Then, activate Virtual Environment:

- **Windows (PowerShell),** run:

  ```
  venv\Scripts\activate
  ```

- **Linux / macOS,** run:

  ```
  source venv/bin/activate
  ```

Step 3: Install Dependencies

Install all required libraries using pip and the provided requirements.txt file.

Run:

```
pip install -r requirements.txt
```

Step 4: Prepare Victim Files

Before running the simulator:

- Navigate to Victim_side/Data/
- Make sure it has some files, such as: .txt, .pdf, .jpg

These files represent victim's data and should only contain dummy or academic content.

Step 5: Run the Simulator

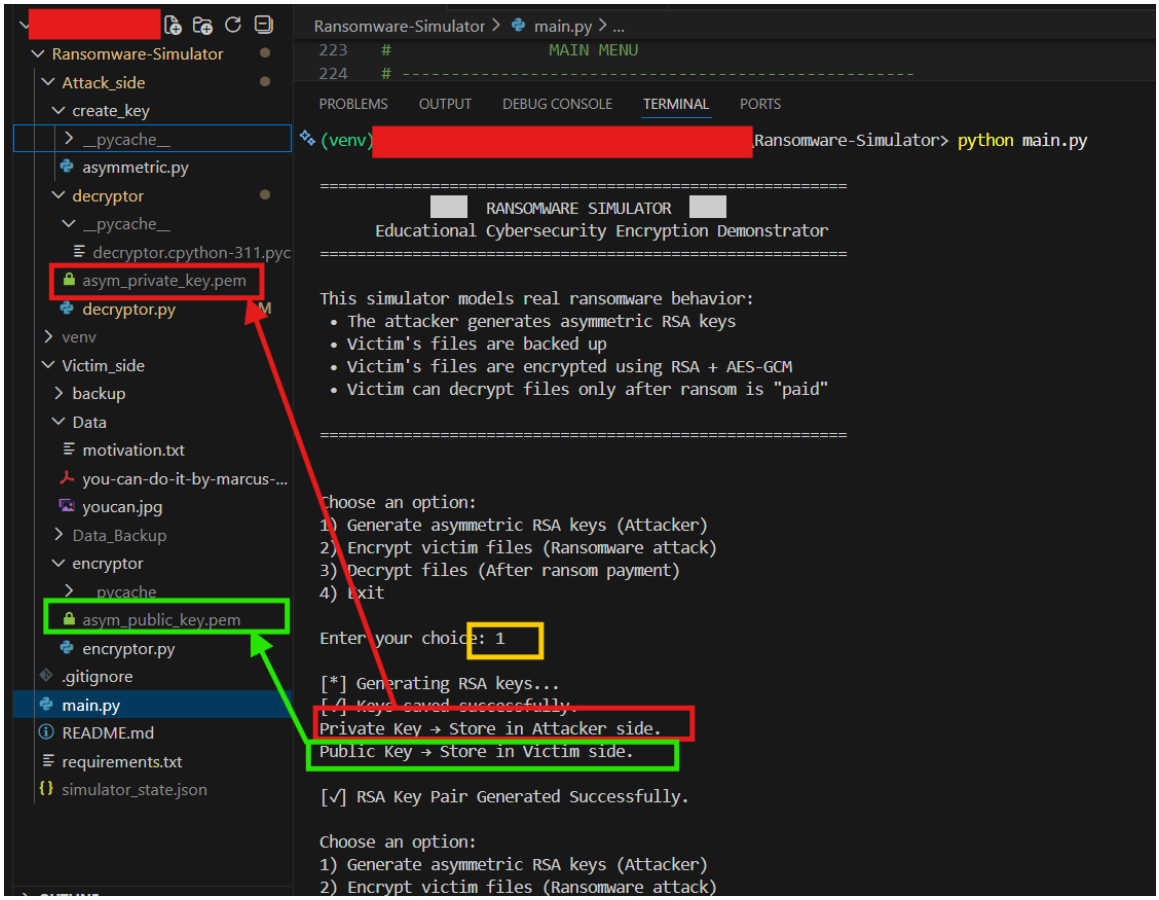
Execute the main program using the Python interpreter.

```
python main.py
```

3. Example Execution Flow

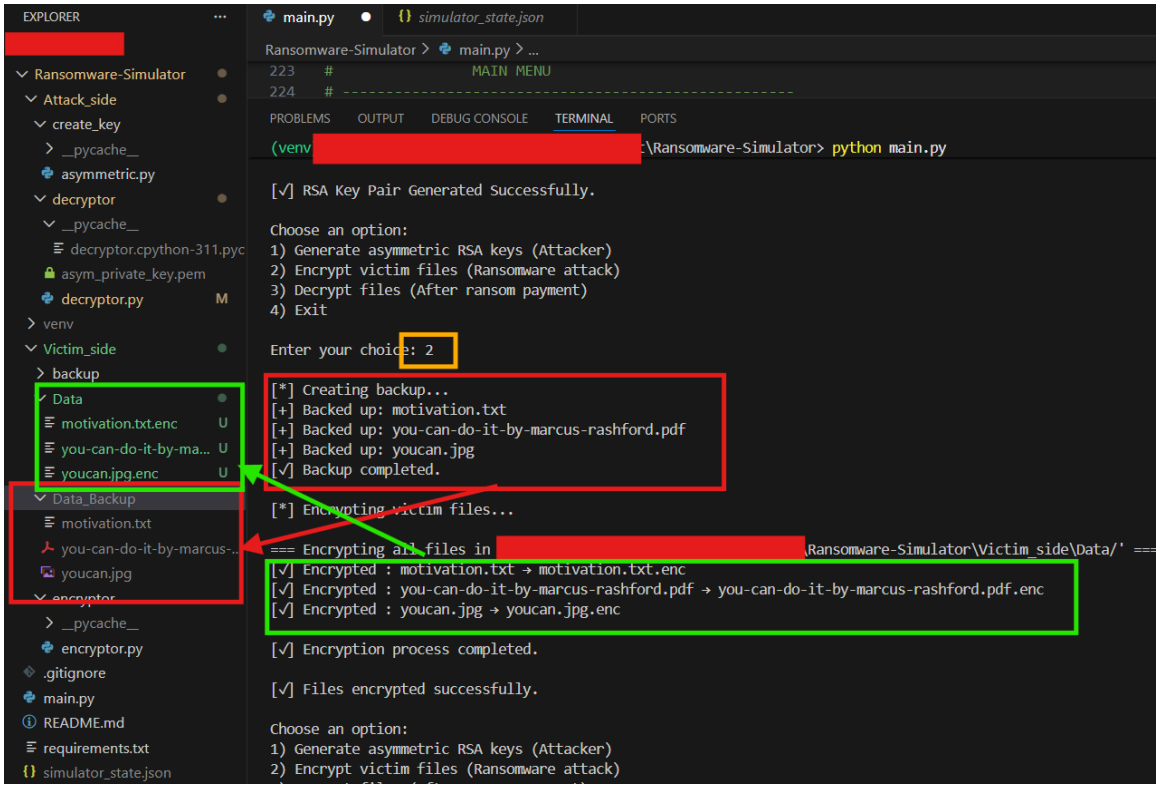
Example 1: Key Generation

The user selects option 1 to generate RSA keys. The system creates a 2048-bit RSA key pair and stores the private key on the attacker side and the public key on the victim side.



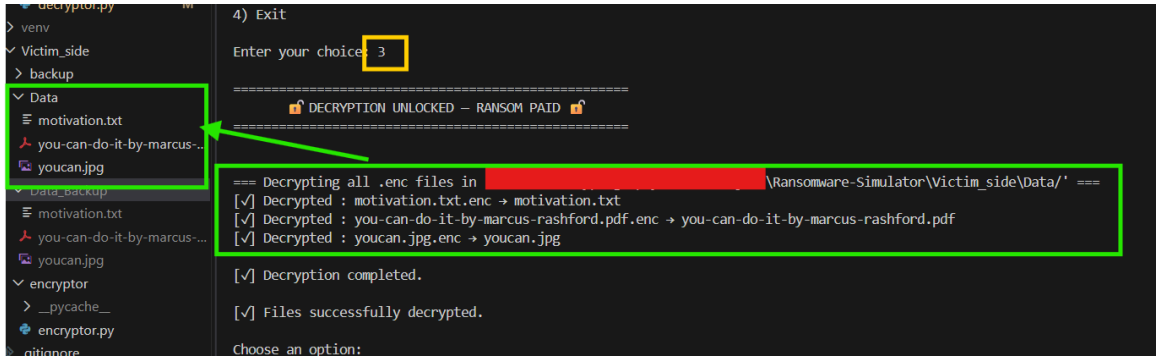
Example 2: Encryption Process

The user selects option 2 to encrypt victim files. The simulator automatically backs up files, encrypts them using hybrid encryption, and removes the original plaintext files and RSA public key from victim side.



Example 3: Decryption Process

The user selects option 3 to decrypt files after ransom payment. The private RSA key is used to recover AES keys and restore original files.



4. Result Summary

After encryption, all victim files are converted to .enc format and cannot be opened normally. After decryption, the original files are completely restored without any data corruption. This emulator prevents unsafe actions such as re-encrypting encrypted files or overwriting the key.

V. Conclusion and Future Work

1. Conclusion

This project successfully demonstrates the inner workings of modern ransomware in a safe and ethical manner. By implementing real cryptographic algorithms such as RSA-OAEP and AES-256-GCM, the simulator provides students with hands-on experience in complex encryption systems.

Modular design, file state detection, and security mechanisms ensure that the simulator closely mimics real ransomware behavior while preventing accidental attacks. The project combines theoretical cryptographic concepts with practical cybersecurity implementations in an effective manner.

2. Future Work

Several enhancements can be added to improve realism and learning value in future versions:

- Integration of a graphical user interface (GUI)
- Simulated command-and-control (C2) communication in a sandboxed environment
- Logging and forensic artifacts for DFIR analysis
- Performance analysis with large file sets
- Ransom payment system
- Simulation of different ransomware variants and encryption strategies
- Integration with automated testing frameworks

VI. References

1. https://docs.python.org/3/library/crypto.html
2. https://sip-international.com/blog/5-ways-to-recover-data-ransomware/