RSA Encryption Using SageMath

JU IT 4th year Project
Presented By : Group 27

Group Members -

Suvajit Sadhukhan (302211001005)
Sayan Das (302211001006)
Saugata Ghosh (302211001007)
Subhankar Das (002111001147)

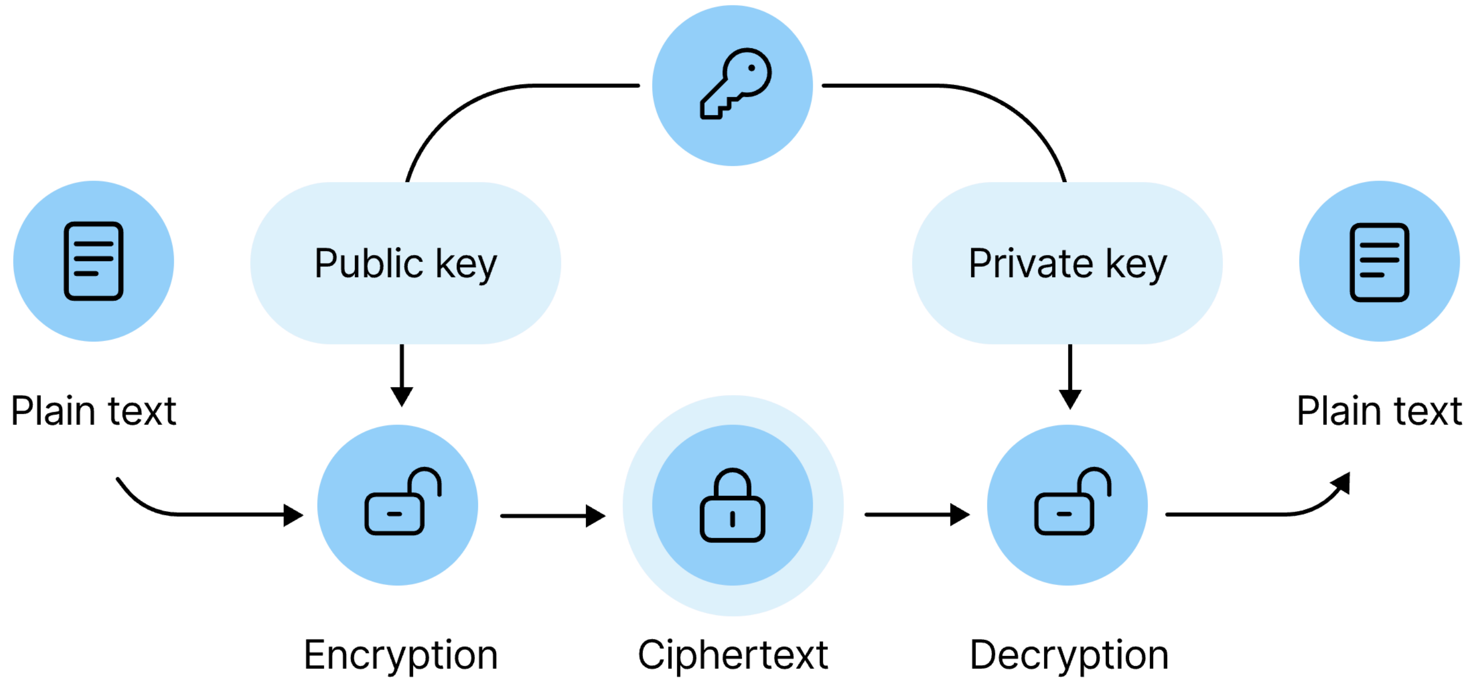
Guide : Mr. Utpal Kumar Roy
Assistant Professor

Content

Introduction to RSA Cryptography

- RSA (Rivest–Shamir–Adleman) is one of the earliest and most widely used public-key cryptographic algorithms. It is named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman, who introduced it in 1977.
- Enables secure data transmission and digital signatures.
- Its security relies on the computational difficulty of factoring very large integers.
- RSA is an asymmetric encryption algorithm, meaning it uses a pair of keys:
  - A ***public key*** for encryption (shared openly)
  - A ***private key*** for decryption (kept secret)

It secures data transmission by ensuring that only the intended recipient (who possesses the private key) can decrypt the encrypted message.



Project Components & Workflow

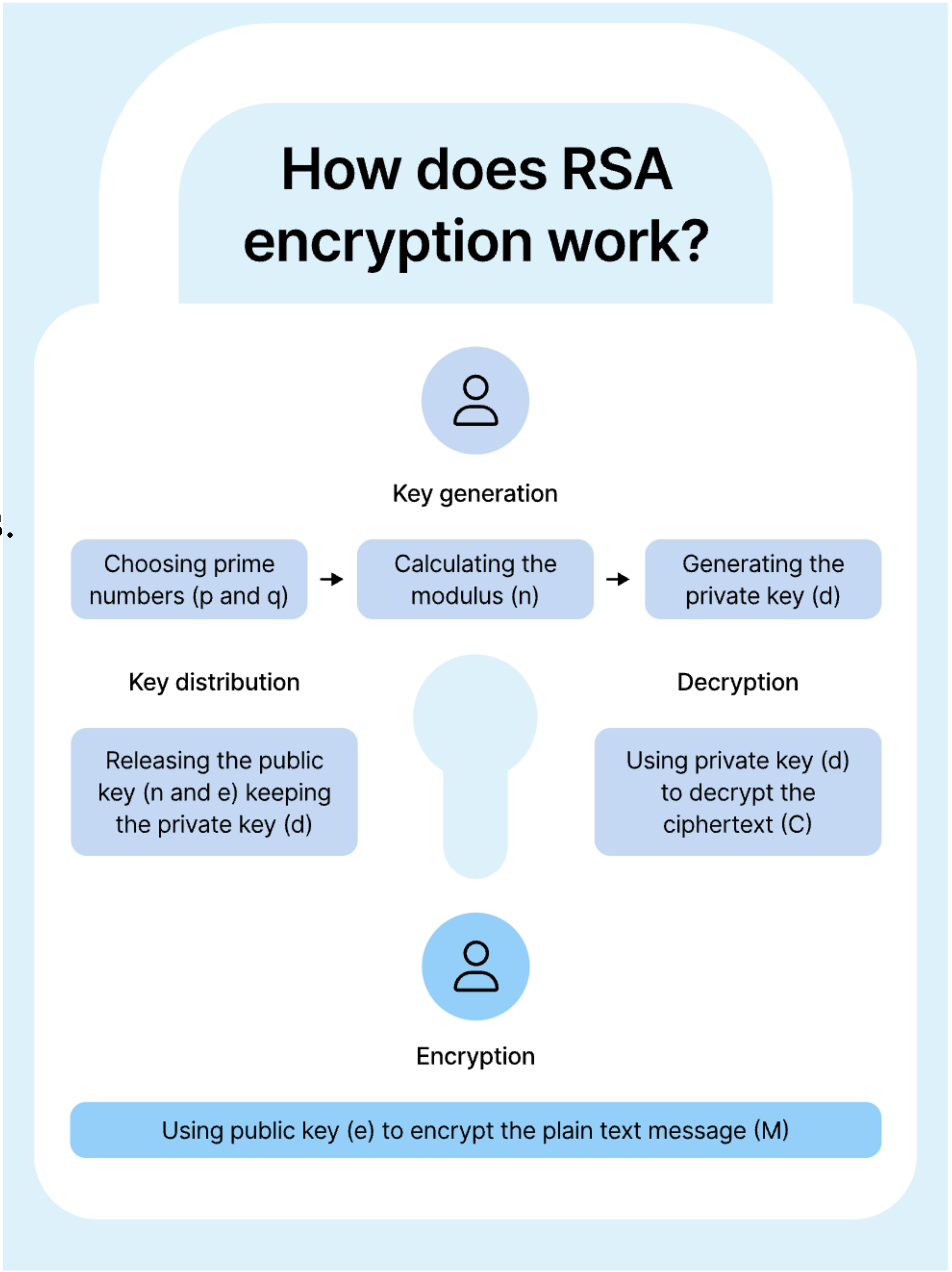
- **Overview:** This project implements the RSA cryptosystem through a suite of Python scripts (*leveraging SageMath for large number arithmetic, prime generation, modular inverse*) and a web-based interface for easy interaction.

- **Core Backend Scripts (SageMath):**

1. ***rsa_keygenerator.py***: Responsible for creating the RSA public and private key pairs.

2. ***rsa_encrypt.py***: Takes any input file (binary data) and a public key to produce encrypted ciphertext.

3. ***rsa_decrypt.py***: Takes ciphertext and the corresponding private key to recover the original binary file.


How does RSA encryption work?
Key generation
Choosing prime numbers (p and q)
Calculating the modulus (n)
Generating the private key (d)
Key distribution
Releasing the public key (n and e) keeping the private key (d)
Decryption
Using private key (d) to decrypt the ciphertext (C)
Encryption
Using public key (e) to encrypt the plain text message (M)

Contd…

- **Testing & Benchmarking Scripts:**

4. ***test_rsa.py***: A crucial driver script that automates testing of the entire encryption/ decryption pipeline.

5. ***benchmark_rsa.py***: Measures and logs the time taken for key generation, encryption, and decryption.

- **Frontend Web Application:**

6. ***streamlit_app.py***: A web application built with ***Streamlit*** providing a Graphical User Interface (GUI) for the RSA operations.

- **Typical Workflow (CLI & Web App):**

1. ***Command Line*****:** Execute scripts directly for batch processing or integration.

2. ***Web Application*****:** Use the Streamlit interface for interactive key generation, file upload/download, encryption/decryption of any file, and verification.

Mathematical Foundations

- ***Difficulty of Factoring:*** It's computationally easy to multiply p and q to get n, but extremely difficult to find p and q given only n (if p and q are large enough). This is the core of RSA's security.
- ***Modular Arithmetic:***
  Operations are performed within a finite set of integers (modulo n).
- ***Euler's Phi Function $\phi(n)$:***
  Number of positive Integer, a less than n and co-prime to n i.e., $\gcd(a, n) = 1$
  if n is product of two primes p and q then $\phi(n) = (p - 1) \cdot (q - 1)$
- ***Euler's Theorem :***
  if $\gcd(m, n) = 1$, then $m^{\phi(n)} \equiv 1 \mod n$
- ***Key Generation:***
  Choose e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$
  Compute $d = e^{-1} \mod \phi(n)$ [i.e., $e \cdot d \equiv 1 \mod \phi(n)$,
  then d is the modular inverse of e modulo $\phi(n)$].

Mathematical Foundations (contd)

- ***Encryption/Decryption Relationship:***
  Ciphertext, $c = m^e \mod n$; where, $m < n$
  decrypted data, $m' = c^d \mod n$
  $= (m^e)^d \mod n \equiv m^{ed} \mod n$
  Since, $ed \equiv 1 \mod \phi(n)$, we can write $ed = k \cdot \phi(n) + 1$ for some integer k.
  $$\begin{aligned} \therefore m' &= m^{k\cdot\phi(n)+1} \mod n \\ &= \left[m^{k\cdot\phi(n)} \mod n\right] \cdot \left[m \mod n\right] \\ &= \left[m^{\phi(n)} \mod n\right]^k \cdot \left[m \mod n\right] \\ &= (1)^k \cdot \left[m \mod n\right] \text{ (from \textit{Euler's theorem})} \\ &= m \mod n \end{aligned}$$
  $\therefore m' = m$
  (Here, m is the integer representation of a padded block of binary data).

- *It is practically possible to always get a message m such that* $\gcd(m, n) = 1$, *where $n = pq$ in RSA. But —*
  - The number of integers between 1 and $n - 1$ that are not co-prime to n is very small.
  - The only values of m for which $\gcd(m, n) \neq 1$ are the multiples of p or q,
  - Number of such bad m's is: $(n - \phi(n)) = p + q - 1$, which is tiny compared to n.
  - In practical uses, for large RSA keys (e.g., 2048-bit), this probability that a random m is co-prime to n is virtually 100 % .

Key Generation (*rsa_keygenerator.py*)

Input:

bit_length: Integer specifying the desired bit length for the ***prime numbers*** (e.g., 1024)

Output:

Two CSV files:

public_key.csv containing (e, n)

private_key.csv containing (d, n)

Steps

1. Validate Input
   - Check if exactly one argument (*bit_length*) is provided
   - Verify the argument is a positive integer

2. Generate Prime Numbers
   - For each prime p and q:
   - Generate a *random number* in the range $\left[2^{(bits-1)}, 2^{bits} - 1\right]$
   - Find the next prime number after the *random number*
   - Verify the prime has exactly the required bit length
   - Repeat until valid primes $(q\,! = p)$ are found

3. Compute Modulus and Totient
   - Calculate $n = p \cdot q$
   - Calculate $\phi(n) = (p - 1) \cdot (q - 1)$

Contd…

4. Select Public Exponent
    - Start with common value $e = 65537$
    - If $\gcd(e,\ \phi(n)) \neq 1$:
        - Find the next prime number after current e
        - Repeat until $\gcd(e,\ \phi(n)) = 1$

5. Calculate Private Exponent
    - Compute $d = e^{-1} \mod \phi(n)$ using modular inverse

6. Store Keys
    - Write (e, n) to public_key.csv
    - Write (d, n) to private_key.csv

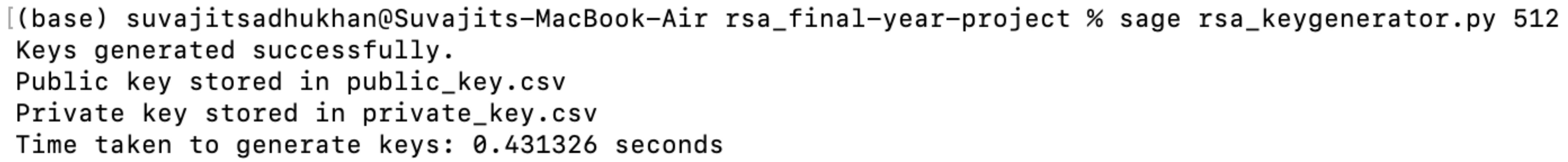
7. Output Results
    - Print success message and time taken

Why 65537 ?

- It is a Fermat prime number (specifically $F_4 = 2^{2^4} + 1$).
- It has only two 1's in its binary representation:
  $65537_{10} = 10000000000000001_2$
  This makes modular exponentiation fast, since exponentiation time depends on the number of 1's in the binary form.

Contd…

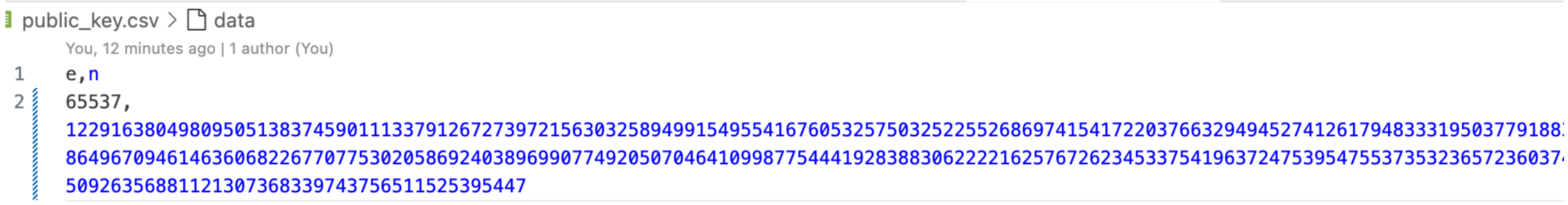
Command Line Invocation : *sage rsa_keygenerator.py 512*



```
[(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_keygenerator.py 512
 Keys generated successfully.
 Public key stored in public_key.csv
 Private key stored in private_key.csv
 Time taken to generate keys: 0.431326 seconds
```

Note:
here, 512 is the bit length of primes
∴ $n \approx 1024$ bits

Output files :
1. public_key.csv

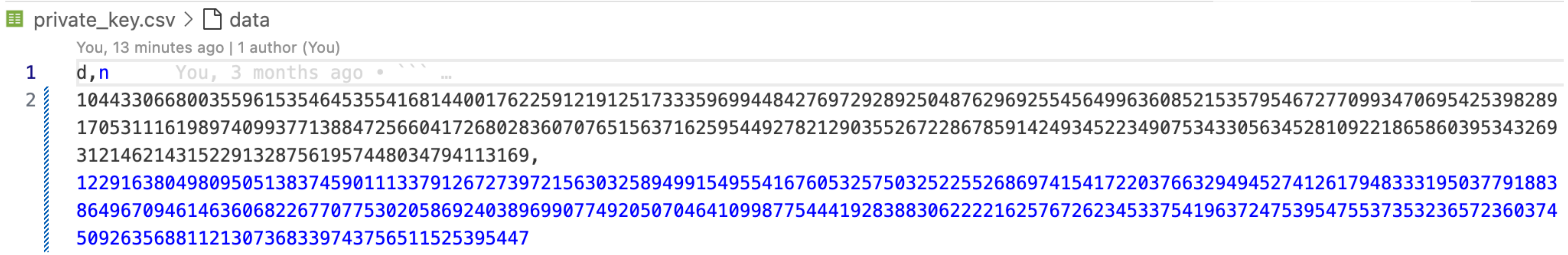


public_key.csv > data

You, 12 minutes ago | 1 author (You)

```
e,n
65537,
12291638049809505138374590111337912672739721563032589499154955416760532575032522552686974154172203766329494527412617948333195037791 88
86496709461463606822677077530205869240389699077492050704641099877544419283883062222162576726234533754196372475395475537353236572360 37
509263568811213073683397437565115253954 47
```

2. private_key.csv



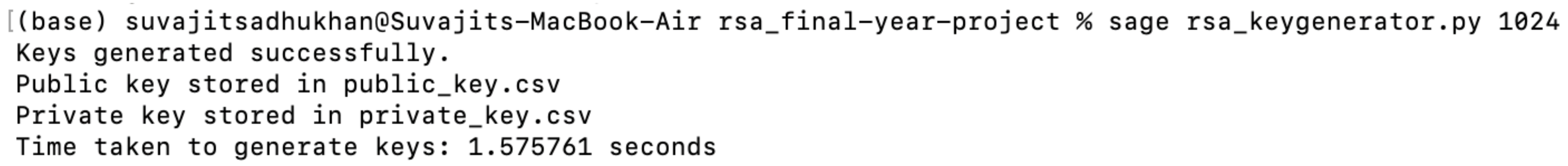
private_key.csv > data

You, 13 minutes ago | 1 author (You)

```
d,n      You, 3 months ago • ``` …
104433066800355961535464535541681440017622591219125173335969944842769729289250487629692554564996360852153579546727709934706954253 98289
170531116198974099377138847256604172680283607076515637162595449278212903552672286785914249345223490753433056345281092218658603953 43269
3121462143152291328756195744803479411 3169,
122916380498095051383745901113379126727397215630325894991549554167605325750325225526869741541722037663294945274126179483331950377 91883
864967094614636068226770775302058692403896990774920507046410998775444192838830622221625767262345337541963724753954755373532365723 60374
509263568811213073683397437565115253954 47
```

Contd…

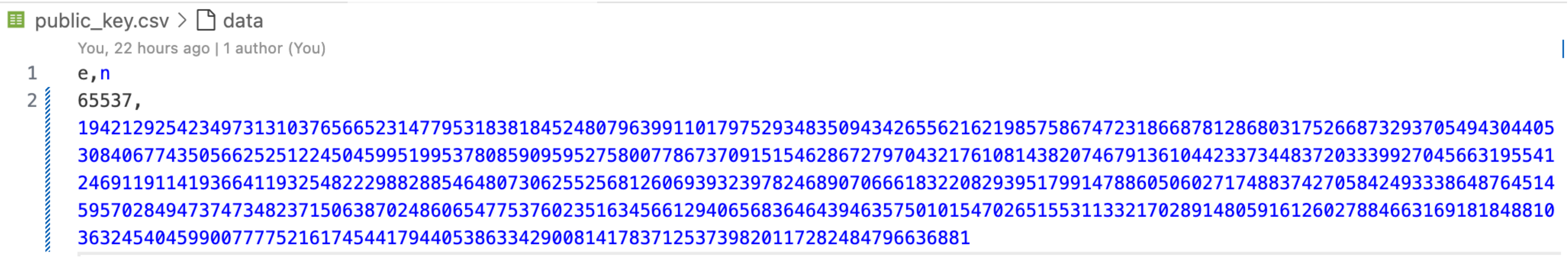
Another Instance of Command Line Invocation : sage rsa_keygenerator.py 1024



```
[(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_keygenerator.py 1024
 Keys generated successfully.
 Public key stored in public_key.csv
 Private key stored in private_key.csv
 Time taken to generate keys: 1.575761 seconds
```

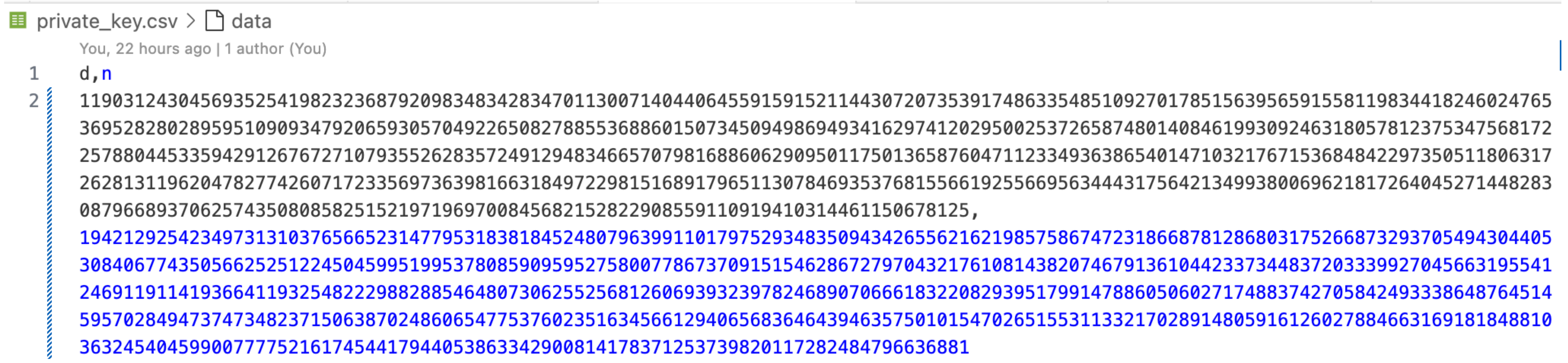
Note:
here, 1024 is the bit length of primes
$\therefore n \approx 2048$ bits

Output files :
1. public_key.csv



```
e,n
65537,
19421292542349731310376566523147795318381845248079639911017975293483509434265562162198575867472318668781286803175266873293705494304405
30840677435056625251224504599519953780859095952758007786737091515462867279704321761081438207467913610442337344837203339927045663195541
24691191141936641193254822298828854648073062552568126069393239782468907066618322082939517991478860506027174883742705842493338648764514
59570284947374734823715063870248606547753760235163456612940656836464394635750101547026515531133217028914805916126027884663169181848810
363245404599007777521617454417944053863342900814178371253739820117282484796636881
```

2. private_key.csv



```
d,n
11903124304569352541982323687920983483428347011300714044064559159152114430720735391748633548510927017851563956591558119834418246024765
36952828028959510909347920659305704922650827885536886015073450949869493416297412029500253726587480140846199309246318057812375347568172
25788044533594291267672710793552628357249129483466570798168860629095011750136587604711233493638654014710321767153684842297350511806317
26281311962047827742607172335697363981663184972298151689179651130784693537681556619255669563444317564213499380069621817264045271448283
087966893706257435080858251521971969700845682152822908559110919410314461150678125,
19421292542349731310376566523147795318381845248079639911017975293483509434265562162198575867472318668781286803175266873293705494304405
30840677435056625251224504599519953780859095952758007786737091515462867279704321761081438207467913610442337344837203339927045663195541
24691191141936641193254822298828854648073062552568126069393239782468907066618322082939517991478860506027174883742705842493338648764514
59570284947374734823715063870248606547753760235163456612940656836464394635750101547026515531133217028914805916126027884663169181848810
363245404599007777521617454417944053863342900814178371253739820117282484796636881
```

RSA Encryption (*rsa_encrypt.py*)

Input:

input_filename: File to be encrypted (any format)
public_key_csv: CSV file containing public key (e, n)

Output:

[input_filename]_cipher.[ext]: File containing ciphertext integers (one per line)

Steps

1. Read Public Key
   - Open the *public_key_csv* file.
   - Extract the RSA public key (e, n).
   - Validate that the key format is correct.

2. Read Input File
   - Open the file in binary mode.
   - Store the file contents as a byte sequence.

3. Calculate Block_Size
   - Compute maximum *block_size* in bytes: *(n.nbits() - 1) // 8*
   - Verify *block_size* ≥ *15* bytes (*1 byte header + 14 bytes tail padding*)
   - Calculate available *data_size = block_size - 15*

4. File Chunking
   - Split input file bytes into *chunks* of maximum size = *min(data_size, 255) bytes.*
     (255 limit due to 1-byte length header)

Contd…

5. Block Processing (per *chunk*)
   - Create header byte with chunk length (L)
   - Add intermediate padding of random bytes if *chunk* < *data_size*
   - Append *14-byte* random tail padding
   - Calculate:

     block_bytes = header + chunk + intermediate padding (if needed) + tail padding
   - Verify *length* of *block_byte to block_size*, it should match
   - Convert *block_byte* (padded block) to an to integer (big-endian)

6. Encryption
   - Compute ciphertext integer for each padded block: $c = m^e \mod n$
   - Store ciphered block (integer) in a list

7. Output
   - Generate output filename by appending "_*cipher*" before extension
   - Write ciphertext integers (one per line) to output file
   - Print encryption summary including timing

RSA Decryption (*rsa_decrypt.py*)

Input:

ciphertext_filename: File containing RSA-encrypted integer ciphertexts.
public_key_csv: CSV file containing RSA private key (d, n)

Output:

[original_filename]_decrypted[ext]: Reconstructed original binary file

Steps

1. Read Private Key
   - Open the *public_key_csv* file.
   - Extract RSA private key (d, n).
   - Validate that the key format is correct.

2. Read Ciphertext File
   - Check existence of *ciphertext_filename*.
   - Read ciphertext file line by line.
   - Convert each line into an integer (c).

3. Detemine Block_Size
   - Compute *block_size* in bytes: *(n.nbits() - 1) // 8*
   - Ensure block size meets the encryption padding constraints (*block_size* ≥ *15* bytes)

Contd…

4. Decrypt: for each Ciphertext Integer (c)
   - Compute decrypted integer $m' = c^d \mod n$
   - Convert decrypted integer into *block_size* bytes.
   - Extract *header* (1st byte), which gives *L* (actual data length).
   - Extract the next *L* bytes as the *valid data chunk*.
   - Discard remaining random padding.
   - Store *recovered data*.

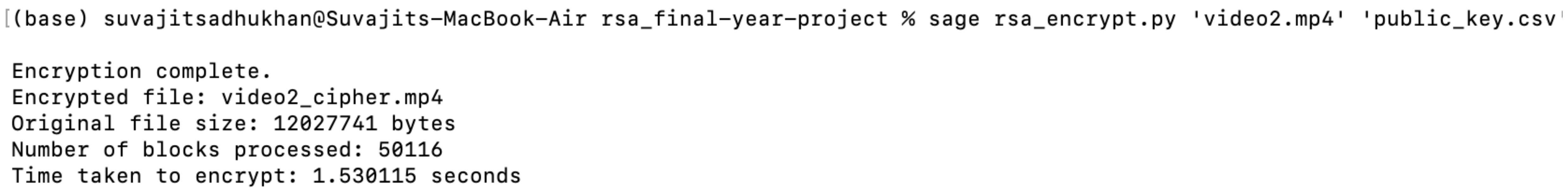
5. Data Reconstruction
   - Concatenate all *recovered data chunks*.

6. Output File Generation
   - Generate an output filename by replacing *_cipher* with *_decrypted.*
   - Write reconstructed binary data into the output file.

7. Print Summary
   - Print decryption summary including timing

Command Line Invocation: Encryption/Decryption

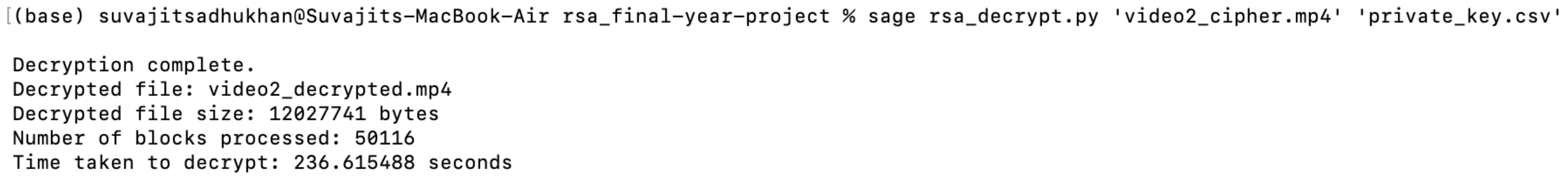
Encryption using 1024 bit primes, i.e., $n \approx 2048$ bits :
Usage: sage rsa_encrypt.py <input_filename> <public_key_csv>



```
[(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_encrypt.py 'video2.mp4' 'public_key.csv'

Encryption complete.
Encrypted file: video2_cipher.mp4
Original file size: 12027741 bytes
Number of blocks processed: 50116
Time taken to encrypt: 1.530115 seconds
```

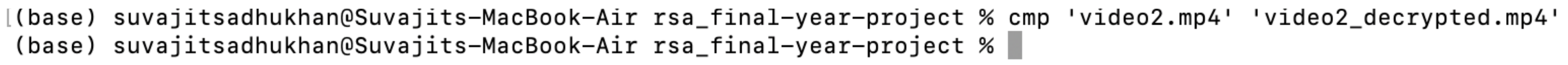
Decryption using corresponding private key of 2048 bit RSA:
Usage: sage rsa_decrypt.py <ciphertext_filename> <private_key_csv>



```
[(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_decrypt.py 'video2_cipher.mp4' 'private_key.csv'

Decryption complete.
Decrypted file: video2_decrypted.mp4
Decrypted file size: 12027741 bytes
Number of blocks processed: 50116
Time taken to decrypt: 236.615488 seconds
```

we compare the original input file and decrypted file.



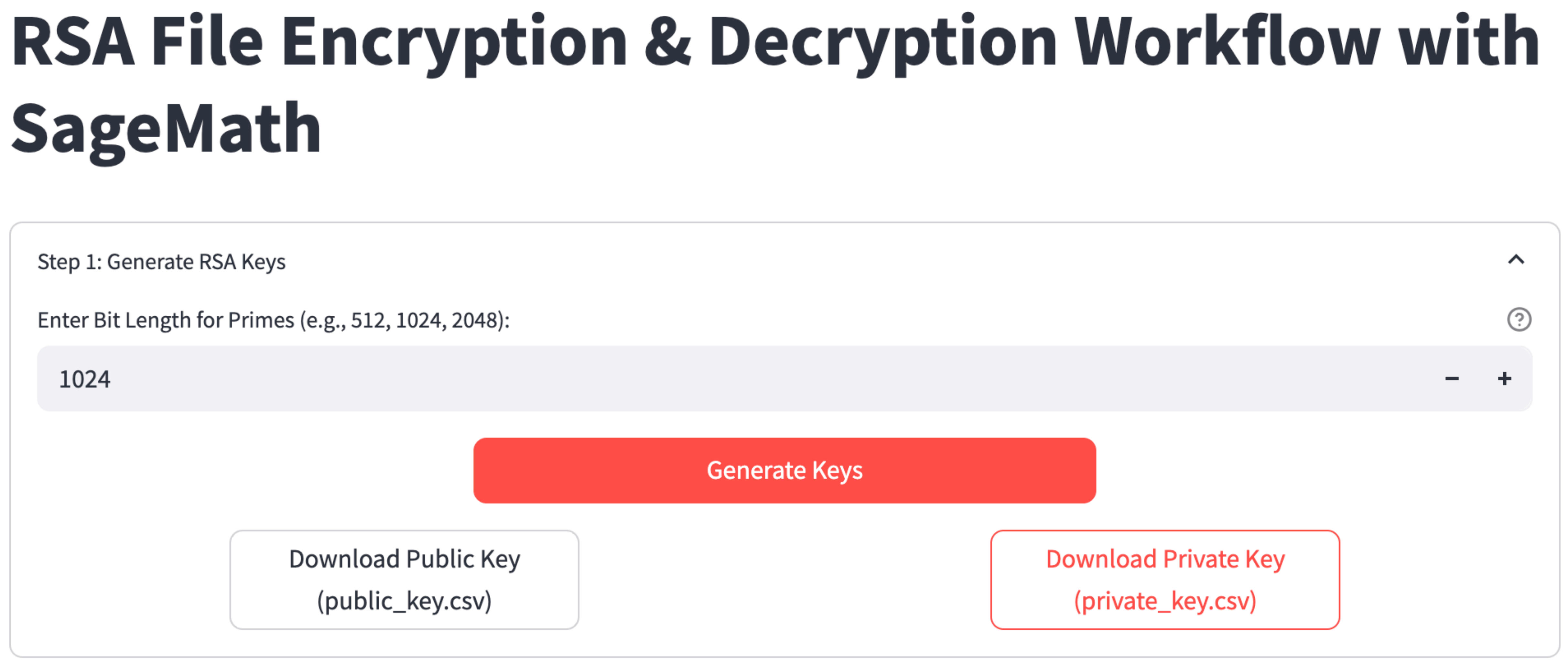
```
[(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % cmp 'video2.mp4' 'video2_decrypted.mp4'
(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % 
```

We get no error(s), that means we our encryption/decryption process is successfully done. And we reconstructed our original file successfully.

GUI using Streamlit

The Streamlit GUI offers a simple, step-by-step process for users.
Hosted Link - https://huggingface.co/spaces/Nobita69/RSA-Cryptography-Tool

1. Key Generation



RSA File Encryption & Decryption Workflow with SageMath

Step 1: Generate RSA Keys

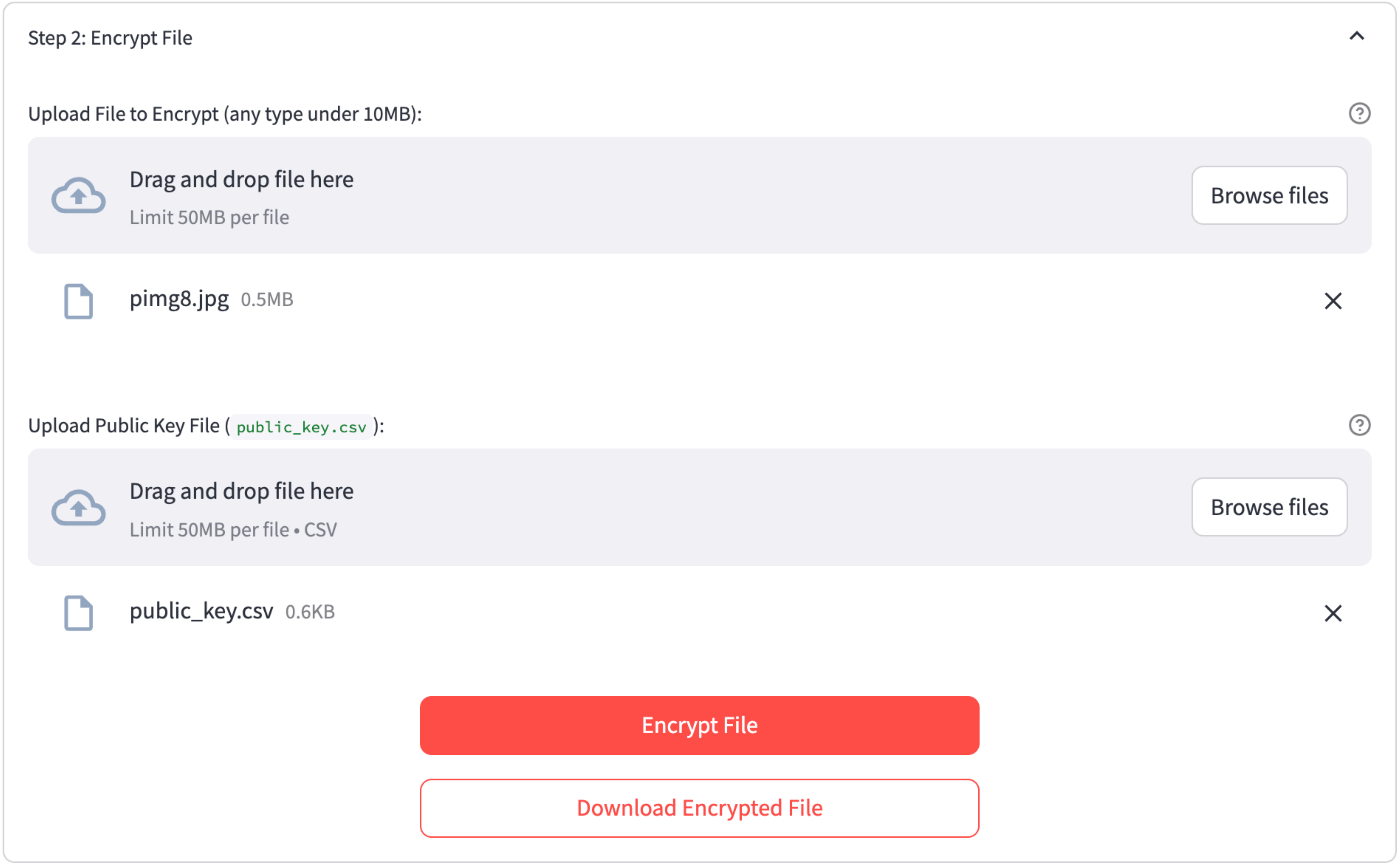
Enter Bit Length for Primes (e.g., 512, 1024, 2048):

1024

Generate Keys
Download Public Key (public_key.csv)
Download Private Key (private_key.csv)

GUI

2. Encryption


Step 2: Encrypt File

Upload File to Encrypt (any type under 10MB):

Drag and drop file here
Limit 50MB per file

Browse files

pimg8.jpg 0.5MB

Upload Public Key File (`public_key.csv`):

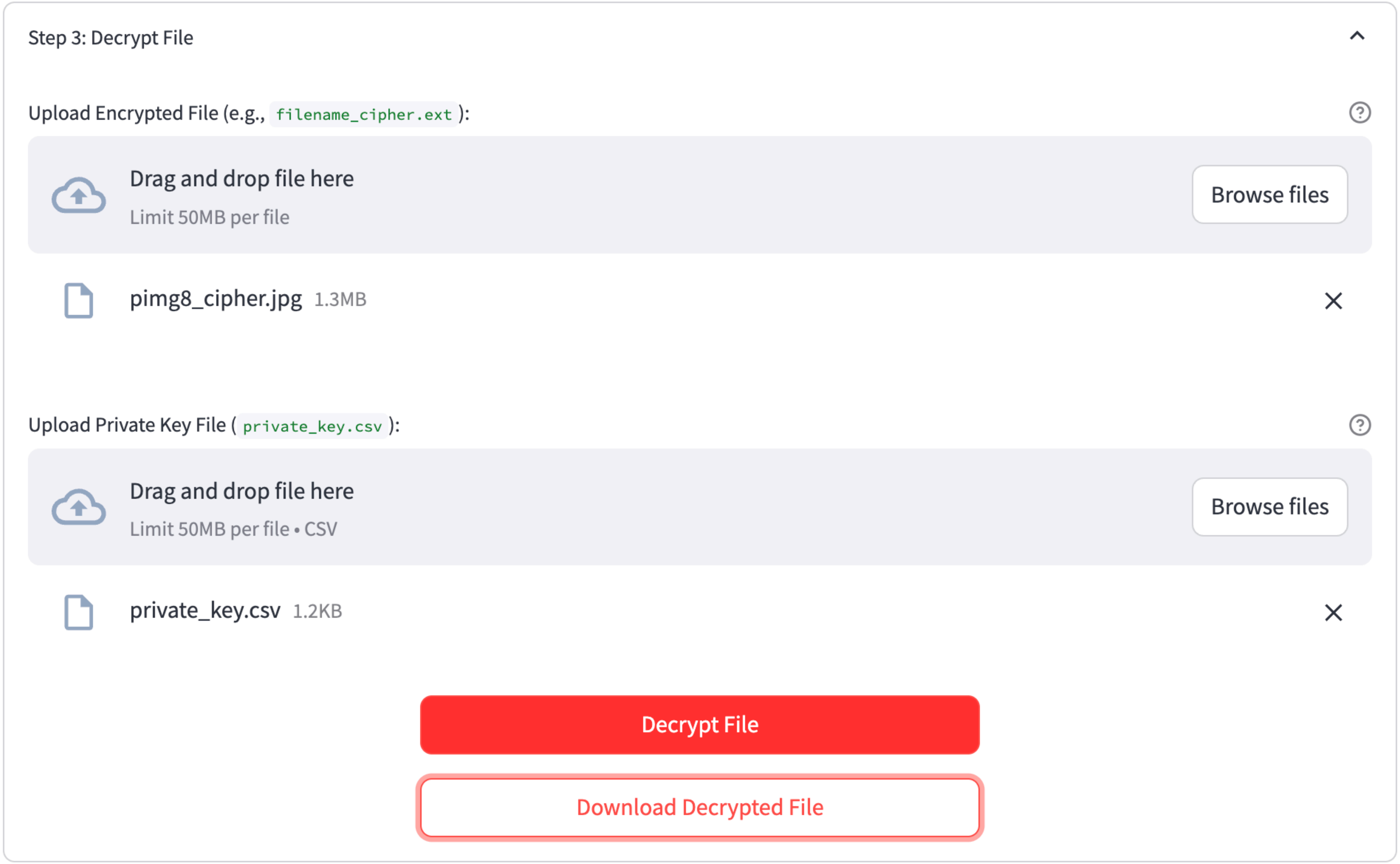
Drag and drop file here
Limit 50MB per file • CSV

Browse files

public_key.csv 0.6KB

GUI

3. Decryption



Step 3: Decrypt File

Upload Encrypted File (e.g., `filename_cipher.ext`):

Drag and drop file here

Limit 50MB per file

Browse files

pimg8_cipher.jpg 1.3MB

Upload Private Key File (`private_key.csv`):

Drag and drop file here

Limit 50MB per file • CSV

Browse files

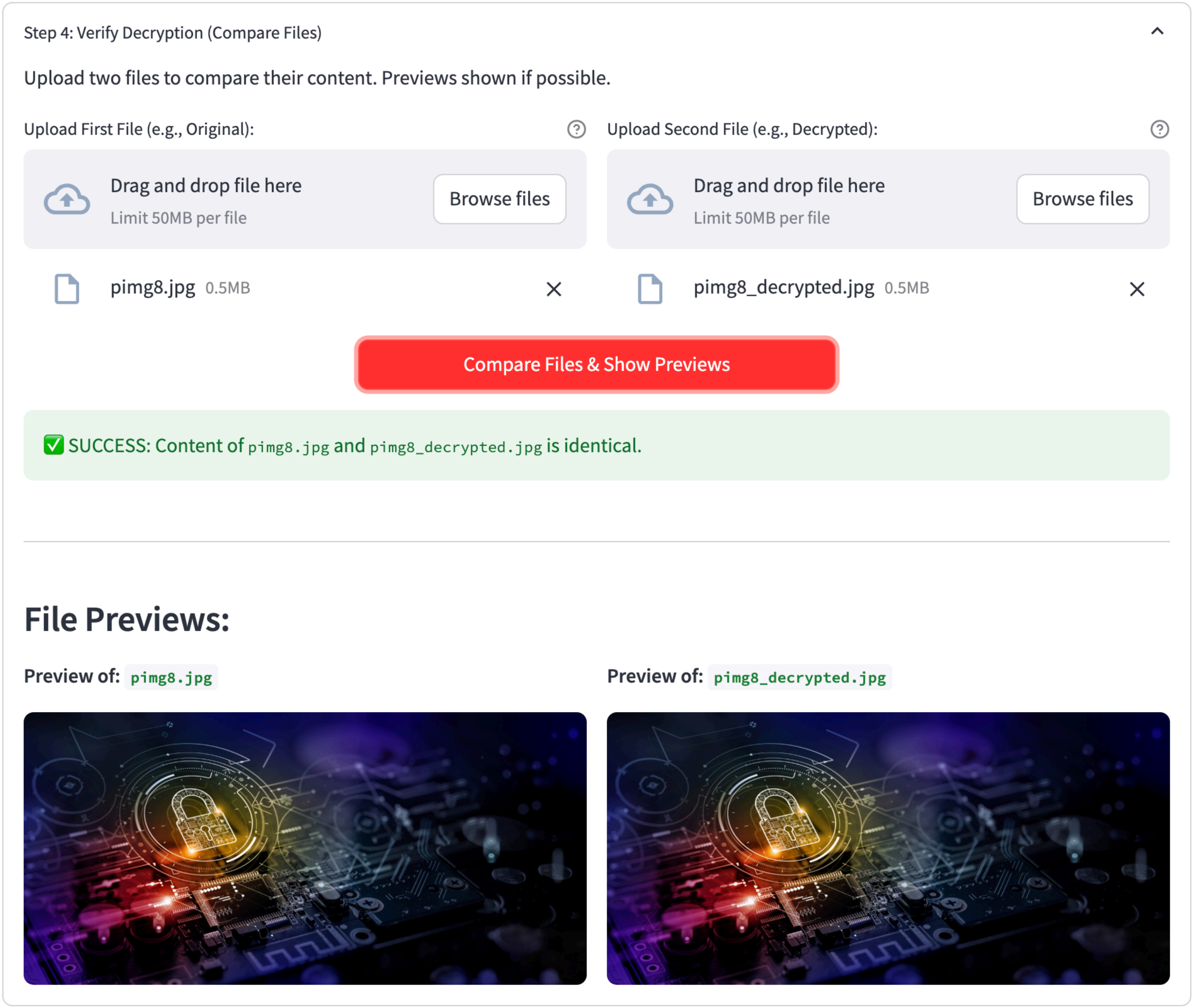
private_key.csv 1.2KB

Decrypt File

Download Decrypted File

GUI

4. Comparing Original File with Decrypted File



Step 4: Verify Decryption (Compare Files)

Upload two files to compare their content. Previews shown if possible.

Upload First File (e.g., Original):

Drag and drop file here
Limit 50MB per file
Browse files

pimg8.jpg 0.5MB

Upload Second File (e.g., Decrypted):

Drag and drop file here
Limit 50MB per file
Browse files

pimg8_decrypted.jpg 0.5MB

Compare Files & Show Previews

✅ SUCCESS: Content of `pimg8.jpg` and `pimg8_decrypted.jpg` is identical.

File Previews:

Preview of: `pimg8.jpg`

Preview of: `pimg8_decrypted.jpg`

Testing the Implementation (*test_rsa.py*)

Purpose: To rigorously verify that the RSA encryption and decryption scripts (the backend logic) function correctly and reliably under a variety of conditions and inputs.

Methodology:

Group 1: Normal Key Size *(1024-bit primes for p and q, resulting ≈ 2048-bit modulus n)*

- Key Generation: Runs *rsa_keygenerator.py 1024*.
- ***data_area*** Calculation.

Here, *data_area = block_size -15*
where, *block_size* = (i/p bit's length -1) // 8

- Tests with messages (byte sequences) of varying lengths:
  1. Exactly *data_area* bytes.
  2. *data_area – 1* bytes.
  3. *data_area + 1* bytes (multi-block).
  4. Very small message (few bytes).
  5. Large message (1MB random bytes, simulating generic binary data).
- Compares original byte sequence with decrypted byte sequence.
- Result: All tests successful, original data perfectly reconstructed.

Group 2: Small Key Size *(64-bit primes for p and q, resulting ≈ 128-bit modulus n)*

- Key Generation: Runs *rsa_keygenerator.py 64*.
- ***data_area*** becomes 0.
- Encryption attempt is expected to fail due to insufficient block size for padding.

Benchmarking RSA Project

Purpose: A *benchmark_rsa.py* script is dedicatedly created to quantitatively measure and analyze the time performance of the three core RSA operations (key generation, encryption, decryption) executed by the backend SageMath scripts.

1. Key Generation Benchmark: Systematically runs *rsa_keygenerator.py* for prime bit lengths: 512 to 2048 bits with increasing step of 256 bits.

Key Size (bits for p, q)	***Time Taken (seconds)***	***Modulus Size n (approx. bits)***
512	3.170814	1024
768	4.356060	1536
1024	3.894331	2048
1280	5.500485	2560
1536	9.544675	3072
1792	13.020152	3584
2048	22.300396	4096

2. Encryption Benchmark (using fixed 1024-bit prime keys): Encrypts plaintext files of sizes: 1KB, 10KB, 100KB, 1MB, 10MB and records encryption time for each.

Message Size	***Encryption Time (seconds)***	***Number of Blocks (Approx. for 255-byte block_area)***
1KB	2.239249	~4-5 blocks
10KB	2.275603	~40 blocks
100KB	2.305799	~400 blocks
1MB	2.407949	~4000 blocks
10MB	3.861074	~40000 blocks

Benchmarking RSA Project

3. Decryption Benchmark (using fixed 1024-bit prime keys):

Decrypts previous ciphertexts using private key of corresponding public key and records decryption time for each.

Message Size	***Decryption Time (seconds)***	***Number of Blocks (Approx.)***
1KB	2.258703	~4-5 blocks
10KB	2.560639	~40 blocks
100KB	4.344418	~400 blocks
1MB	22.933321	~4000 blocks
10MB	332.388417	~40000 blocks

Handling Different File Types and Performance Considerations:

The *type* of the file (e.g, .txt, .pdf, .jpg, .png, .mp3, .mp4, executables) does not directly alter the mathematical complexity of the RSA operations per block once the file is read as a sequence of bytes. For instance, a 1MB text file and a 1MB PNG image file, having the same byte count, will take a very similar amount of time to encrypt or decrypt using the same RSA key.

Security Analysis of RSA: Types of Attacks and Defence Mechanisms

1. Factoring Attacks:

- Directly attempting to factor the modulus n using algorithms like GNFS.
- The defence relies entirely on the user choosing a sufficiently large *bit_length* when running *rsa_keygenerator.py*. Our RSA can generate upto 4096-bit RSA key (most secure).

2. Low Public Exponent Attacks:

- If a small public exponent e (like) is used *without* proper padding, attacks can recover the plaintext m if $m^e < n$.
- The *rsa_keygenerator.py* script defaults to , which is not considered cryptographically "small" in this context. Furthermore, the custom padding scheme ensures that the integer m fed into the exponentiation is always large.

3. Chosen Ciphertext Attacks (CCA):

- The attacker obtains the decryption of chosen ciphertexts to deduce information about the private key or decrypt a target ciphertext.
- The custom padding scheme implemented in *rsa_encrypt.py* (*1-byte length header + message + random intermediate padding + 14-byte random tail*) provides quite a good defence to this attack up to some extent. However, OAEP is the best padding Scheme.

Contd…

4. Common Modulus Attack:

- Occurs if multiple users share the same n but have different pairs.
- This attack is related to key management, not the core algorithm implementation. The defence relies on users generating unique keys for each distinct entity.

5. Implementation Errors / Side-Channel Attacks:

- Flaws in software/hardware leaking information (power analysis, cache timing).
- Relies heavily on the security of the underlying Python interpreter, SageMath libraries, and the operating system.

6. Timing Attacks:

- A type of side-channel attack measuring the precise time taken for decryption operations. Variations in time can leak information about the private exponent.
- This project relies on SageMath's *power_mod* function. It implement countermeasures such constant-time modular exponentiation techniques.

7. Poor Random Number Generation:

- Predictable random numbers compromise prime generation (p, q) and padding randomness.
- The custom padding in rsa_encrypt.py uses *os.urandom(),* is generally considered a cryptographically secure pseudo-random number generator as it draws entropy from the operating system. SageMath's *randint()* used for prime candidate generation, also relies on a secure RNG.

Conclusion & Future Work

Achievements:

- Functional command-line RSA implementation for any file type (binary data).
- Interactive Streamlit web app for user-friendly RSA demonstration with diverse files.
- Robust testing suite and performance benchmarks.

Key Learnings:

- Practical understanding RSA mechanics for binary data.
- Importance of padding.
- Performance characteristics of asymmetric cryptography.
- Wrapping CLI tools with a Streamlit GUI for broader accessibility.

Potential Future Work:

- Implement Standardized Padding (OAEP) - *High Priority*.
- Decryption Optimization (CRT).
- Web App Enhancements (robust error handling, direct text input, secure deployment).
- Attack Exploration. Key Management Features. Hybrid Cryptosystem.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, 1978. [Online]. Available: https://people.csail.mit.edu/rivest/Rsapaper.pdf

[2] C. Paar, and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer, 2010. [Online]. Available: https://uim.fei.stuba.sk/wp-content/uploads/2018/02/Understanding_Cryptography_Chptr_7-The_RSA_Cryptosystem.pdf

[3] D. Boneh, *Twenty Years of Attacks on the RSA Cryptosystem*, Notices of the AMS, 1999. [Online]. Available: https://www.ams.org/notices/199902/boneh.pdf

[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996. [Online]. Available: https://dl.icdst.org/pdfs/files3/f7ba35bf7149b541644785c9270cc6b8.pdf

[5] S. A. A. Shah, M. A. Gondal, and M. Hussain, "Systematic and Critical Review of RSA Based Public Key Cryptographic Schemes: Past and Present Status," *ResearchGate*, 2021. [Online]. Available: https://www.researchgate.net/publication/356372929

[6] A. A. A. Yousif, and M. A. Maarof, "Methods toward Enhancing RSA Algorithm: A Survey," *SSRN Electronic Journal*, 2019. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3412776

[7] D. J. Bernstein, N. Heninger, P. Lou, and L. Valenta, "Post-Quantum RSA," *IACR Cryptology ePrint Archive*, 2017. [Online]. Available: https://eprint.iacr.org/2017/351.pdf

[8] A. Tuteja, and A. Shrivastava, "A Literature Review of Some Modern RSA Variants," *International Journal for Scientific Research & Development (IJSRD)*, 2014. [Online]. Available: https://ijsrd.com/articles/IJSRDV2I8134.pdf

[9] S. Sowjanya, and K. S. Rao, "A Study and Performance Analysis of RSA Algorithm," *International Journal of Computer Science and Mobile Computing (IJCSMC)*, 2013. [Online]. Available: https://ijcsmc.com/docs/papers/June2013/V2I6201330.pdf

Q&A

Thank you for your attention!

We are now open for Questions.

Live demo of the *GUI App* / *GitHub* Repository

Group Members -

Suvajit Sadhukhan
Sayan Das
Saugata Ghosh
Subhankar Das