RSA Encryption Using SageMath: A Practical Implementation and Analysis

Sayan Das[1], Saugata Ghosh[1], Suvajit Sadhukhan[1], Subhankar Das[1]

[1] Department of Information Technology, Faculty of Engineering and Technology, Jadavpur University, Salt Lake Campus, E-mail: enquiry.jums@jadavpuruniversity.in

Abstract- This paper details the implementation and analysis of the RSA public-key crypto-system using SageMath and Python. The project encompasses the complete lifecycle of RSA, including robust key generation, message encryption with custom block padding, and subsequent decryption. SageMath's advanced number theory capabilities are leveraged for efficient generation of large prime numbers, modular arithmetic, and inverse computations, crucial for RSA's security. The implementation was rigorously tested for functional correctness across various message sizes and key lengths. Performance benchmarks were conducted to evaluate the computational costs of key generation, encryption, and decryption, highlighting the inherent asymmetry in RSA operations. The security of the implementation is discussed, considering its reliance on the difficulty of integer factorisation and the chosen parameters, while also acknowledging the limitations of the custom padding scheme used compared to standardised approaches like OAEP. This work provides a practical understanding of RSA's core principles and its real-world performance characteristics.

Index Terms- Cryptography, Decryption, Encryption, Public-key Cryptography, RSA, SageMath, factoring attacks, timing attacks, chosen ciphertext attacks (CCA), low public exponent attacks, side-channel attacks, poor random number generation (RNG).

I. INTRODUCTION

Cryptography is fundamental to modern digital security, enabling confidential communication and the safeguarding of sensitive data. As digital interactions proliferate, the necessity for robust encryption systems becomes paramount [1]. The RSA (Rivest–Shamir–Adleman) algorithm, a pioneering public-key cryptosystem, remains a cornerstone in securing digital communications, leveraging number theory and modular arithmetic for secure key exchange and message encryption. This project was motivated by the importance of understanding and implementing such cryptographic systems, especially in an era of increasing cyber threats.

The primary aim of this project is to develop a comprehensive understanding of the RSA algorithm by implementing it programmatically, evaluating its performance characteristics, and analysing its security against known threats. By constructing an RSA system from its foundational components, this work seeks to provide a hands-on insight into cryptographic principles and demonstrate how mathematical theory translates into practical data protection. The need for such a project stems from the educational value of deeply understanding widely-used security protocols, which are often treated as black boxes.

The implementation utilises Python for scripting and orchestration, integrated with SageMath, an open-source mathematical software system. SageMath's capabilities in handling large number arithmetic, prime number generation, and modular operations are pivotal to the project [10]. On a societal scale, understanding and improving upon systems like RSA contributes to enhanced data privacy and security in countless applications, from secure web browsing (HTTPS) to encrypted email and digital signatures, thereby fostering trust in digital infrastructures.

II. BASIC CONCEPTS/TECHNOLOGY USED

The RSA algorithm's security and functionality are rooted in several key mathematical concepts and are implemented using specific software technologies. Understanding these is crucial for appreciating the project's mechanics.

At the heart of RSA are **prime numbers**. Two distinct, large prime numbers, denoted as p and q, are chosen. Their product, $n = p \cdot q$, forms the modulus, which is a public component of both the public and private keys. The security of RSA heavily relies on the computational difficulty of factoring this large integer n back into its prime components p and q [1].

Another critical concept is **Euler's Totient Function**, denoted as $\phi(n)$. For RSA, where $n = p \cdot q$, $\phi(n) = (p-1)(q-1)$. This function counts the number of positive integers less than n that are relatively prime to n. $\phi(n)$ is essential for calculating the private key exponent.

The RSA system employs a **public key** and a **private key**. The public key consists of the modulus n and a public exponent e, typically a small prime number like 65537, which is co-prime to $\phi(n)$ (i.e., $gcd(e, \phi(n)) = 1$). The private key consists of the same modulus n and a private exponent d, which is the modular multiplicative inverse of e modulo $\phi(n)$ That is, $e \cdot d \equiv 1 \left(\mod \phi(n) \right)$. Deriving d without knowing p and q (and thus $\phi(n)$) is computationally infeasible if n is large enough.

Modular Arithmetic forms the basis of RSA operations. Encryption of a message M (represented as an integer) is performed as $C = M^e \mod n$, where C is the cipher-text. Decryption of cipher-text C is performed as $M = C^d \mod n$. These operations ensure that the results remain within a manageable range and provide the one-way function property critical for security.

SageMath [10] was chosen as a core technology due to its extensive support for number theory and cryptographical mathematics. It provides built-in functions for generating large random prime numbers (*next_prime*, *randint*), computing the greatest common divisor (*gcd*), finding modular inverses (inverse_mod), and performing efficient modular exponentiation (*power_mod*) on arbitrarily large integers. This simplifies the implementation of complex mathematical operations that are otherwise cumbersome and error-prone to code from scratch.

Python [11] served as the primary programming language for scripting the overall workflow, including file handling (reading plaintext, writing cipher-text, storing keys in CSV files), string and byte manipulation (encoding text to bytes for processing and decoding back), implementing the message blocking and custom padding logic, and orchestrating calls to SageMath functions. Standard Python libraries like csv, os, sys, and time were used for these tasks.

The combination of Python's scripting versatility and SageMath's specialised mathematical engine provided a robust and efficient environment for developing and analysing the RSA crypto-system. A conceptual block diagram of the RSA process is shown below (conceptual, as actual diagram generation is not possible here):

[Plaintext] $\longrightarrow$ [Blocking & Padding] $\longrightarrow$ [M (integer)] $\xrightarrow{(e,n)}$ [C (integer)] $\xrightarrow{(d,n)}$ [M' (integer)] $\longrightarrow$ [Depadding & Reassembly] $\longrightarrow$ [Plaintext']

III. STUDY OF SIMILAR PROJECTS OR TECHNOLOGY / LITERATURE REVIEW

The RSA algorithm, since its inception by Rivest, Shamir, and Adleman in 1978 [1], has been the subject of extensive research and numerous implementations. A review of existing literature reveals various focuses, from theoretical enhancements to security analyses and practical performance evaluations.

Several studies, such as those by Shah et al. [5] and Yousif & Maarof [6], provide comprehensive surveys of RSA-based schemes and methods proposed to enhance its security or performance. These often discuss modifications like using three or more primes, or alternative key generation approaches, though not all are practical for general use due to increased complexity or overhead [4]. Paar and Pelzl [2] offer a detailed textbook explanation of RSA's operations and security considerations, forming a foundational understanding.

Dan Boneh's work [3] on "Twenty Years of Attacks on the RSA Cryptosystem" highlights various vulnerabilities that can arise from improper implementation or parameter choices, emphasising the importance of secure padding schemes and appropriate key management, topics also covered in the "Handbook of Applied Cryptography" by Menezes et al. [4]. These works underscore that while the mathematical basis of RSA is strong, its practical security heavily depends on careful implementation.

Performance analyses, like the one by Sowjanya and Rao [9], have examined RSA execution times with varying key sizes, often noting the performance impact of larger keys. However, many older studies may not fully account for modern computational optimisations or specific library performance like SageMath's. More recent concerns involve the threat of quantum computing, leading to research in Post-Quantum RSA [7], which explores parameters resistant to quantum attacks, though often at the cost of significantly larger key sizes.

This project contributes by providing a practical, step-by-step implementation using a modern tool (SageMath), complete with functionality testing and performance benchmarking for key generation, encryption, and decryption using a custom, albeit non-standard, padding scheme. While many libraries offer RSA, building it from components with SageMath offers deeper insight. The unique aspect is the detailed walkthrough of this specific SageMath-based implementation and its performance profile, rather than proposing a novel RSA variant.

IV. PROPOSED MODEL / TOOL

The proposed system is a functional implementation of the RSA cryptosystem, realised through a suite of Python scripts leveraging SageMath for core cryptographic operations. The system is divided into three main components: Key Generation, Encryption, and Decryption.

1. Key Generation Model:
The key generation process (*rsa_keygenerator.py*) takes a desired bit length (e.g., 512, 1024) for the prime numbers as input.

- **Prime Generation:** Two distinct large prime numbers, p and q, of the specified bit length are generated. This uses SageMath's *randint()* to select a candidate within the range $[2^{(bits-1)}, 2^{bits} - 1]$ and then *next_prime()* to find the first prime greater than or equal to the candidate. A check *p.nbits() == bits* ensures the prime is of the exact bit length.
- **Modulus Calculation:** The modulus n is computed as $n = p \cdot q$.
- **Euler's Totient:** $\phi(n)$ is calculated as $(p-1) \cdot (q-1)$.
- **Public Exponent Selection:** The public exponent e is typically set to 65537 (a common Fermat prime $F_4 = 2^{16} + 1$). It is verified that $gcd(e, \phi(n)) = 1$. If not, *next_prime(e)* is used iteratively until a coprime e is found.
- **Private Exponent Calculation:** The private exponent d is computed as the modular multiplicative inverse of e modulo $\phi(n)$, (i.e., d = *inverse_mod(e,* $\phi(n)$) using SageMath.
- **Key Storage:** The public key (e, n) and private key (d, n) are stored in separate *.csv* files (*public_key.csv, private_key.csv*).

A conceptual block diagram for key generation:
[Bit Length Input] ⟶ [Generate p, q of given bit size] ⟶ [Compute n, $\phi(n)$] ⟶ [Select e, Compute d] ⟶ [Store Public Key (e, n), Private Key (d, n)]

2. Encryption Tool (*rsa_encrypt.py*):
This script takes a plaintext file and the public key file as input.

- **Read Public Key:** e and n are read from *public_key.csv*.
- **Read Plaintext:** The plaintext file is read and its content is encoded into UTF-8 bytes.
- **Block Sizing:**
  - *block_size = (n.nbits() − 1) // 8* (maximum bytes per block to ensure $m < n$).
  - *data_size = block_size − 15* (1 byte for length header, 14 bytes for tail padding).
- **Padding and Blocking:** The plaintext bytes are divided into chunks of at most data_size. Each chunk undergoes custom padding:
  - **Header (1 byte):** Stores the actual length L of the message data in the current chunk.
  - **Message Chunk (L bytes):** The actual message data.
  - **Intermediate Random Padding (*data_size - L bytes*):** If $L <$ *data_size*, random bytes from *os.urandom()* are added to fill up to *data_size*.
  - **Tail Random Padding (14 bytes):** 14 random bytes from *os.urandom()* are appended.

- The final *block_bytes* (*Header* + *Chunk* + *Intermediate Padding* + *Tail Padding*) has a fixed length of *block_size*.

- **Integer Conversion:** Each block_bytes is converted into a large integer m_int (big-endian).
- **Modular Exponentiation:** The cipher-text integer *c_int* is computed as *c_int = power_mod(m_int, e, n)*.
- **Store Ciphertext:** Each *c_int* is written to an output cipher-text file, one integer per line.

3. Decryption Tool (*rsa_decrypt.py*):
This script takes a cipher-text file and the private key file as input.

- **Read Private Key:** d and n are read from *private_key.csv*.
- **Read Ciphertext:** Cipher-text integers *c_int* are read line by line from the file.
- **Block Size Determination:** *block_size* is determined as in encryption.
- **Modular Exponentiation:** For each *c_int*, the padded message block integer *m_int* is recovered as *m_int = power_mod(c_int, d, n)*.
- **Byte Conversion:** *m_int* is converted back to *block_bytes* of length *block_size* (big-endian).
- **De-padding:**
  - The first byte *L = block_bytes[0]* (header) gives the length of the original message part.
  - The original message part is extracted as *message_part = block_bytes[1 : 1+L]*. The intermediate and tail padding are implicitly discarded.
- **Concatenate & Decode:** All extracted message_parts are concatenated. The resulting byte sequence is decoded from UTF-8 to retrieve the original plaintext string.
- **Store Plaintext:** The decrypted plaintext is written to an output file.

This model provides a complete RSA workflow, from key creation to secure message exchange, with specific attention to handling message data through blocking and a custom padding mechanism.

V. IMPLEMENTATION AND RESULTS

The RSA cryptosystem described in Section IV was implemented using Python 3.9+ and SageMath 9.3. The implementation's correctness and performance were evaluated through systematic testing and benchmarking.

Functional Correctness Testing:
A dedicated test script (*test_rsa.py*) was developed to automate the verification process.

- **Case Study 1: Standard Key Size (*1024-bit primes*):**
  - Keys were generated using 1024-bit primes (resulting in a ~2048-bit modulus n).
  - The block_size and data_area (space for message within a block before padding) were calculated.
  - Several messages of varying lengths were tested:
    - Message length exactly equal to data_area.
    - Message one byte shorter/longer than data_area.
    - Very short message (e.g., "HelloRSA!X").
    - Large message (1 MB of random characters).
  - For each message, it was encrypted and then decrypted. The final decrypted message was compared to the original.
  - **Findings:** All tests with the 1024-bit key pair resulted in perfect reconstruction of the original plaintext, confirming the functional correctness of the key generation, encryption (including padding and blocking), and decryption processes.

- **Case Study 2: Small Key Size (*64-bit primes - Edge Case*):**
  - Keys were generated using 64-bit primes (modulus n ~128 bits).
  - This resulted in a block_size of 15 bytes, leading to a data_area of 0 due to the 15-byte padding overhead.
  - Encryption was attempted with a dummy message.
  - **Findings:** The encryption script correctly exited with an error, indicating that the block size was too small for the padding requirements. This confirmed that the implementation handles this predictable error condition appropriately.

Performance Benchmarking (*benchmark_rsa.py*):

- **Task 1: Key Generation Time:**
  Time taken to generate RSA key pairs for prime bit lengths: 512, 768, 1024, 1280, 1536, 1792, and 2048 bits.
  - **Findings:**
    - 512 bits: ~0.41s
    - 1024 bits: ~1.61s (A separate 1024-bit generation for Enc/Dec tasks took ~2.07s)
    - 2048 bits: ~19.84s
  - **Analysis:** Key generation time generally increased with key size, attributed to the increased difficulty for SageMath's next_prime() to find larger primes. Minor variations (e.g., 1024 bits sometimes faster than 768 bits in one run) can occur due to random prime search variations or system load.
- **Task 2: Encryption Time (*1024-bit primes, e=65537*):**
  Time taken to encrypt messages of 1KB, 10KB, 100KB, 1MB, and 10MB.
  - **Findings:**
    - 1KB: ~0.0002s
    - 1MB: ~0.125s
    - 10MB: ~1.241s
  - **Analysis:** Encryption time increased relatively linearly with message size. The small public exponent e=65537 results in fast modular exponentiation. The time is dominated by the number of blocks processed and I/O overhead for larger files.
- **Task 3: Decryption Time (*1024-bit primes*):**
  Time taken to decrypt corresponding cipher-texts for message sizes 1KB, 10KB, 100KB, 1MB, and 10MB.
  - **Findings:**
    - 1KB: ~0.024s
    - 1MB: ~20.618s
    - 10MB: ~229.986s
  - **Analysis:** Decryption time increased significantly and non-linearly with message size. The private exponent d is large (comparable to n), making C^d mod n computationally much more intensive than encryption. Decryption is clearly the performance bottleneck in RSA for large data volumes.

These results confirm the expected behaviour of RSA: functional correctness under typical conditions and the characteristic performance asymmetry between encryption and decryption.

VI. CONCLUSION

This project successfully implemented the RSA public-key cryptosystem, covering key generation, encryption, and decryption, utilising Python and the SageMath computer algebra system. The functional correctness of the implementation was validated through comprehensive test cases, demonstrating its ability to reliably encrypt and decrypt messages of various sizes. Performance benchmarks provided quantitative insights into the computational costs: key generation time scales with key size, encryption is relatively fast with a standard small public exponent, and decryption is significantly more computationally intensive, representing the primary performance bottleneck.

The security analysis reaffirmed that RSA's strength lies in the difficulty of factoring the large modulus n, underscoring the necessity of using sufficiently large key sizes (e.g., 2048-bit modulus or larger). While the implementation incorporates defences like a standard public exponent (65537) and secure random number generation (*os.urandom*) for padding, its reliance on a custom padding scheme is a notable limitation. This custom scheme, while adding randomness, does not offer the provable security guarantees against attacks like Chosen Ciphertext Attacks (CCA) that standardised schemes like OAEP (Optimal Asymmetric Encryption Padding) provide.

In conclusion, this project offers a practical demonstration of RSA principles, its implementation intricacies using SageMath, and its performance profile. For future work, the most critical enhancement would be to replace the custom padding with OAEP to align with industry best practices and significantly improve security. Further exploration could include implementing decryption optimisations like the Chinese Remainder Theorem (CRT) and comparing the SageMath-based implementation against other cryptographic libraries.

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

[10] SageMath, *(n.d.). Open-source mathematics software system for algebra and cryptography*. [Online]. Available: https://www.sagemath.org

[11] Python Software Foundation, *(n.d.). Python Language Reference, Version 3.x*. [Online]. Available: https://www.python.org/