IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Mathematical Modelling of Engineering Problems
Vol. 12, No. 6, June, 2025, pp. 2085-2096
Journal homepage: http://iieta.org/journals/mmep

MORSA: An innovative Modified and Optimized RSA Framework Using Parallel Computing Environment

Prashnatita Pal[1*], Bikash Chandra Sahana[1], Jayanta Poray[2]

[1] Department of Electronics and Communication Engineering, National Institute of Technology, Patna 800005, India
[2] Department of Computer Science and Engineering, Techno India University, West Bengal 700091, India

Corresponding Author Email: prashnatitap@gmail.com

https://doi.org/10.18280/mmep.120624

Received: 5 March 2025
Revised: 19 May 2025
Accepted: 28 May 2025
Available online: 30 June 2025

Keywords:
decryption, encryption, modified Rivest-Shamir-Adleman algorithm, Particle Swarm Optimization (PSO), security, public key, parallel computing

ABSTRACT

In this modern era of digital communication and data transmission, the information security and confidentiality are inevitable. Among many other alternatives, the Rivest-Shamir-Adleman (RSA) cryptosystem has set a new security benchmark and is a well-known and public key encryption model. However, as computing power increases dramatically, standard RSA confronts new challenges and weaknesses, necessitating more security safeguards and performance enhancements. This paper presents the improvement of the existing state-of-the-art model through the design of Modified and Optimized Rivest-Shamir-Adleman (MORSA) Cryptosystem, enhancing the security of the sensitive data against severe cyber-attacks. MORSA addresses the shortcomings of the original RSA algorithm through various improvements. Firstly, the MORSA algorithm generates larger keys that are more resistant and secure against brute force and factoring attacks. Secondly, the research examines optimal prime number generating techniques for large prime selection. Additionally, the MORSA cryptosystem uses a dynamic modulus selection technique that adjusts the modulus depending on input data size to improve encryption speed while maintaining security. Finally, parallel computer resources are utilized to accelerate encryption and decryption. The study then analyses efficiency and compares to traditional RSA and other modern encryption algorithms. MORSA performs better in encryption speed, resource usage, and cryptographic attack resistance.

1. INTRODUCTION

Throughout in today's interconnected world, the secure transfer of information is of the highest importance to safeguard sensitive data from falling into wrong hands. The rapid advancements in computing technology have led to an important growth in digital communication, making it essential to employ robust encryption techniques that can withstand sophisticated cyber threats. The Rivest-Shamir-Adleman (RSA) cryptosystem [1], based on the mathematical properties of prime numbers, has been a key component of public-key encryption for decades. However, as computing power continues to evolve, traditional RSA faces new challenges that require innovative methods to maintain its effectiveness and security. This paper presents an innovative cryptographic solution, the Modified and Optimized RSA Cryptosystem (MORSA), designed to address the limitations of conventional RSA and enhance the security of information transfer. MORSA seeks to enhance the encryption process and safeguard data against modern cyber adversaries.

Our work focuses on under-studied RSA models, which are upgraded or include modified forms of the regular RSA algorithm. In this work, interventions are activities like modified or improved, and other phrases that describe RSA cryptosystem. The Outcomes determine whether the upgraded RSA methods provide considerable benefits.

Attacks against RSA may be roughly divided into two groups [1]. The first group comprises mathematical purpose. The second category feat the faults and weaknesses in the execution of the function. The directly target the underlying mathematical function attack are many types, such as low-decryption exponent attack [2], Factorization Attacks [2], Partial Key Exposure Attack [2], Wiener's low private exponent attack [3], Common modulus attack [3]. Secondly, the faults and weaknesses in the implementation of the functions are Attack with Timing [3], Evaluation of Power [3], Side-Channel Attacks and Chosen-Ciphertext Attacks (CCA) [3]. The counter and preventative measures of this attacks in terms of Strong Primes, Key Size [4], Multi-prime RSA, Public Exponent, and private Exponent.

The selection of prime numbers in RSA directly impacts the algorithm's resilience against various attacks. By choosing prime numbers that are large, generated securely, and have suitable properties, RSA implementations can mitigate vulnerabilities and resist attacks more effectively. Additionally, considerations such as efficient modular exponentiation can indirectly impact the effectiveness of side-channel and timing attack mitigations. After that optimized the

modified RSA algorithm using Particle Swarm Optimization (PSO) optimization technique. Consider utilizing parallel processing techniques to enhance performance, especially for large data sets also introduced. Therefore, the aim that any type of information or data is encrypted with help of complex nature asymmetric key based cryptosystem but low decryption and encryption.

Next, evaluate the effectiveness of proposed MORSA, performance analyses and comprehensive simulations are conducted, also comparing its strengths as well as weaknesses with RSA and other ongoing encryption methods. The results establish that MORSA do better than standard RSA in terms of encryption speed and more endurance to cryptographic attacks where as supporting an analogous level of security.

The proposed MORSA represents a powerful result for secure advanced application in the face of ever-increasing cyber threats. By incorporating key improvements to the traditional RSA algorithm, MORSA empowers organizations and individuals to safeguard their sensitive data and maintain the confidentiality and integrity of data communications in an increasingly interconnected world.

Thus, here represent the main works of paper:

- An all-inclusive survey of the same types effort that have already been planned as well as discovered by researchers and industry professionals in this domain.
- Selected four different significant prime integer numbers instead of two prime number (standard RSA) using random prime generation algorithms and next applied optimized prime number generation algorithms to efficiently select large prime numbers required during key generation.
- Introduce dynamic modulus selection feature at proposed cryptosystem, where the modulus size is adjusted based on the size of the input data.
- The implementations of parallel computing algorithm on proposed cryptosystem to utilizes in a high-performance computing environment.

The remaining of the paper prearranged is such a way that Section 2 provides an impression of the different modified RSA algorithm. Section 3 explains our three experimental methods to implement both optimized prime number generation algorithm and dynamic modulus selection as well as the parallelized method of this proposed technique. Section 4 presents the outcomes of our research. Section 5 describes the security claims of MORSA cryptosystem followed by discussion, conclusion and future work express in Sections 6 and 7 respectively.

2. RELATED WORK

The Sergiy Gnatyuk, Yuliia Polishchuk, Elza Jintcharadze, and Maksim Iavich were co-authors of the 2018 publication. To strengthen aviation security between stationary and in-motion objects, the authors suggest a novel hybrid combination model combining AES and ElGamal encryption techniques. In the encryption approach, the author suggests encrypting the message twice. Secure and quick file sharing is what Noekeon's model is all about. According to the author, the suggested model is an attempt to circumvent the inefficient features of the Noekeon algorithm, which is the result of merging RSA and Data Encryption Standard (DES) [3]. Many still hold the RSA public-key cryptosystem in high regard for its effectiveness. When it comes to electronic encryption and digital signatures, it is the first algorithm of its kind. It uses integer factorization for security and relies on Euler's theorem, a cornerstone of contemporary cryptography, for its development. Many RSA encryption algorithms have become relatively simple to attack due to the fast growth of computer technology [4]. It is important to research new viable public-key cryptographic algorithms to complement or replace existing public-key cryptographic algorithms since traditional public-key cryptographic algorithms are continuously encountering different problems. The author followed all applicable procedures while searching for scholarly literature about conventional or modified RSA techniques and their many uses; she located around eighty-four such articles [5]. Security in the cloud [6], image encryption [7], wireless safety, and more disciplines were developed from these publications. Smart devices and Internet of Things (IoT) devices have shown good outcomes in the use of lightweight cryptographic systems that use RSA in recent years [8]. Following the lead of the literature [5], we also reviewed the works published in the previous two years [9].

Suhael et al. [10] used new encryption technology to provide effective security and produce the outer system as a waveform such that the original data would not be modified or attacked. Timing attacks are resisted by the algorithm. The RSA cryptographic algorithm was created by Haldar and Paul [11] employing three keys, however, the research still needs some protection to address the issue of information transfer from the server. Kwame et al. [12] suggested a framework that offers dual layer of security for the RSA; however, this security mechanism is not universally applicable since the methods for key production have made the proposed system more complicated. Çetin and Sınak [13] also created an improved RSA cryptosystem design. Four large prime integers are used in the procedure. The computational and spatial complexity of the innovative system is greater than that of traditional RSA due to these many primes. Multiplying two significant figures yields the general component of n. Four significant prime numbers are multiplied to get the amount of encryption and decryption. Brute Forte Attack is resistant to the newly developed algorithm. Additionally, the new method is much more effective than classical RSA. Stergio et al. [14] suggested a parallel method using a novel parallel type of data structure called a simultaneous search list of character in blocks. The RSA cryptosystem is intended to be executed at a faster pace, and its security is not addressed by the system. Using the use of several keys for connection, Abdulshaheed et al. [15] suggested an optimized type of CRT-RSA Method for safe and reliable transmission; the level of security was compared to traditional RSA. The results of the experiments showed that the suggested algorithm increases security and reduces the participation of outsiders in communication, but it has the disadvantage that it uses more resources than traditional RSA. To increase security of data in the ambiance of cloud context, the results conducted by Roussellet et al. [16] propose Quasai modified levy flying distribution for the RSA.

The encryption system developed by RSA handles safe key creation and data protection, protecting data from unauthorized access. The suggested strategy [17] uses the Cuckoo Search Algorithms (CSA) to protect and solve data integrity issues while introducing an effective RSA cryptosystem. To prevent brute force attacks and improve key encryption, CSA is used. The suggested technique increases the length of the private key while still operating more quickly than traditional RSA. In their work, Suhael et al. [18]

performed double encryption utilizing both RSA and Advanced Encryption Standard (AES), encrypting the file twice. When compared to traditional RSA, the approaches boost security since the appropriate keys are produced during algorithm execution. Kaliyamoorthy and Ramalingam [19] proposed a secured message transmission in the cloud utilizing the RSA method and an improved play fair cipher; the study's goals are to protect the key and offer security for the data transferred. The suggested system encrypts the content employing the play fair cipher in the first step, then conducts an XOR calculation on the text in the second stage, then uses the RSA algorithm to complete the process in the third stage. Though more computationally intensive, the suggested approach raises RSA's security level over traditional RSA. By combining the RSA and AES techniques with confirmation from a third party, Shree et al. [20] added an estimate and guarantees the confidentiality of encrypted data. By preventing unauthorized access to the data, the system carefully managed security and privacy concerns and ensured authentication.

The RSA cryptosystem has undergone several revisions and variations throughout the years with the purpose of addressing different issues or adding new capabilities. Here are some examples of several RSA-modified cryptosystems shown in Table 1.

Table 1. Comparatives analysis of different modifications of RSA algorithms

Reference	RSA-Modified Cryptosystems	Description
[21]	Chinese Remainder Theorem (CRT) RSA	This update uses the CRT to accelerate the decryption of an RSA key using the Chinese Remainder Theorem (CRT) RSA. To reduce the quantity of modular involution required, it anticipates pre-calculating certain values during the key creation phase and employing them throughout the decryption process.
[22]	Multi-Prime RSA	This method generates keys by engaging various prime numbers, as countered to just two large prime numbers. The effectiveness of decryption, key generation, and encryption, processes may improve.
[23]	Blinded RSA	Blinded RSA employs randomization throughout the encryption and decryption procedures to thwart side-channel attacks. Blinding, or randomization, conceals important information to prevent leakage via auxiliary channels like time or power usage
[24]	Multi-Exponent RSA	Various exponents are employed for encryption and decryption in multi-exponent RSA, enabling optimal performance in certain situations. By choosing the exponents, it may enhance the effectiveness of either decryption or encryption procedures.

There have been several modifications and variants of the RSA cryptosystem proposed over the years. While these modifications aim to address certain limitations or provide additional features, they can also introduce new vulnerabilities or drawbacks.

2.1 Drawbacks associated with different types of modified RSA cryptosystems

i. Low Public Key Efficiency [25]: RSA with Chinese Remainder Theorem (CRT), can enhance the effectiveness of decryption with help of the CRT to accelerate the modular exponentiation method. However, these schemes frequently require additional parameters and computations, which increase the size of the public key. This can lead to more storage requirements and increased transmission costs.

ii. Reduced Security Margin [26]: Certain variations to RSA, for instance RSA with small public exponents (e.g., small Fermat primes), aim to increase efficiency by using smaller exponents. Although this can accelerate decryption and encryption, it can also reduce the security margin of the algorithm. Small exponents make the encryption more exposed to attacks such as the Wiener attack, where an attacker can factorise the modulus when the exponent is too small.

iii. Vulnerability Attacks in Side-Channel [27]: Some implementations or modifications of RSA may be susceptible to side-channel attacks. Side-channel attacks exploit information leaked during the cryptographic operation, such as power consumption, timing information, or electromagnetic radiation. If proper countermeasures are not taken, sensitive data may be extracted via side-channel attacks., including the private key.

iv. Increased Key Size and Computation Complexity [28]: Some modifications of RSA aim to enhance security by using larger key sizes or more complex mathematical operations. While this can provide additional security, it also leads to increased computational requirements. Larger key sizes can impact encryption and decryption performance, requiring more computational resources and potentially slowing down the cryptographic operations.

2.2 Current research trends and research question

Kitchenham et al. [29] stated that there are three parts to the question structure: population, interventions, and outcomes. All the under-researched RSA models, or more accurately, algorithms that are improved or altered variants of the real standard RSA algorithm, make up the Population, which is the center of our study. Actions such as changed or enhanced, among others, that demonstrate the kind of therapies with RSA are the interventions in our situation according to training. The ultimate verdict on whether the improved RSA algorithms have shown any notable benefit is found in the outcomes.

RQ1. What the authors goal in improving the RSA model?

RQ2. What are the conventional RSA algorithms many revisions to date?

RQ3. How do suggested RSA schemes address present-day data or network security issues?

RQ4. Which RSA scheme domain has received greater attention?

2.3 Countermeasures and preventative measures

a. Since key: Size Since it has been shown that 512 RSA

keys are too short and unsafe for usage, many of the standards that are now in place mandate the use of 1024/2048/4096 bits for RSA keys.

b. Strong Primes it is suggested that moduli with a bit size of 1024/2048/4096 employ more than three factors [12].
c. Multi-prime RSA: To speed up the RSA private operation, there are proposals of using more than two primes to generate the modulus.
d. Public Exponent: At the very least, the private exponent need to be 300 bits in length for the usual key that is 1024 bits in length.

In summary, the selection of prime numbers in RSA directly impacts the algorithm and resilience against various attacks. By choosing prime numbers that are large, generated securely, and have suitable properties, RSA implementations can mitigate vulnerabilities and resist attacks more effectively. Additionally, considerations such as efficient modular exponentiation can indirectly impact the effectiveness of side-channel and timing attack mitigations.

The main goal of our work is the proposed MORSA algorithm, an important generation improved process that ensures that the generated keys are fairly large and resistant to modern cryptographic attacks. Additionally, the new generation of key generation technologies reduces the overhead connected to key sizes. This means that resource-limiting devices can use severe encryption. Second, this paper examines the use of optimized algorithms to generate prime numbers to efficiently select a large number of primes. This optimization reduces the performance observed in traditional RSAs due to the high complexity of computing involved in prime number production. Furthermore, the MORSA cryptosystem introduces a dynamic modulus selection approach, where the modulus is varied based on the input data size, thus improving the encryption speed while preserving the same level of security. This feature allows for adaptive encryption suitable for a diverse range of applications with varying data sizes. Next, Analyze and optimize the implementation for efficiency and speed with respected to Standard RSA algorithms, consider parallel processing techniques to enhance performance, especially for large data sets. The workflow diagram of proposed MORSA algorithm shown in Figure 1.

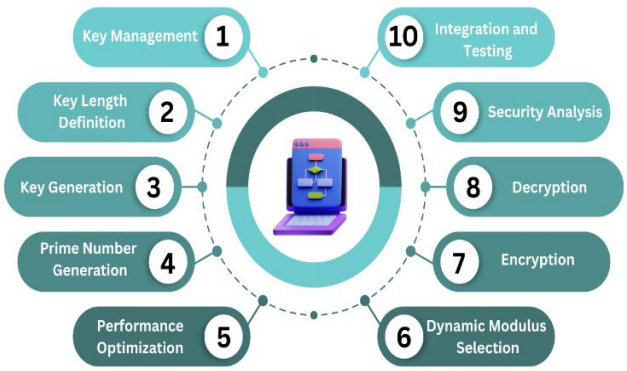

Figure 1. Workflow diagram of proposed MORSA algorithm

3. PROPOSED MORSA ALGORITHM

Step 1: Prime Selection

- Select four **large, random, distinct prime numbers**: p, q, r, and s.
- **Purpose**: Using more primes increases the modulus size (n=p*q*r*s), which increases cryptographic strength.

Step 2: Compute Modulus n

- n = p*q*r*s
- This n serves as a modulus function for both private and public keys.

Step 3: Calculate Euler's Totient Function φ(n)

- Formula: φ(n) =(p-1)(q-1)(r-1)(s-1)
- This is required for key generation (used to ensure values are coprime, and for modular inverses).

Step 4: Select Encryption Exponent function e

- Criteria: 1<e<φ(n)
- gcd (e, φ(n)) =1 (must be coprime)
- e should be **small** (to optimize encryption speed, like 3, 17, or 65537)

Step 5: Compute Modified Exponent f

- Define f =(e*a)+b, where a and b are integers.
- **Purpose**: Allows **dynamic modulus selection**, and can serve as a simple obfuscation of the public exponent.
- f still needs to be used in modular exponentiation securely.

Step 6: Compute Decryption Key d

- You must compute d such that:
  - $(d*f) \bmod \varphi(n) = 1 \rightarrow d \equiv f^{-1} \bmod \varphi(n)$ (modular inverse of f)
  - $\gcd(d,\varphi(n))=1$ (must also be coprime to φ(n))

Step 7: Public Key indicator = (f, n)

- Used to encrypt messages: $C = M^f \bmod n$

Step 8: Private Key indicator = (d, n)

- Used to decrypt messages: $M = C^d \bmod n$

Encryption & Decryption Process

- **Encryption**: $C = M^f \bmod n$
- **Decryption**: $M = C^d \bmod n$

This works correctly **if and only if** d is the modular inverse of f modulo φ(n).

This satisfies:

$(M^f)^d \equiv M^{(f*d)} \equiv M \bmod n$ since $f*d \equiv 1 \bmod \varphi(n)$

Strengths

- Larger modulus (n) makes brute-force attacks harder.
- Using f = e * a + b introduces randomness/complexity in the public key.
- Dynamic modulus scaling could potentially offer adaptability for different message sizes.

Here's a successful demonstration of the encryption scheme using randomly generated 16-bit prime numbers:

Key Values

- **Primes**:
  - p = 46307
  - q = 44983
  - r = 45751
  - s = 42787
- **Modulus**:
  - n = 4077626943713015897
- **Totient**:
  - φ(n) = 4077263824162074000

Key Generation

- Chosen e: 257
- Random a = 2, b = 3
- Computed f = e * a + b = 517
- Private exponent d = 670343181922197853 (modular inverse of f mod φ(n))

Encryption/Decryption Example

- Message M:12345
- Ciphertext C:442548085500316469
- Decrypted M:12345 (matches original)

4. METHODOLOGY TO IMPLEMENT MORSA

Here describe how to Implement the large random and optimized prime number generation algorithm during key generation using PSO optimizing technique, dynamic Modulus Selection, MORSA decryption and encryption process using Parallel of computing resources technique. Figure 2 represents four distant features proposed MORSA system.

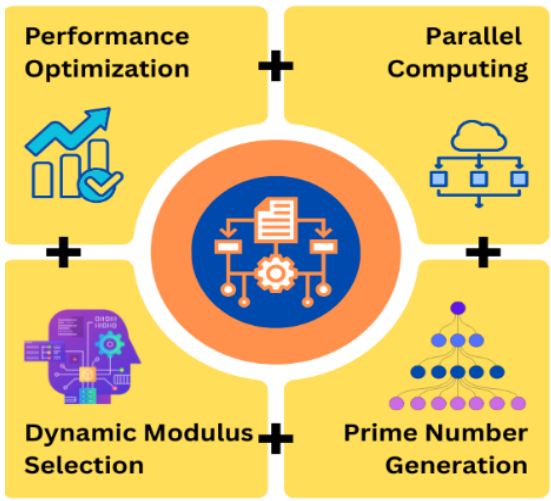

Figure 2. Four distant features of proposed MORSA system

4.1 Implement the large Randon and optimized prime number generation algorithm

4.1.1 Large random prime generation procedure

The goal is to efficiently calculate large random numbers with a specific bit size. The standard method for manually implementing a random prime generator that can generate linear values with satisfactory levels of accuracy is specified as follows:

i. Random number of the desired bit size.

ii. Make sure that the selected numbers cannot be split due to the first 100 prime numbers (these are presented). Based on the acceptable error rate, we use a certain number of iterations by Rabin Miller Primality test [30] to get the number that is probably prime.

iii. Implement the optimized prime number generation algorithm to efficiently select large prime numbers required during key generation using PSO optimising technique.

iv. Ensure that the prime number generation process is robust and capable of generating sufficiently large prime numbers for proposed method.

4.1.2 Optimization of Large prime number generation algorithm

Prime number generation is traditionally a discrete problem because prime numbers are inherently discrete values—whole numbers that are greater than 1 and divisible only by 1 and themselves. However, in the context of optimization algorithms, it can be treated as a continuous optimization problem if consider it within an optimization framework. This involves adapting continuous optimization techniques to search for prime numbers, despite their discrete nature.

Applying PSO to prime number generation is an unconventional use of the algorithm, but it can be explored as a new way to search for prime numbers in a defined range or optimize some prime-related function. Let's break down how prime number generation can be viewed as a continuous optimization problem and what adaptations would be required.

a. Understanding the Discreteness of Primes

Prime numbers are discrete because they belong to the set of integers {2,3,5,7, 11…}. So, traditionally, finding prime numbers involves checking numbers in this discrete set (integers). This is what makes prime number generation a discrete problem. PSO are often used for continuous optimization tasks where variables can take any value from a range (usually real numbers). Despite this, these algorithms can sometimes be adapted to work with discrete values by introducing methods like rounding, discretization, or relaxation.

b. Continuous Optimization for Discrete Problems

- Represent potential solutions (candidate numbers) as continuous variables.
- Adapt the algorithm to either discretize the continuous solutions or directly check for the primality of the solutions at each iteration.

c. Step-by-Step Adaptation for Continuous Optimization for Prime Number Generation

i. Discrete PSO Variant: Use a variant of PSO designed for discrete spaces, where particles represent integers rather than real-valued vectors.

ii. Fitness Function (Primality Test): The fitness function evaluates whether the number represented by a particle is prime. If the number is prime, the fitness is high; if not, the fitness is low.

```
def is_prime no.(n):
  if n <= 1:
    return False
  for i in range (2,int (n * 0.5) + 1):
  if n % i = 0:
     return False
  return True
```

iii. Fitness with Probabilistic Primality: For large numbers, you might use a probabilistic primality test (e.g., Miller-Rabin) and use the confidence as fitness:

```
def fitness(x):
  # Return a probability estimate (pseudo-code)
  return miller_rabin_confidence(x)
```

- Calculation Swarm Size (Number of Particles) for prime number search: If they consider Small Primes (i.e., <10,000) that means 20 particles so Number of Iterations will be 30. Similarly, Medium Primes (~10^6): 30-40 particles, 50-70 iterations and Large Primes (≥10^{19}): 40-60 particles, 70-100+ iterations.

With help of above condition, we get the percentage of success rate shown in Figure 3.

Velocity and Position Update: In PSO, the particles move

toward better solutions. However, after updating the particle's position (which may be continuous), you will need to round or map the particle's position to a valid integer (the next candidate number) to ensure it is still a valid candidate for primality testing.

iv. **Discretization**: Once the particle moves in the continuous space, it would be rounded or converted to an integer. The PSO algorithm itself will explore continuous space, but the solution will always be tested as a discrete integer.

v. **Optimization Process**: PSO will continue searching for prime numbers by moving particles through the continuous space and checking if their corresponding values are prime numbers. Particles with prime numbers will have better fitness values and be attracted toward areas with more prime numbers.

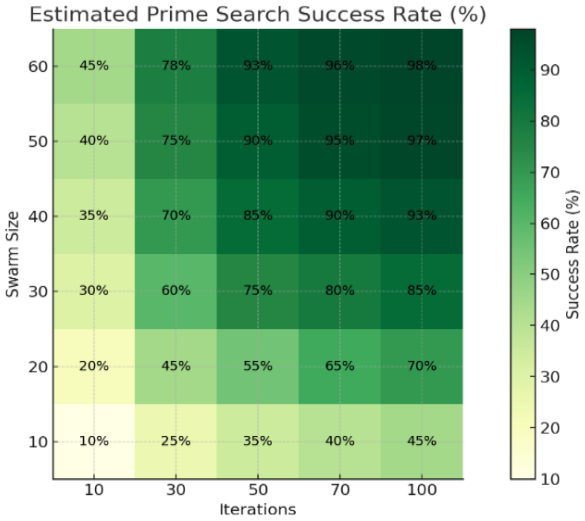

Figure 3. Estimated prime search success rate with respect to swam size

4.2 Dynamic modulus selection

Understanding Modulus Size in MORSA: The modulus size (n) in MORSA is typically 1024, 2048, 3072, or 4096 bits. The maximum size of the data that can be encrypted directly is limited to (modulus size in bytes - padding overhead). Table 2 indicates the general guideline for modulus selection. Figure 3 shows the estimated prime search success rate with respect to swam size.

Table 2. General guideline for modulus size selection

Modulus Size	Max Input Size (approx.)	Security Level
1024 bits	~117 bytes (with PKCS#1)	Weak
2048 bits	~245 bytes	Good
3072 bits	~373 bytes	Strong
4096 bits	~501 bytes	Very strong

Note: Exact input size depends on the padding scheme used (e.g., OAEP).

b. Adaptive Modulus Sizing Logic (Pseudocode)

```
def select_modulus_size(input_data_bytes):
  # Estimate required size with padding
  overhead = 42 # for OAEP padding (approx.)
  total_bytes = len(input_data_bytes) + overhead
  if total_bytes <= 117:
    return 1024 # bytes ~117
  elif total_bytes <= 245:
    return 2048 # bytes ~245
  elif total_bytes <= 373:
    return 3072
  elif total_bytes <= 501:
    return 4096
  else:
    raise ValueError ("Input data too large use parallel processing for direct MORSA encryption.")
```

d. The relation between MORSA key size (modulus size in bits) and the maximum plaintext size you can encrypt using OAEP padding is based on the following formula:

Max plaintext size (in bytes) = (modulus_size_in_bytes) - 2 * hash_len - 2

Where:

- modulus_size_in_bytes = key_size_in_bits / 8
- hash_len = length of the hash output used in OAEP (e.g., SHA-256 = 32 bytes, SHA-1 = 20 bytes)

Table 3. MORSA key size vs max OAEP plaintext size

MORSA Key Size (bits)	Modulus Size (bytes)	Max OAEP Plaintext (bytes)
1024	128	128 -2×32 -2 = 62
2048	256	256 - 2×32 - 2 = 190
3072	384	384 - 2×32 - 2 = 318
4096	512	512 - 2×32 - 2 = 446

First, determine the size of the input data. This can be measured in terms of the number of bits, bytes, or other relevant units. Next, establish criteria or rules for selecting the modulus size based on the input data size. This can be done using predefined thresholds or a mathematical formula. Then, based on the input size and the defined criteria, select an appropriate modulus size. Finally, use the selected modulus size in your cryptographic or mathematical operations. Table 3 represents relation between the MORSA key size vs. max OAEP plaintext size.

4.3 MORSA decryption and encryption process using parallel computing resources technique

It is already introduced speeding up modular exponentiation using optimized technique and dynamic modulus system to increase efficiency of MORSA cryptosystem. But still now MORSA cryptosystem faces some challenges when dealing with large input data, including Performance Issues with Large Data with Slow Encryption and Decryption: and High Computational Cost, Padding Overhead, Memory Constraints, Modulus Size Constraints, Scalability**.** Parallelization for MORSA Operations is the one impotent solution. Encrypting/decrypting multiple blocks of data in parallel.

4.3.1 Parallelization for MORSA cryptosystem operations

Parallelization can help improve the performance of the MORSA Cryptosystem, especially when working with large datasets. Because MORSA's core is based on RSA (and includes modular equation), parallelization in various regions can be used to reduce time complexity and optimize systems for large-scale operations.

a. Important areas of parallelization: Encryption and decryption of several sections: When processing large files or messages, the data must be split into smaller sections corresponding to module N. These sections can be encrypted or decrypted in parallel because each section operates independently. Here,

the data is split into small sections, each section is encrypted in parallel using a thread pool. These sections can be encrypted or decrypted in parallel because each section operates independently. Here, the data is split into small sections, each section being encrypted in parallel using a thread pool.

i. Using Python's concurrent.futures library, we can parallelize the **encryption o**f each section:

```
import concurrent.futures
def encrypt_section(section, public_key):
  “Encrypt a single section."
  return encrypt(section, public_key)
def parallel_encrypt(data, public_key, section_size, num_workers=6):
  "Encrypt data in parallel by splitting it into section."
  # Split data into sections
  Section = [data[i:i+section_size] for i in range (0, len(data), section _size)]
# Use Thread Pool Executor to parallelize the encryption of section
  with concurrent. futures. Thread Pool xecutor(max_workers=num_workers) as executor:
    encrypted_sections = list (executor.map(lambda section: encrypt_section(section, public_key), sections))
    return encrypted sections
```

ii. Similarly, for **decryption**, sections of encrypted data can be processed independently:

```
def decrypt_section (section, private_key):
  "Decrypt a single section."
  return decrypt(section, private_key)
def parallel_decrypt(encrypted_data, private_key, section _size, num_workers=6):
  "Decrypt encrypted data in parallel by splitting it into sections."
  # Split encrypted data into sections
  sections = [encrypted_data[i:i+chunk_size] for i in range(0, len(encrypted_data), section_size)]
  # Use ThreadPoolExecutor to parallelize the decryption of sections
  with concurrent.futures. Thread Pool Executor(max_workers=num_workers) as executor:
    decrypted_sections = list(executor.map(lambda section: decrypt_section(section, private_key), sections))
    return decrypted_sections
```

b. **Modular Exponentiation** (MORSA Operations): The modular exponentiation operation $M^f \bmod n$ (for encryption) and $C^d \bmod n$ (for decryption) is computationally expensive. Although MORSA is inherently sequential for a single message, it can be parallelized when dealing with multiple messages or blocks. This is especially useful when processing multiple chunks of encrypted data in parallel. In the example above, modular exponentiation for each chunk is parallelized using a thread pool, which speeds up the overall process.

```
def modular_exponentiation(modulus, base, exponent,):
"Modular exponentiation."
Returnpow(modulus,base,exponent,)
def parallel_modular_exponentiation(sections, exponent, modulus, num_workers=6):
"Perform modular exponentiation on each section in parallel." with concurrent futures.
Thread Pool Executor(max_workers=num_workers) as executor:
  results = list(executor.map(lambda section : modular_exponentiation (section, exponent, modulus), sections))
  return results
```

c. **Key Generation:** Generating large prime numbers for the keys and the modulus n can be parallelized. While probabilistic primality tests (like the Miller-Rabin test) can be expensive, they can be run on different cores for each candidate prime. Additionally, the process of finding multiple primes (e.g., p,q,r,s) for a multi-prime system like MORSA can be parallelized.

```
import concurrent.futures
from sympy import is prime
def generate_prime_candidate(bits=1024):
  "Generate a random candidate for prime number."
  return random.getrandbits(bits)
def check_and_generate_prime(bits=1024):
  "Generate a prime number."
  candidate = generate_prime_candidate(bits)
  while not isprime(candidate):
    candidate = generate_prime_candidate(bits)
  return candidate
def parallel_generate_primes(num_primes=4, bits=1024, num_workers=6):
  "Generate multiple primes in parallel."
  with concurrent.futures.ThreadPoolExecutor(max_workers=num_ workers) as executor:
    primes = list(executor.map(lambda_:section _and_generate_prime(bits), range(num_primes)))
  return primes
```

d. **Key Splitting and Combining:** When using multiple primes (as in the MORSA system), operations like splitting the private key into smaller parts (modular inverses for each prime) and combining results can benefit from parallelism. In this case, modular inverses for each prime can be computed in parallel, reducing the time spent on this operation.

```
def mod_inverse_parallel(a, moduli, num_workers=6):
  "Compute modular inverse in parallel for each prime factor."
  def mod_inv_for_prime(p):
    return mod_inverse(a, p)
  with concurrent. futures. Thread Pool Executor(max_workers=num_workers) as executor:
    inverses = list(executor.map(mod_inv_for_prime, moduli))
    return inverses
```

e. **Parallelizing Padding and Unpadding:** Padding and unpadding of the message to ensure it fits into the modulus size can be done in parallel for multiple chunks. This is particularly useful when dealing with larger files or messages.

```
def parallel_pkcs1_pad(sections, n_size, num_workers=6):
  "Apply PKCS#1 padding in parallel to each chunk."
  with concurrent. futures. Thread Pool Executor(max_workers=num_workers) as executor:
    padded_sections = list(executor.map(lambda section: pkcs1_pad(section,n_size),sections))
  return padded_sections
def parallel_pkcs1_unpad(sections, n_size, num_workers=6):
```

"Unpad PKCS#1 padding in parallel."

with concurrent. futures. Thread Pool Executor(max_workers=num_workers) as executor:

unpadded_sections = list(executor.map(lambda section: pkcs1_unpad(secton, n_size), sections))

return unpadded_sections

In our proposed model, multiple threads are parallelly executed equ-length sections. By parallelizing key parts of the MORSA cryptosystem, including the encryption/decryption of sections, modular exponentiation, key generation, and padding, we can improve the system's efficiency and scalability, especially when working with large data sets.

5. RESULT ANALYSIS OF MORSA CRYPTOSYSTEM

Implementation of proposed work have required following system: CPU- Intel Core I7-2670QM at 2.20GHz frequency, Memory 16.0 GB, GPU: NVIDIA GeForce GT630M consists of 96 cores, and Windows Home Premium.

5.1 Performance analysis of proposed MORSA algorithm

Compare the amount of time required by various algorithms shown in Table 3, we have demonstrated the outcomes of the various methods like Key Generation Time, Decryption Time, and Encryption Time, described here. We applied 20-bit modules and 20-bit messages using python libraries. Table 4 shows the comparison of algorithms using 20-bits modules size and 20-bits message size version different version RSA and proposed MORSA algorithm.

The effectiveness of any specific encryption and decryption approach is dependent upon the speed at which a cryptographic algorithm is implemented, and the period of execution identifies the algorithm's speed or execution speed.

As shown in Figures 4 and 5, the required time for the proposed MORSA and SRSA algorithm is applied for the news size of bytes. When you view the message, a graphical comparison of the proposed algorithm MORSA and the encryption time of the traditional SRSA size shows the same bytes.

Next, test the proposed MORSA with different input bit sizes. Table 4 shows the performance of the original RSA (SRSA) algorithm by Rivest, Shamir, and Adleman [1]. Table 4 also displays the MORSA scheme's performance regarding of key generation, encryption time, and decryption. By comparing the tables, it is possible to determine that MORSA requires more time for key creation than RSA. The fact that the time to break the system is long due to the additional complexity makes Modified and optimized RSA's (MORSA) longer key generation period advantageous.

Next, test the proposed MORSA with different input bit sizes. Table 5 shows the performance of the original RSA (SRSA) algorithm by Rivest, Shamir, and Adleman [1]. Table 5 also displays the MORSA scheme's performance regarding of key generation, encryption time, and decryption. By comparing the tables, it is possible to determine that MORSA requires more time for key creation than RSA. The fact that the time to break the system is long due to the additional complexity makes Modified and optimized RSA's (MORSA) longer key generation period advantageous.

Table 5 shows the encryption and decryption result compares of the proposed algorithms MORSA and SRSA with respect to large message size. It is evident from the findings in Table 5 that the computational complexity of both encryption and decryption on MORSA has a more advanced cryptosystem than SRSA, indicating that it will be more complicated, and the attackers need considerably more time to breach easily than that of the SRSA. Table 6 presents the encryption and decryption times (in seconds) for MORSA and RSA algorithms across different input sizes (in KB).

5.2 Analysis of computational complexity and big-O-notation algorithm of MORSA

a. **Computational Complexity:** Let's examine each stage of the modified key creation, message encryption, and text decryption procedure in terms of its temporal complexity. The big O notation describes the computational complexity of an algorithm in terms of its input size — in this case, the bit-length of the primes used (denoted k). Here's an analysis of the MORSA algorithm and define Parameters k = bit-length of each prime number (e.g., 1024 bits in real-world use). The operations include prime generation, modular exponentiation, and modular inversion.

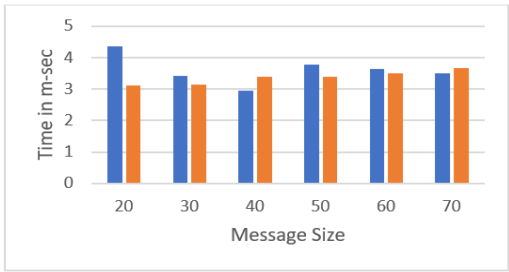

Figure 4. Plotting the proposed MORSA and SRSA algorithms' encryption times versus message sizes in bytes

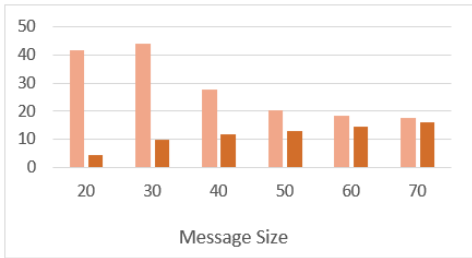

Figure 5. Plotting the proposed MORSA and SRSA algorithms' decryption times versus message sizes in bytes

Table 4. Analysis of security and efficiency of different

S. No	Algorithm	Time of Key Generation (µsec)	Time of Encryption (µsec)	Time of Decryption (µsec)
1	SRSA [31]	989	989	1998
2	Prime numbers RSA [32]	988	1988	13992
4	Diffie-key-RSA [33]	999	1000	999
5	IRSA [34]	905	985	2010
6	MREA [35]	1598	3174	3896
7	**MORSA**	**1875**	**4057**	**14976**

Table 5. Key generation, encryption and decryption time (ms) of SRSA and MORSA with different keys respectively

Length of p, q, s, r (in bits)	Analysing Tine for MORSA		
	Key Generation Time(ms)	Encryption Time (ms)	Decryption Time(ms)
100	231	0.27	1.36
128	241.23	0.54	2.34
256	248.7	1.23	12.62
512	365.6	2.7	78.65
1024	1165.7	6.89	376.2
2048	6089.6	19.9	2169.6
4096	15,514	55.67	17,789.7
Length of p, q (in bits)	**Analysing Tine for SRSA**		
	Key Generation Time(ms)	**Encryption Time (ms)**	**Decryption Time(ms)**
100	76.63	0.16	0.25
128	90.46	0.17	0.28
256	94.96	0.35	0.96
512	177.47	0.56	5.2
1024	570.9	1.69	26.18
2048	4201.47	3.32	130.83
4096	54368.00	11.17	1116.24

Table 6. Encryption and decryption tine (sec) of MORSA and RSA with distinct sizes (KB)

Size of Message (KB)	Encryption Time (sec) MORSA	Encryption Time SRSA (sec)	Decryption Time (sec) MORSA	Decryption Time (sec) SRSA
10	**2**	0	**0.1**	0
1000	**4**	2	**2**	2
5000	**12**	8	**7**	7
10000	**20**	16	**16**	16
20000	**35**	28	**27**	25
50000	**80**	71	**66**	62
100000	**165**	146	**160**	152

Table 7. Represented step-by-step calculation of complexity measurement of MORSA

Step	Operation	Complexity	Reason
1	Generate 4 distinct primes (p, q, r, s)	O (k log k) (each) × 4 → O (k log k) total	Primality testing (e.g., Miller-Rabin) and random generation
2	Multiply 4 primes for n = p*q*r*s	$O(k^2)$	Big integers, each of k bits
3	Compute φ(n) = (p−1)(q−1)(r−1)(s−1)	$O(k^2)$	3 multiplications of k-bit numbers
4	Choose small e such that gcd(e, φ(n)) = 1	O(k)	Using Euclidean algorithm
5	Compute f = e*a + b, check gcd (f, φ(n)) = 1	O(k)	Simple arithmetic and Euclidean check
6	Compute $d = f^{-1} \bmod \varphi(n)$	$O(k^2)$	Using Extended Euclidean Algorithm
7	Encrypt: $C = M^f \bmod n$	$O(k^3)$	Modular exponentiation with exponent $\sim\log_2(f) \leq k$
8	Decrypt: $M = C^d \bmod n$	$O(k^3)$	Modular exponentiation with large d (up to k bits)

Table 8. Complexity comparison: SRSA vs MORSA

Operation	SRSA	MORSA	Notes
Key Generation	**$O(k^2)$**	**$O(k^2)$**	Extra step to compute f, but still polynomial
Prime generation	$O(k^2)$	$O(k^2)$	Both use 2-4 large primes
Modular inverse	$O(k^2)$	$O(k^2)$	Finding $d=f^{-1}\bmod \phi(n)$ dominates
Select e	O(1)	O(1)	Common e values like 65537
Compute f	-	O(log k)	Lightweight; adds negligible cost
Encryption	**$O(k^2)$**	**$O(k^3)$**	Fast in SRSA (small e), slow in modified (large f)
Decryption	**$O(k^3)$**	**$O(k^3)$**	Always slow due to large d

Table 9. Analysis of MORSA and classical RSA algorithm as well as different variant of RSA cryptosystem with different parameter

Encryption Technique	Complexity Encryption	Complexity Decryption	Avalanche Effect
MORSA	**$O(n^3)$**	**$O(n^3)$**	**57.52%**
SRSA	$O(n^2)$	$O(n^3)$	50.20%
MREA [35]	O(n)	$O(n^3)$	46.10%
RBMRSA	$O(n^3)$	$O(n^3)$	35.13%

a. Big-O-notation algorithm measurement of MORSA

Total Complexity Summary as per Table 7.

i. Key Generation (Steps 1-6): O (k log k + k^2) ≈ O(k^2)
ii. Encryption: O(k^3) — due to modular exponentiation with small exponent f.
iii. Decryption: O(k^3) modular exponentiation with large exponent

For computing the modular inverse have the highest time complexity. It's crucial to remember that although encryption and decryption may be carried out several times for various communications, key creation is normally carried out only once. As a result, the key generation process would dominate the total time complexity of RSA. The modified RSA technique has an O (n^2*(log n)^3) time complexity for key creation, where n is the bit length of the key, and an O(k^3) time complexity for encryption and decryption, where k is the bit length of the modulus. Table 8 represents complexity Comparison of SRSA vs MORSA, and it is observed that the proposed method is better.

b. Avalanche effect:

Avalanche effect refers to the degree of variance in ciphertext that may be caused by small changes or variations in plaintext. To be effective, an encryption algorithm or cipher must provide totally new results with only a little adjustment to the input. An algorithm's security is directly correlated to the number of changes it can withstand in the ciphertext; in other words, a bigger avalanche effect indicates a more secure algorithm. To ensure that an attacker would have a hard time performing statistical evaluation on the ciphertext, we flip only one bit in the avalanche effect (%) to test how sensitive the proposed algorithm is. It looks at how much change the ciphertext would be if the plaintexts were changed just a little bit. The result of changing a single bit in MORSA's 1 K.B. plaintext resulted in a 57.52% shift in the ciphertext, while RSA's shift was assessed to be 50.2% shown in Table 9.

5.3 MORSA decryption and encryption process using parallel of computing resources technique

To implement this technique, write a simple Python program. using Python XMLRPC. Additionally, it can change the Python Encryption Library (Crypto) to support the MORSA algorithm. Every computer utilized had this Python program running as a daemon. It is evaluated with a variety of file sizes.

The working set include {10,100,200,300,400,500} byte comparison of encryption for parallel and sequential for large prime number.

Table 10 shows the relationship between the amount of data and the execution time (in seconds) for the MORSA algorithm. The first column shows the character length of the byte input to the algorithm. The second column shows the parallel encryption time, and the third column shows the sequential encryption time to process the data input. Table 11 presents that the decoding time is used to run MORSA. Runtime is calculated in seconds. The speed-up coefficient for parallel calculations running on P processors is derived as the ratio: These equations are referenced when calculating the speed and efficiency of each parallel execution of the RSA algorithm [8]. Table 11 presents total time for sequential and parallel character lengths of the sequential and parallel acceleration sequential bits of time in parallel acceleration. In this way, we can compare how effective a parallelized approach is for sequential approaches. In the above table the length of the character at the byte entry of the algorithm shows the total time of parallelism in the second column. The third column shows the sequential speed calculations for processes of different sizes of data input and the total time in the fourth column. We created a parallel MORSA encryption algorithm tool using OpenMP libraries and performed the experiments on high performance computing.

Table 10. Comparative data in the execution time (in seconds) for the MORSA algorithm with parallel and sequential computing

Length of Character in Byte	Encryption Time for Parallel Computing	Encryption Time for Sequential Computing	Decryption Time for Parallel Computing	Decryption Time for Sequential Computing
10	0.002808	0.000188	0.037315	0.094571
100	0.01378	0.001901	0.274317	0.943160
200	0.001977	0.003675	0.536104	1.873151
300	0.003337	0.005673	0.811431	2.821047
400	0.003881	0.007447	1.092126	3.751115
500	0.004161	0.009513	1.451110	4.721110

Table 11. The decoding time is used to run MORSA and speed-up coefficient for parallel calculations

Length of Character in Byte	Total Time Taken in Sequential	Total Time Taken in Parallel	Speed-Up
10	0.094719	0.040313	2.348333163
100	0.941401	0.027555	3.429652221
200	1.877217	0.548292	3.442997159
300	2.841514	0.811331	3.491313251
400	3.768575	1.051911	3.446132132
500	4.710109	1.451516	3.636755213

6. SECURITY CLAIMS OF MORSA CRYPTOSYSTEM

MORSA uses four big primes instead of two, making the modulus n = p * q * r * s difficult to calculate.

Below explains how and why does this withstand to security claims likes brute-force attacks.

1. Large key space makes brute force impractical: MORSA is the product of 4 big primes (512-1024 bits each). If each prime is 512 bits, n is 2048 bits and if each prime is 1024 bits, n is 4096 bits. Brute-force factorization is exponentially more costly than with 2 primes due to the exponential number of prime combinations. With modulus size, brute force factoring time complexity becomes super-polynomials. Combinations grow

exponentially with additional primes.

2. Increased Primes in Factorization: Although additional primes may make factorization simpler, this is not true. Classical algorithms, such as trial division and Pollard's rho, remain unsuccessful. Increasing the number of bits in n slows even the quickest factoring process, number field sieve (NFS). MORSA's n = p * q * r * s is much greater, therefore NFS chokes.
3. Factoring Hardness Security: RSA can be broken by factoring n = p * q in traditional RSA but MORSA factor n = p * q * r * s. This demands more resources and time to break multiple factors. Here key length (bits) is more important than prime number in repelling brute force. The use of numerous big primes provides increased security.
4. No Use for Brute Force Key Guessing: Some call brute force "guessing the private key d": d is usually hundreds of digits long and independent calculated. Factoring n is necessary to calculate φ(n) and confirm assumed d. Unknown totient makes estimating d impossible.
5. Padded Randomized Encryption: With OAEP or another padding method in MORSA is given the same plaintext with distinct ciphertexts each time, preventing dictionary and chosen-plaintext attacks, which are brute-force tactics.

The answer of the proposed research questions, which are introduced at Section 2 are addressed. In RQ 1: MORSA aims to improve RSA's key generation via PSO, dynamic modulus, parallel computation via Tables 3, 4 and 8; RQ 2: Table 1 represents different modification of conventional RSA cryptosystem; RQ 3: Section 6 shows how the present-day data or network security issues can be minimized with proposed MORSA. RQ 4: RSA's key generation via PSO, dynamic modulus is giving the attention. This was attended in our proposed model.

7. CONCLUSION

The factoring of the huge integer is essential to the security of the RSA algorithm. Instead of using two prime numbers, this study uses four separate prime numbers, which has the effect of increasing the amount of time it takes to locate a large prime number. Since the keys for MORSA is dependent on a big factor value "n" the amount of time required for key generation is increased. When the time it takes to generate a key is increased, the amount of time it takes to break the system also grows, which gives the system more strength. In comparison to the RSA method, the technique for double encryption and decryption that is used by MORSA is straightforward, and as a result, it does not cause any additional burden on the system. More time is required for both encryption and decryption than is required by the RSA technique. To evaluate the effectiveness of the algorithm, the amount of time required for a brute force assault is taken into consideration. One of the limitations of this suggested schema is that it will not function correctly unless "n" different prime integers are taken into consideration simultaneously. It is possible that in the future, it will be beneficial to work on improving the security of the RSA algorithm by including more elements into the encryption and decryption process.

MORSA and McEliz combination in the public key cryptography using hard mathematics [8] (McEliece instead of factoring) provides dual protection. The new MORSA formation gives classical security (hard to break with current classical methods) as well as quantum resistance based on the hardness of decoding random linear codes.

ACKNOWLEDGEMENTS

We would like to show our gratitude to Dr. Rituparna Bhattacharya, HoD, CSE Department, Techno India University West Bengal, India, for sharing her pearls of wisdom with us during this research work.

REFERENCES

[1] Rivest, R.L., Shamir, A., Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2): 120-126. https://doi.org/10.1145/359340.359342

[2] Haldar, B., Paul, P. (2025). Advancing Mobile banking security using modified RSA approach for data transformation enhancement. Shaping Cutting-Edge Technologies and Applications for Digital Banking and Financial Services, 390.

[3] Wang, Z., Dong, H., Chi, Y., Zhang, J., Yang, T., Liu, Q. (2021). Research and implementation of hybrid encryption system based on SM2 and SM4 algorithm. In Proceedings of the 9th International Conference on Computer Engineering and Networks, 1143: 695-702. https://doi.org/10.1007/978-981-15-3753-0_68

[4] Falowo, O.M., Misra, S., Falayi, C.F., Abayomi-Alli, O., Sengul, G. (2022). An improved random bit-stuffing technique with a modified RSA algorithm for resisting attacks in information security (RBMRSA). Egyptian Informatics Journal, 23(2): 291-301. https://doi.org/10.1016/j.eij.2022.02.001

[5] Zheng, M. (2023). Generalized implicit-key attacks on RSA. Journal of Information Security and Applications, 77: 103562. https://doi.org/10.1016/j.jisa.2023.103562

[6] Wang, X., Xiang, Z., Zhang, S., Chen, S., Zeng, X. (2025). Quantum chosen-ciphertext attacks based on Simon's algorithm against unified structures. In Cryptographers' Track at the RSA Conference, Cham: Springer Nature, Switzerland, pp. 99-122. https://doi.org/10.1007/978-3-031-88661-4_5

[7] Nitaj, A., Susilo, W., Tonien, J. (2023). A new attack on some RSA variants. Theoretical Computer Science, 960: 113898. https://doi.org/10.1016/j.tcs.2023.113898

[8] Imam, R., Areeb, Q.M., Alturki, A., Anwer, F. (2021). Systematic and critical review of RSA based public key cryptographic schemes: Past and present status. IEEE Access, 9: 155949-155976. https://doi.org/10.1109/ACCESS.2021.3129224

[9] Moghaddam, F.F., Alrashdan, M.T., Karimi, O. (2023). A hybrid encryption algorithm based on RSA small-e and efficient-RSA for cloud computing environments. Journal of Advanced Computer Networks, 1(3): 238-241.

[10] Suhael, S.M., Ahmed, Z.A., Hussain, A.J. (2025, February). A review on hybrid methods using Playfair and RSA techniques. AIP Conference Proceedings, 3169(1): 030002. https://doi.org/10.1063/5.0256836

[11] Munira, M.S.K. (2025). Digital transformation in banking: A systematic review of trends, technologies, and challenges. SSRN Electronic Journal.

https://doi.org/10.2139/ssrn.5161354
[12] Kwame, A.A., Owa, K., Tawfik, A.H. (2025). For RSA encryption scheme. In Security and Management and Wireless Networks: Proceedings of SAM 2024 and ICWN 2024.
[13] Çetin, F., Sınak, A. (2025). Probabilistic primality tests and RSA algorithm. Akdeniz University Journal of Science and Engineering, 1(1): 8-18.
[14] Stergio, C., Kim, K.E., Gupta, B.G. (2018). Secure an integration of IoT and cloud computing. Future Generation Computer Systems, 78(6): 964-975. https://doi.org/10.1016/j.future.2016.11.031
[15] Abdulshaheed, H.R., Binti, S.A., Sadiq, I.I. (2018). Proposed smart solution based on cloud computing and wireless sensing. International Journal of Pure and Applied Mathematics, 119(18): 427-449.
[16] Roussellet, M., Tteglia, Y., Vigilant, D. (2025). Protected RSA implementations. Embedded Cryptography, 2: 201.
[17] Mohaisen, H.N., Mohammed, M.Q., Nahi, M.H. (2025). Hiding secret data in color video applying modify RSA for cryptography with randomly select frame and pixel to steganography. Journal of Natural and Applied Sciences URAL, 70.
[18] Suhael, S.M., Ahmed, Z.A., Hussain, A.J. (2025). A review on hybrid methods using Playfair and RSA techniques. AIP Conference Proceedings, 3169(1): 030002. https://doi.org/10.1063/5.0256836
[19] Kaliyamoorthy, P., Ramalingam, A.C. (2022). QMLFD based RSA cryptosystem for enhancing data security in the public cloud system. Wireless Personal Communications, 122: 752-782. https://doi.org/10.1007/s11277-021-08924-z
[20] Shree, R., Chelvan, C., Rajesh, M. (2019). An efficient RSA cryptosystem by applying cuckoo search optimization techniques. Concurrency and Computation: Practice and Experience, 31(12): e4845. https://doi.org/10.1002/cpe.4845
[21] Jaspin, K., Selva, S., Sahana, S., Thamnas, G. (2021). Efficient and secured file transfer in cloud through double encryption using AES and RSA algorithm. International Journal of Emerging Smart Computing and Informatics, Pune, India, pp. 791-796. https://doi.org/10.1109/ESCI50559.2021.9397005
[22] Hemanth, P., Raj, N., Yadva, N. (2017). A secured message transfer using RSA algorithm an improved Playfair cipher in cloud computing. International Conference on Convergence in Technology (I2CT), Mumbai, India, pp. 931-936. https://doi.org/10.1109/I2CT.2017.8226265
[23] Almanaun, S., Mahmood, M., Amin, M. (2021). Ensuring the security of encrypted information with hybrid of AES and RSA algorithm with the third-party confirmation. In 5th International Conference on Intelligent Computing and Control Systems, Madurai, India, pp. 337-343. https://doi.org/10.1109/ICICCS51141.2021.9432174
[24] Wu, C.H., Hong, J.H., Wu, C.W. (2021). RSA cryptosystem design based on the Chinese remainder theorem. In Proceedings of the ASP-DAC 2021 Asia and South Pacific Design Automation Conference, Yokohama, Japan, pp. 391-395. https://doi.org/10.1145/370155.37041
[25] Kamardan, M. G., Aminudin, N., Che-Him, N., Sufahani, S., Khalid, K., Roslan, R. (2018). Modified multi prime RSA cryptosystem. Journal of Physics: Conference Series, 995 (1): 012030. https://doi.org/10.1088/1742-6596/995/1/012030
[26] Ueno, R., Homma, N. (2023). How secure is exponent-blinded RSA-CRT with sliding window exponentiation? IACR Transactions on Cryptographic Hardware and Embedded Systems, 2023(2): 241-269. https://doi.org/10.46586/tches.v2023.i2.241-269
[27] Kumari, J.J.J., Thangam, S. (2025). ERSA enhanced RSA: Advanced security to overcome cyber-vulnerability. In N.K. Chaubey, N. Chaubey (Eds.), Advancing Cyber Security Through Quantum Cryptography, IGI Global, pp. 413-440. https://doi.org/10.4018/979-8-3693-5961-7.ch015
[28] Venkatalakshmi, K., Gayathri, P., Likhitha, T., Shinde, S., Kumar, M.O.V. (2022). Design of Montgomery multiplier - RSA algorithm. Journal of Physics: Conference Series, 2325(1): 012022. https://doi.org/10.1088/1742-6596/2325/1/012022
[29] Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., Linkman, S. (2009). Systematic literature reviews in software engineering – A systematic literature review. Information and Software Technology, 51(1): 7-15. https://doi.org/10.1016/j.infsof.2008.09.009
[30] Liskov, M. (2005). Miller–rabin probabilistic primality test. In Encyclopedia of Cryptography and Security. Springer, Boston, MA. https://doi.org/10.1007/0-387-23483-7_253
[31] Ambedkar, B.R., Bedi, S.S. (2011). A new factorization method to factorize RSA public key encryption. International Journal of Computer Science Issues, 8(6).
[32] Ivy, B.P.U., Mandiwa, P., Kumar, M. (2012). A modified RSA cryptosystem based on 'n' prime number. International Journal of Engineering and Computer Science, 1(2): 63-66.
[33] Gupta, S., Sharma, J. (2012). A hybrid encryption algorithm based on RSA and Diffie- Hellma. IEEE International Conference on Computational Intelligence and Computing Research, Coimbatore, India, pp. 1-4. https://doi.org/10.1109/ICCIC.2012.6510190
[34] Bhattacharjee, A., Khaskel, C., Basu, D., Vincent, P.M. (2016). Hybrid security approach by combining diffie hellman and RSA algorithm. International Journal of Pharmacy and Technology Dec, 8(4): 26560-26567.
[35] Dhakar, R.S., Gupta, A.K., Sharma, P. (2012). Modified RSA encryption algorithm (MREA). In 2012 2nd International Conference on Advanced Computing and Communication Technologies, Rohtak, India, pp. 426-429. https://doi.org/10.1109/ACCT.2012.74