Cryptography Project

RSA Implementation, Analysis and Application

Maha Hanif
Sami Agourram
Ayman Youss

January 7, 2025

Contents

1 Key Management System

The *key management* logic is encapsulated in a specialized class (e.g., `RSAKeyManager`), which handles:

- Generating a new RSA keypair.
- Persisting the keypair (public and private keys) to files.
- Storing *metadata*, including creation time, expiration time, and key size.
- Checking whether a key has expired and rotating keys if necessary.

1.1 Methods Used

1. **RSA Key Generation:** A new RSA keypair is generated by calling a method from the core RSA class. The key size can be configured (e.g., 2048 bits for security or smaller for demonstrations).
2. **Metadata Management:** The metadata file (`JSON` format) stores:
   - `creation_time`: UTC timestamp of when the key was generated.
   - `expiration_time`: UTC timestamp, computed by adding a specified number of days to the creation time.
   - `key_size`: The integer size in bits (e.g., 2048).
3. **Expiration Check:** The system compares the current UTC time with the stored `expiration_time`. If the key is past its validity period, it is considered invalid and triggers a rotation.
4. **Key Rotation:** Upon expiration, the old keys are replaced with a fresh keypair. The metadata file is also updated to reflect the new key attributes.

1.2 Correctness and Robustness

- **Correct Key Storage:** Correctness is ensured by verifying the presence of both public and private key files before usage. If keys do not exist, the system prompts a regeneration.
- **Expiration Safety:** Storing the expiration timestamp and enforcing rotation reduces the likelihood of using compromised or aged keys. Once an expired key is detected, the application reliably replaces it.
- **Metadata Integrity:** The system expects the metadata file to remain uncorrupted. If it is missing or unreadable, the code defaults to a safe response (considering the key invalid or expired) to avoid potential confusion around unknown or stale keys.

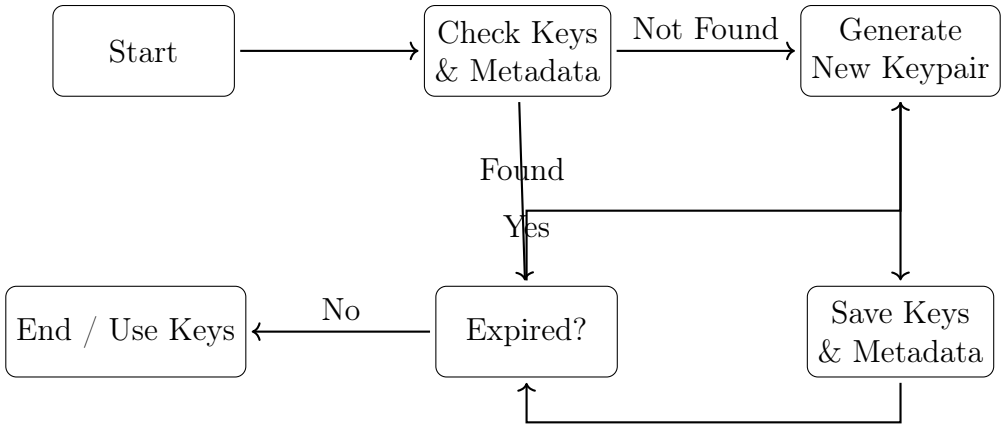

Figure 1: Key Management Workflow

1.3 Conceptual Diagram

In Figure 1:

1. The system begins by checking whether key files and metadata exist.
2. If missing, new keys are created and saved.
3. If present, the system evaluates whether the keys have expired.
4. If expired, the keys are regenerated and saved (rotated).
5. Otherwise, the keys are ready for normal usage.

2 Implementation Details

This section presents a detailed implementation of the RSA cryptosystem in Python, featuring robust key generation, secure padding mechanisms, and timing attack mitigations.

2.1 Core RSA Class Implementation

The implementation centers around a primary `RSA` class that encapsulates all cryptographic operations. The class provides a comprehensive interface for key generation, message padding, encryption, and decryption.

2.1.1 Key Generation

The key generation process implements several crucial security features:

- **Secure Prime Generation:** Utilizes the Miller-Rabin primality test with 128 rounds for high confidence in prime number generation:

```
def is_prime(self, n: int, k: int = 128) -> bool:
    if n == 2 or n == 3:
        return True
    if n < 2 or n % 2 == 0:
        return False

    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2
```

- **Cryptographically Secure Random Number Generation:** Employs Python's `secrets` module instead of the standard `random` module for generating prime candidates:

```
def generate_prime(self, bits: int) -> int:
    while True:
        n = secrets.randbits(bits)
        n |= (1 << bits - 1) | 1
        if self.is_prime(n, 128):
            return n
```

2.1.2 Message Padding

The implementation includes PKCS#1 v1.5 style padding for enhanced security:

- **Padding Implementation:** Follows the format: `00 || 02 || PS || 00 || M`
- **Random Padding:** Generates non-zero random bytes for the padding string (PS)
- **Length Validation:** Ensures messages don't exceed the maximum length based on key size

The padding method includes crucial security checks:

```
def pad_message(self, message: Union[str, bytes]) -> int:
    if isinstance(message, str):
        message = message.encode('utf-8')

    max_message_length = self.key_size // 8 - 11
    if len(message) > max_message_length:
        raise ValueError(f"Message too long")

    padding_length = self.key_size // 8 - len(message) - 3
```

```
padding = b''
while len(padding) < padding_length:
    byte = secrets.token_bytes(1)
    if byte != b'\x00':
        padding += byte
```

2.1.3 Encryption and Decryption

The core cryptographic operations implement several security measures:

- **Timing Attack Mitigation:** Both encryption and decryption operations include constant-time padding:

```
start_time = time.time()
cipher = pow(padded, e, n)
time.sleep(0.001 - ((time.time() - start_time) % 0.001))
```

- **Input Validation:** Checks for key availability and appropriate message sizes:

```
if not self.public_key:
    raise ValueError("No public key available")

if cipher >= n:
    raise ValueError("Ciphertext too large")
```

2.2 Performance Analysis

The implementation includes performance monitoring capabilities:

- **Timing Measurements:** Records encryption and decryption times:

```
start_time = datetime.now()
encrypted = rsa.encrypt(message)
encryption_time = (datetime.now() -
                   start_time).total_seconds()
```

- **Key Generation Monitoring:** Tracks the time taken for key generation and primality testing

2.3 Security Considerations

The implementation addresses several security aspects:

- **Strong Prime Generation:** Uses cryptographically secure random number generation
- **Proper Padding:** Implements PKCS#1 v1.5 padding scheme
- **Timing Attack Protection:** Implements constant-time operations
- **Input Validation:** Includes comprehensive error checking

2.4 Future Enhancements

Potential improvements to the current implementation could include:

- **OAEP Padding:** Upgrading to more modern PKCS#1 v2.0 OAEP padding
- **Additional Side-Channel Protections:** Implementing blinding techniques
- **Performance Optimizations:** Using the Chinese Remainder Theorem for decryption
- **Memory Security:** Implementing secure memory wiping for sensitive values

3 Security Analysis

3.1 Naive Factoring of Small RSA Moduli

3.1.1 Principle

One component of the security analysis is to demonstrate the ease of factoring *small* RSA moduli. An RSA modulus n is simply the product of two primes p and q. If the key size is extremely small (e.g., 32 or 48 bits total), a brute-force or naive approach that attempts dividing n by candidate factors up to $\sqrt{n}$ will succeed quickly.

3.1.2 Method

- **Enumerate possible factors:** Start from 3 and increment in steps of 2 (skipping even numbers), testing for divisibility of n.
- **Stop at $\sqrt{n}$:** If p is a prime factor, it cannot exceed $\sqrt{n}$. Thus, searching beyond $\sqrt{n}$ is unnecessary.

3.1.3 Correctness

The method is grounded in the fundamental property of integer factorization:

$$\text{If } n = p \cdot q, \text{ then } p \leq \sqrt{n}.$$

Hence, it *must* find the factors if n is small enough. For large n (e.g., 2048 bits), this approach is computationally infeasible. The demonstration proves *why* cryptographic best practices require large key sizes.

3.2 Timing Attack Demonstration

3.2.1 Principle

A *timing attack* attempts to deduce private information by measuring how long an operation (e.g., RSA decryption) takes under various conditions. If an implementation is not *constant-time*, different parts of the algorithm might take slightly different amounts of time, revealing subtle clues about the key.

3.2.2 Method

- **Encrypt a set of sample messages:** The system creates ciphertexts for different plaintexts.
- **Measure decryption time repeatedly:** Each ciphertext is decrypted multiple times, and the elapsed time is recorded.
- **Analyze differences:** By comparing the average and standard deviation of decryption times, one can see if certain ciphertexts yield longer or shorter durations, which might indicate how the algorithm branches internally.

4 Application Overview

The RSA Encryption Application is a web-based tool designed to provide a secure and user-friendly platform for performing RSA-based cryptographic operations. It enables users to generate RSA key pairs, encrypt and decrypt messages or files, and manage cryptographic keys efficiently. The application is built using Flask, integrating both Python libraries and custom modules to handle cryptographic operations and file processing.

4.1 Key Features

Key Generation and Management

- Generate secure RSA key pairs with a key size of 2048 bits.
- Automatically manage key expiration and rotation, ensuring secure cryptographic practices.
- Save and load keys for reuse, minimizing the need for repeated key generation.

Message Encryption and Decryption

- Encrypt plaintext messages using the RSA public key.
- Decrypt ciphertexts using the corresponding private key.
- Display encrypted and decrypted messages within the application for user convenience.

File Encryption and Decryption

- Support for encrypting text files, images, and PDFs.
- Utilize Optical Character Recognition (OCR) to extract text from files before encryption.
- Handle decryption of encrypted files, restoring them to their original content or format.

Session Management

- Use Flask sessions to store temporary data, such as encrypted and decrypted messages, during a user session.
- Provide feedback to users through session-based notifications for errors or successful operations.

User Interface

- A clean, responsive web interface for performing cryptographic operations.
- Display metadata such as key expiration status, helping users keep track of key validity.

Security Features

- Secure storage and handling of cryptographic keys in a designated directory.
- Cleanup mechanisms to ensure old keys are removed when new keys are generated.
- Protection against expired keys by enforcing key rotation when necessary.

Extensibility

- Modular design allows for easy integration of additional cryptographic algorithms or features.
- Customizable templates and functionality to support advanced encryption workflows.

5 Conclusion

In summary, this project illustrates:

- A robust **key management** lifecycle, with metadata-driven key generation, expiration checks, and automatic rotation.
- A detailed **implementation** of the RSA cryptosystem in Python, featuring robust key generation, secure padding mechanisms, and timing attack mitigations.
- The **vulnerability** of small RSA keys to naive factoring, emphasizing the need for key sizes of at least 2048 bits for practical security.
- A **timing attack** demonstration that reveals how measuring decryption times can potentially leak sensitive information when cryptographic operations are not constant-time.
- A **Web app** that demonstrates the project's features andbhow they can be applied into real world examples.

Future enhancements could include:

- Use OAEP padding instead of PKCS1 v1.5
- More rigorous integration with a secure keystore or hardware security module (HSM).
- Tools for rotating keys based on usage thresholds (not just a time-based expiration).
- Countermeasures for timing attacks (constant-time modular exponentiation, blinding techniques, etc.).