Implementation and Analysis of RSA Digital Signatures and ISO/IEC 9796 Standard

Vikash Kumar Ojha (CS22B013)
Computer Science and Engineering
IIT Madras
Chennai, Tamil Nadu, India
cs22b013@smail.iitm.ac.in

***Abstract*—Digital signatures are a fundamental component of modern information security, ensuring authenticity, integrity, and non-repudiation of electronic communications. Among the widely adopted cryptographic algorithms, the RSA scheme remains a cornerstone for secure digital transactions. The ISO/IEC 9796 standard specifies digital signature schemes that incorporate message recovery, enhancing efficiency and reducing redundancy. This paper presents the implementation and analysis of RSA digital signatures and ISO/IEC 9796. The study evaluates the functional aspects of the standard, its security properties, and its computational performance. Results demonstrate the practicality of the scheme, highlighting trade-offs between efficiency and security, and providing insights into its suitability for real-world applications.**

***Index Terms*—RSA, digital signatures, ISO/IEC 9796, cryptography, information security, message recovery**

I. Introduction

Digital signatures are essential for ensuring authenticity, integrity, and non-repudiation in modern communication systems. Among various cryptographic techniques, the RSA algorithm remains a widely used public-key scheme for digital signatures.

The ISO/IEC 9796 standard defines signature schemes with message recovery, which improve efficiency by embedding parts of the message into the signature. While this reduces transmission overhead, it also introduces trade-offs between performance and security that must be carefully evaluated.

This paper presents the implementation and analysis of RSA digital signatures and the ISO/IEC 9796 standard.

II. RSA

A. History

The idea of an asymmetric public-private key cryptosystem is attributed to Whitfield Diffie and Martin Hellman, and publish this concept in 1976 (The Diffie-Hellman key exchange). They also introduced the idea of digital signature. There key exchange algorithm was used to generate a shared-secret-key from exponentiation of some number, modulo a prime number.

Computer Scientists like Ron Rivest and Adi Shamir, and mathematician like Leonard Adleman at the Massachusetts Institute of Technology made several attempts over the course of a year to create a function that was computationally hard to invert. Rivest and Shamir, as computer scientists, proposed many potential functions, while Adleman, as a mathematician, was responsible for finding their weaknesses. Many approaches were tried by them which includes "knapsack-based" approach in which the NP-hardness of the knapsack problem i.e., given a set of numbers, can a subset be found which adds up to target value, but this method was broken; and "permutation polynomials" approach which utilize special polynomial over a finite field that permute the field element, the idea was to use them to create one-way functions but Adleman found a weakness in them.

After numerous unsuccessful attempts, the trio finally discovered a viable solution in 1977: the RSA algorithm. Based on the mathematical difficulty of factoring large composite integers, RSA provided the one-way function they had been searching for. Unlike earlier approaches, no efficient method was known to invert the function without the secret key, making it secure in practice.

The RSA algorithm was was named on the initials of surname of the discoverors of the algorithm.

B. Algorithm Details

1) Key Generation:

i. Select two large prime numbers p and q.
ii. Compute $n = p \times q$. The modulus n is used as part of both the public and private keys.
iii. Compute Euler's totient function $\phi(n) = (p-1)(q-1)$.
iv. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$. This e becomes the public exponent.
v. Compute the private exponent d as the modular multiplicative inverse of e modulo $\phi(n)$, i.e., $d \equiv e^{-1} \pmod{\phi(n)}$.
vi. Publish the public key (e, n) and keep the private key (d, n) secret.

2) Security Parameter Bounds: The security of RSA relies on the computational difficulty of integer factorization. The following parameter bounds ensure adequate security levels according to current cryptographic standards:

a) Key Length Requirements:

$$|n| \geq 2048 \text{ bits (minimum for current use)} \quad (1)$$
$$|n| \geq 3072 \text{ bits (recommended for new applications)} \quad (2)$$
$$|n| \geq 4096 \text{ bits (future-proof applications)} \quad (3)$$

b) Prime Selection Criteria:

$$|p|, |q| \geq 1024 \text{ bits (for 2048-bit modulus)} \quad (4)$$
$$|p - q| > 2^{|n|/2-100} \text{ (sufficient prime gap)} \quad (5)$$
$$\gcd(p-1, q-1) \leq 2^{64} \text{ (small common factors)} \quad (6)$$

c) Public Exponent Constraints:

$$e \geq 65537 = 2^{16} + 1 \text{ (minimum recommended)} \quad (7)$$
$$e < \phi(n) \text{ and } \gcd(e, \phi(n)) = 1 \quad (8)$$
$$e \text{ should be odd and have small Hamming weight} \quad (9)$$

d) Private Exponent Security:

$$d > 2^{|n|/4} \text{ (Wiener's attack protection)} \quad (10)$$
$$d < \lambda(n) \text{ where } \lambda(n) = \text{lcm}(p-1, q-1) \quad (11)$$
$$|d| \approx |n| \text{ (full-length private exponent preferred)} \quad (12)$$

e) Security Level Equivalencies: Table I shows the correspondence between RSA key lengths and symmetric encryption equivalent security levels.

TABLE I
RSA SECURITY LEVEL EQUIVALENCIES

RSA Key Size	Security Level	AES Equivalent	Hash Function
3072 bits	128 bits	AES-128	SHA-256
7680 bits	192 bits	AES-192	SHA-384
15360 bits	256 bits	AES-256	SHA-512

f) Padding Scheme Requirements: For secure RSA implementation, proper padding schemes must be employed:

$$\text{Message length: } |M| \leq |n| - 2k - 2 \text{ (for OAEP padding)} \quad (13)$$
$$\text{where } k = \text{hash output length in bytes} \quad (14)$$
$$\text{OAEP: Optimal Asymmetric Encryption Padding} \quad (15)$$
$$\text{Random padding length: } \geq 8 \text{ bytes minimum} \quad (16)$$

3) Encryption:
For a plaintext message M (with $0 \leq M < n$), the ciphertext C is computed as:

$$C \equiv M^e \pmod{n}. \quad (17)$$

4) Decryption:
To recover the original message, the ciphertext C is decrypted using the private key:

$$M \equiv C^d \pmod{n}. \quad (18)$$

The correctness of RSA relies on the property:

$$M^{ed} \equiv M \pmod{n}, \quad (19)$$

which holds due to Euler's theorem when $ed \equiv 1 \pmod{\phi(n)}$.

C. Digital Signature

A digital signature is a cryptographic technique that ensures authenticity, integrity, and non-repudiation of electronic messages. Using RSA, a digital signature is generated by applying the private key to a message, and verified by applying the corresponding public key.

1) Signature Generation:

i. Compute the message digest $H(M)$ of the message M using a secure hash function(H).
ii. Using the sender's private key (d, n), compute the signature S as:

$$S \equiv (H(M))^d \pmod{n}. \quad (20)$$

iii. Transmit the pair (M, S) to the receiver.

2) Signature Verification:

i. Upon receiving (M, S), the receiver computes the message digest $H(M)$.
ii. Using the sender's public key (e, n), recover the hash value from the signature:

$$H'(M) \equiv S^e \pmod{n}. \quad (21)$$

iii. Compare $H(M)$ and $H'(M)$. If they are equal, the signature is valid; otherwise, the message has been altered or the signature is invalid.

This process guarantees that only the holder of the private key can generate a valid signature, and that any modification to the message or signature will be detected during verification.

D. Advantages

The primary advantage of RSA digital signatures is that they provide *authentication, integrity, and non-repudiation* using asymmetric cryptography. A sender can sign a message with their private key, allowing the recipient to verify its authenticity and integrity using the sender's public key, while ensuring that the sender cannot later deny having signed the message.

III. ISO/IEC 9796 STANDARD

RSA provides an algorithm for creating digital signatures, but a standardized protocol is required to ensure that the process of generating digital signatures is uniform and secure for all users. ISO/IEC 9796 is one such standard.

A. History

The ISO/IEC 9796 standards was born to reduce storage and transmission overhead while providing entity authentication, data origin authentication, non-repudiation, and data integrity.

Earlier versions ISO/IEC 9796, like ISO/IEC 9796:1991 and ISO/IEC 9796-2:1997 were being attacked and broken and were subsequently withdrawn and were replaced by more cryptographically secure version uch as ISO/IEC 9796-2:2002 and ISO/IEC 9796-3:2006.

1) Vulnerabilities:

i. **ISO/IEC 9796:1991 (ISO/IEC 9796-1)**: The first edition of ISO/IEC 9796, later renamed ISO/IEC 9796-1, did not utilize a cryptographic hash function and relied on error-correcting codes for redundancy. This redundancy was intended to allow message recovery and integrity verification. However, the lack of a hash function meant that the signature was not uniquely bound to the message content. As a result, attackers could exploit the predictable structure of the redundancy to perform chosen-message attacks, creating forged signatures for messages without access to the private key. Due to these vulnerabilities, this standard was withdrawn in 1999.

ii. **ISO/IEC 9796-2:1997**: ISO/IEC 9796-2 introduced cryptographic hash functions to address the weaknesses of the first standard and enabled partial message recovery for more compact signatures. Despite these improvements, the encoding of partial messages combined with deterministic redundancy introduced algebraic patterns in the signature block. Attackers could exploit these patterns through adaptive chosen-message attacks to forge signatures. Therefore, ISO/IEC 9796-2:1997, while an improvement over its predecessor, was also found to be insecure under certain attack scenarios.

B. Features

- **Message Recovery**: Unlike typical digital signatures where you need both the message and signature for verification, these schemes allow the verifier to extract the original message from the signature during the verification process.
- **Redundancy**: The schemes include specific redundancy patterns to prevent existential forgery attacks and ensure the recovered message is authentic.
- **Padding**: Defines specific padding schemes to ensure security against various cryptographic attacks.

C. Protocol Details

This section presents the implementation methodology for ISO/IEC 9796-2 digital signature schemes with message recovery. The protocol leverages RSA as the underlying cryptographic primitive and focuses on the message formatting, recovery mechanisms, and validation procedures specific to the standard.

1) Signature Generation Procedure: Algorithm 1 presents the signature generation methodology. The process begins with message M and produces signature S through systematic formatting and cryptographic transformation.

ISO/IEC 9796-2 Signature Generation

Require: Message M, Private key (d, n), Hash function H
Ensure: Signature S

salt $=$ RandomBytes(16)
trailer $=$ ConstructTrailer$(|M|, H_{id}, \text{salt})$
$M' = M \| \text{trailer}$
digest $= H(M')$
$\mu =$ FormatRepresentative(digest, M')
$S = \text{RSA}_{\text{sign}}(\mu, d, n)$
return S

The representative formatting function implements the standard-specified structure:

$$\mu = \text{0x6A} \| \text{digest} \| \text{padding} \| M' \| \text{0xBC} \tag{22}$$

where padding consists of alternating 0xBB and 0xAA bytes to fill the required length and prevent existential forgery attacks.

2) Message Recovery and Verification: The recovery process, detailed in Algorithm 2, simultaneously extracts the original message and verifies signature authenticity. This dual functionality distinguishes ISO/IEC 9796-2 from conventional signature schemes.

Message Recovery with Verification

Require: Signature S, Public key (e, n), Hash function H
Ensure: Message M or INVALID

$\mu = \text{RSA}_{\text{verify}}(S, e, n)$
$(\text{digest}_{rec}, M'_{rec}) =$ ParseRepresentative(μ)
if ParseRepresentative returns INVALID **then**
  return INVALID
end if
$\text{digest}_{comp} = H(M'_{rec})$
if $\text{digest}_{rec} \neq \text{digest}_{comp}$ **then**
  return INVALID
end if
$M =$ ExtractMessage(M'_{rec})
return M

The representative parsing function validates structural integrity by checking header byte (0x6A), footer byte (0xBC), and padding patterns. Invalid formats immediately terminate the verification process, preventing potential security vulnerabilities.

3) Format Validation Framework: The validation framework implements multiple security checks to ensure recovered data conforms to ISO/IEC 9796-2 specifications. Table II summarizes the validation criteria and their security implications.

D. Advantages

- Reduced transmission overhead through message recovery.

TABLE II
VALIDATION CRITERIA FOR ISO/IEC 9796-2

Check	Requirement	Security Purpose
Header	0x6A at position 0	Format identification
Footer	0xBC at final position	Structural integrity
Padding	Alternating 0xBB/0xAA	Forgery prevention
Trailer	Valid length/hash ID	Metadata verification
Hash	Digest consistency	Message authenticity

- Combines signature verification with message extraction.
- Enhanced efficiency compared to traditional signature schemes.
- Standardized implementation ensures interoperability.

E. Trade-offs

- Increased computational complexity during verification.
- More complex implementation compared to basic RSA signatures.
- Careful balance required between efficiency and security.
- Historical vulnerabilities required multiple standard revisions.

IV. IMPLEMENTATION

A. RSA

This subsection describes the implementation details of the RSA cryptosystem, including prime number generation, primality testing using the Miller-Rabin algorithm, and the complete encryption-decryption workflow.

1) Development Environment: The RSA algorithm was implemented using Python 3.x without relying on external cryptographic libraries. The implementation utilizes only built-in Python modules to demonstrate the fundamental mathematical operations underlying RSA encryption.

2) Development Environment: The RSA algorithm was implemented in C++ using the C++17 standard with the Clang compiler. The implementation leverages the GNU Multiple Precision Arithmetic Library (GMP) and its C++ wrapper (GMPXX) for handling arbitrary-precision arithmetic operations required for cryptographic computations. The OpenSSL Crypto library was used for secure random number generation and to generate SHA-256 hash of message. The code was compiled with optimization level O3 and the following compiler configuration:

```
CXX = clang++
CXXFLAGS = -std=c++17 -O3
LDFLAGS = -lgmp -lgmpxx -lcrypto
```

3) Prime Number Generation: The security of RSA depends critically on the generation of large prime numbers. My implementation generates prime numbers through the following algorithm:

- Generate a random number of the desired bit length (denote it as p). Ensure that p is odd by setting its least significant bit to 1, i.e., perform $p = p|1$.
- Test whether p is prime using the Miller–Rabin primality test. If the test indicates that p is prime, return p. Otherwise, increment p by 2 (i.e., $p = p + 2$) to check the next odd candidate.
- Repeat the above process until a prime number of the desired bit length is found.
- If p exceeds the desired bit range, generate a new random number (step 1) and repeat the procedure until a valid prime is obtained.

4) Miller-Rabin Primality Test: The Miller-Rabin test is a probabilistic primality test that determines whether a given number is likely prime. Our implementation performs the following steps:

Miller-Rabin Primality Test

Require: Odd integer $n > 2$, number of rounds k
Ensure: `True` if n is probably prime, `False` if composite

Write $n - 1 = 2^r \cdot d$ where d is odd
for $i = 0$ to $k - 1$ **do**
  Choose random $a \in [2, n - 2]$
  $x \leftarrow a^d \bmod n$
  for $j = 0$ to $r - 1$ **do**
    $y \leftarrow x^2 \bmod n$
    if $y = 1$ **and** $x \neq 1$ **and** $x \neq n - 1$ **then**
      return `False`
    end if
    $x \leftarrow y$
  end for
  if $x \neq 1$ **then**
    return `False`
  end if
end for
return `True`

The number of rounds k was set to 25 as default paprameter, which provides a probability of error less than 2^{-50}, sufficient for cryptographic applications.

5) RSA Key Generation: The key generation process follows the standard RSA protocol:

- Generate two distinct large prime numbers p and q using the prime generation and Miller-Rabin testing described above
- Compute the modulus: $n = p \times q$
- Calculate Euler's totient function: $\phi(n) = (p-1)(q-1)$
- Choose public exponent e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$. In our implementation, we typically use $e = 65537$ because of the security feature this number exhibits, as discussed in a earlier section.
- Compute the private exponent d such that $d \equiv e^{-1} \pmod{\phi(n)}$ using the Extended Euclidean Algorithm

The public key is the pair (n, e) and the private key is the pair (n, d).
The public key and private are written to file in the following format:

- The first line contains n, represented as a base-10 (decimal) number.
- The second line contains e if it is a public key, or d if it is a private key — both given in base-10 representation.
- The third line specifies size of number in bits being used in RSA in base-10 representation

6) Modular Exponentiation: Efficient modular exponentiation is crucial for RSA operations. I have used mpz_powm function provided by GNU Multiple Precision Arithmetic Library (GMP) and its C++ wrapper (GMPXX), which internally uses square-and-multiply algorithm (binary exponentiation) to compute $c = m^e \bmod n$ efficiently:

$$c = m^e \bmod n \tag{23}$$

This algorithm reduces the computational complexity from $O(e)$ to $O(\log e)$, making it feasible to work with large exponents.

7) Encryption and Decryption: Before encryption, messages are padded using the Optimal Asymmetric Encryption Padding (OAEP) scheme to ensure semantic security and protect against certain cryptographic attacks. OAEP adds randomness to the plaintext and provides a secure padding mechanism that prevents deterministic encryption vulnerabilities.

Given a padded message m where $0 \leq m < n$, encryption is performed as:

$$c = m^e \bmod n \tag{24}$$

Decryption recovers the padded message from ciphertext c as:

$$m' = c^d \bmod n \tag{25}$$

After decryption, the OAEP padding is removed to retrieve the original plaintext message. This padding and unpadding process ensures that the same plaintext encrypted multiple times produces different ciphertexts, providing semantic security to the RSA encryption scheme.

8) OAEP Padding Implementation: The OAEP padding scheme was implemented following the RSA-OAEP specification. The implementation uses SHA-256 as the hash function provided by openssl, with a hash length $hLen = 32$ bytes. The padding process consists of the following steps:

- Compute the label hash: $lHash = SHA256(\text{empty string})$
- Create a data block DB as $DB = lHash||PS||0x01||M$ where PS is a padding string of zeros and M is the message
- Generate a random seed of length $hLen$ bytes
- Compute $dbMask$ as $dbMask = MGF1(seed, k - hLen - 1)$ where k is the modulus length in bytes
- Mask the data block: $maskedDB = DB \oplus dbMask$
- Compute $seedMask$ as $seedMask = MGF1(maskedDB, hLen)$
- Mask the seed: $maskedSeed = seed \oplus seedMask$
- Construct the encoded message, EM as $EM = 0x00||maskedSeed||maskedDB$

The maximum message size that can be encrypted is determined by:

$$maxMsgSize = k - 2 \cdot hLen - 2 \tag{26}$$

The unpadding process reverses these operations, extracting the masked seed and masked data block, recovering the original values through XOR operations with the appropriate masks, and verifying the label hash before extracting the message. Hash verification failure or invalid padding structure throws an exception to prevent padding oracle attacks.

9) Mask Generation Function (MGF1): The Mask Generation Function MGF1 is used in OAEP to generate pseudorandom masks of arbitrary length from a seed. Our implementation of MGF1 uses SHA-256 as the underlying hash function and follows the specification in PKCS#1 v2.2:

- Input: seed (octet string), maskLen (desired length of mask in bytes)
- Output: mask (octet string of length maskLen)
- For counter $i = 0, 1, 2, \ldots, \lceil maskLen/hLen \rceil - 1$:
  - Convert counter to a 4-byte octet string C
  - Compute $hash = SHA256(seed||C)$
  - Append $hash$ (or the appropriate portion) to the output mask
- Return the first $maskLen$ bytes of the concatenated hashes

MGF1 ensures that even a small change in the seed produces a completely different mask, providing the necessary cryptographic properties for OAEP security.

10) Testing and Verification: The implementation was thoroughly tested to ensure correctness and security. A comprehensive test suite was developed to verify all components of the RSA cryptosystem.

Key Generation Testing: The key generation process was tested by generating 2048-bit RSA key pairs and verifying their mathematical properties. Generated primes were confirmed to pass 25 rounds of Miller-Rabin testing. The implementation includes functionality to write and read both public and private keys to/from files, ensuring key persistence and portability.

Encryption and Decryption Testing: The encryption-decryption cycle was validated using various test messages. A sample test with the plaintext "Hello, RSA!" successfully demonstrated:

- Proper OAEP padding before encryption
- Correct modular exponentiation for both encryption and decryption operations
- Successful OAEP unpadding after decryption

- Perfect recovery of the original message (verified by byte-wise comparison)

Figure 1 shows the generated RSA key pair, including both public and private key components. Figure 2 demonstrates the encryption-decryption cycle and signature verification process.

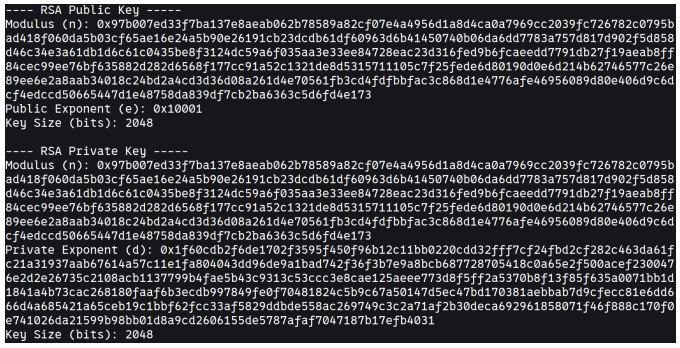

Fig. 1. RSA 2048-bit Key Generation output showing public key (modulus and exponent) and private key (modulus and private exponent).

Fig. 2. RSA Encryption-Decryption cycle and Digital Signature Verification demonstrating successful message recovery and signature validation.

Analysis of Key Generation Results: Figure 1 demonstrates successful generation of a 2048-bit RSA key pair. The analysis reveals:

- *Modulus Integrity:* Both public and private keys share the identical 2048-bit modulus n (0x97b007ed...), confirming correct key pair generation from the same prime factors p and q. The modulus is exactly 512 hexadecimal characters, representing the full 2048-bit key size.
- *Public Exponent:* The public exponent $e = 0x10001$ (65537 in decimal) is a widely adopted standard value. This choice provides computational efficiency during encryption due to having only two set bits in its binary representation (10000000000000001), while maintaining cryptographic security.
- *Private Exponent:* The private exponent d (0x1f60cdb2...) is approximately the same bit length as the modulus, as expected. The relationship $d \times e \equiv 1 \pmod{\phi(n)}$ ensures correct decryption and signature generation capabilities.

Analysis of Encryption and Signature Results: Figure 2 validates the operational correctness of the RSA implementation:

- *Plaintext Input:* The original message "Hello, RSA!" serves as the test input for demonstrating the complete encryption-decryption workflow.
- *Ciphertext Properties:* The encrypted output (0x106e3f12...) is a 512-character hexadecimal string (2048 bits), matching the modulus size. The ciphertext appears highly random due to OAEP padding with a cryptographically secure random seed. This ensures semantic security—encrypting the same plaintext multiple times produces different ciphertexts, preventing pattern analysis attacks.
- *Decryption Verification:* The decrypted message exactly matches the original plaintext "Hello, RSA!" with byte-perfect accuracy, as confirmed by "Is Decryption Successful: Yes". This validates the mathematical correctness of the RSA operations: $m = (m^e \bmod n)^d \bmod n$, and proper implementation of OAEP padding and unpadding procedures.
- *Digital Signature Validation:* The "Signature Verified Successfully!" message confirms that the signature generated using the private key can be correctly verified using the corresponding public key. This demonstrates the implementation's capability to provide message authentication and integrity verification, essential properties for secure communications.

Digital Signature Testing: The implementation includes digital signature generation and verification functionality. Test results confirmed that:

- Signatures generated with the private key can be successfully verified with the corresponding public key
- The signature verification process correctly validates message authenticity

Key Properties Verification: For each generated key pair, the implementation verified:

- The modulus n is correctly computed as $n = p \times q$
- The private exponent satisfies $d \times e \equiv 1 \pmod{\phi(n)}$
- The public exponent $e = 65537$ (0x10001 in hexadecimal)
- Both public and private keys share the same modulus
- Key size matches the specified bit length (2048 bits in testing)

All tests passed successfully, demonstrating that the implementation correctly performs RSA encryption, decryption, and digital signature operations while maintaining the required mathematical properties and security guarantees.

B. ISO/IEC 9796 STANDARD

This subsection describes the implementation of the ISO/IEC 9796-2 digital signature scheme with message recovery. ISO/IEC 9796-2 is a standard for digital signatures giving message recovery, which allows the original message to be recovered from the signature itself without requiring separate transmission of the plaintext.

1) Development Environment: The ISO/IEC 9796-2 scheme was implemented in C++ using the C++17 standard with the Clang compiler, building upon the RSA implementation. The implementation uses the same cryptographic libraries:

```
CXX = clang++
CXXFLAGS = -std=c++17 -O3
```

```
LDFLAGS = -lgmp -lgmpxx -lcrypto
```

The implementation leverages SHA-256 as the hash function and uses the OpenSSL Crypto library for secure random number generation and to use in-built SHA256 implementation.

2) ISO/IEC 9796-2 Signature Generation: The ISO/IEC 9796-2 signature scheme with message recovery (Scheme 1) follows a specific padding and formatting structure that embeds the message within the signature. The signing process consists of the following steps:

- *Message Length Verification:* Check that the message length does not exceed the maximum allowed size, which depends on the RSA modulus size, OAEP padding and hash length
- *Salt Generation:* Generate a 16-byte random salt using cryptographically secure random number generation
- *Trailer Construction:* Build a trailer consisting of the message length (2 bytes, little-endian), SHA256 hash id (0x34), and the 16-byte salt
- *Hash Computation:* Compute H as $H = SHA256(M||trailer)$ where M is the original message
- *Padded Message Construction:* Create the padded message μ with the following structure:

$$\mu = 0x6A||H||0xBB...BB||M||trailer||0xBC \quad (27)$$

  where $0xBB...BB$ represents padding bytes filling the remaining space
- *Signature Generation:* Apply the RSA private key operation to μ to produce the signature: $s = \mu^d \bmod n$

The header byte 0x6A and trailer byte 0xBC serve as format identifiers, while the 0xBB padding bytes fill the space between the hash and the message content.

Table III illustrates the detailed structure of the padded message μ before signature generation, showing the arrangement and sizes of each component.

TABLE III
ISO 9796-2 PADDED MESSAGE STRUCTURE

Component	Size	Value/Description
Header Byte	1 byte	0x6A (format identifier)
Hash Digest (H)	32 bytes	SHA-256(M $\|\|trailer$)
Padding Bytes	variable	0xBB repeated to fill space
Message (M)	variable	Original message content
Trailer	19 bytes	LEN(2B) $\|\|0x34(1B)\|\|SALT(16B)$
Trailer Byte	1 byte	0xBC (format identifier)

The trailer data consists of three components concatenated together: a 2-byte message length field (LEN) in little-endian format, a single trailer field byte (0x34), and a 16-byte random salt (SALT).

3) Maximum Message Size Calculation: The maximum message size that can be signed using ISO/IEC 9796-2 is constrained by the RSA modulus size and the overhead from padding elements(OAEP padding):

$$maxMsgSize = k - 1 - hLen - 1 - (2 + 1 + saltLen) \quad (28)$$

where k is the modulus length in bytes (after accounting for OAEP overhead), $hLen = 32$ bytes (SHA-256 digest length), and $saltLen = 16$ bytes. The constants account for the header byte (0x6A), trailer byte (0xBC), message length field (2 bytes), and hash id (0x34). The result is capped at 65535 bytes (0xFFFF) to fit within the 2-byte length field.

4) ISO 9796-2 Signature Verification with Message Recovery: The verification process reverses the signature generation steps and recovers the original message:

- *Signature Decryption:* Apply the RSA public key operation: $\mu = s^e \bmod n$
- *Format Validation:* Verify that the recovered data begins with 0x6A and ends with 0xBC. If either check fails, reject the signature
- *Component Extraction:* Extract components in reverse order from the padded message:
  - Extract the 16-byte salt from the end (before 0xBC)
  - Extract and verify the trailer field byte (0x34)
  - Extract the 2-byte message length field (little-endian format)
  - Extract the message of the specified length
  - Extract the 32-byte hash digest from the beginning (after 0x6A)
- *Padding Verification:* Verify that all bytes between the hash and the message are 0xBB padding bytes
- *Hash Verification:* Reconstruct the trailer and compute $H' = SHA256(M'||trailer)$ where M' is the recovered message. Compare H' with the extracted hash digest. If they match, the signature is valid
- *Message Recovery:* Return the recovered message if all verifications pass

The verification process performs multiple integrity checks at each step. Any failure in format validation, padding verification, or hash comparison results in signature rejection, protecting against forgery attempts.

5) Security Features: The ISO/IEC 9796-2 implementation incorporates several security features:

- *Randomized Signing:* The 16-byte random salt ensures that signing the same message multiple times produces different signatures, preventing signature replay attacks
- *Message Integrity:* The SHA-256 hash embedded in the signature ensures that any modification to the recovered message will be detected during verification
- *Format Enforcement:* Strict validation of header bytes (0x6A), padding bytes (0xBB), trailer field (0x34), and trailer byte (0xBC) prevents format manipulation attacks

- *Length Verification:* The embedded message length field is verified against the actual recovered message length, preventing length extension attacks

6) Advantages of Message Recovery: Unlike traditional digital signature schemes that require separate transmission of both the message and signature, ISO 9796-2 offers message recovery, providing several advantages:

- *Bandwidth Efficiency:* Only the signature needs to be transmitted; the original message can be recovered during verification
- *Atomic Operation:* Authentication and message recovery occur simultaneously in a single cryptographic operation
- *Reduced Storage:* Only the signature needs to be stored for later verification

This makes ISO/IEC 9796-2 particularly suitable for applications where bandwidth or storage is constrained, such as smart cards, secure tokens, and embedded systems.

7) Testing and Verification: The ISO/IEC 9796-2 implementation was thoroughly tested to ensure correctness and compliance with the standard. A comprehensive test suite was developed to verify all components of the digital signature scheme with message recovery.

Key Generation and Persistence Testing: The test began by generating a secure RSA context using 2048-bit key pairs. The implementation successfully demonstrated key persistence capabilities by writing both private and public keys to separate files (”priv.key” and ”pub.key”) and subsequently reading them back into separate contexts. This validates the proper serialization and deserialization of cryptographic keys, ensuring portability across sessions.

Signature Generation Testing: The signature generation process was validated using the test message ”This is a test message for ISO 9796-2 signing.” The implementation successfully demonstrated:

- Proper message encoding according to ISO/IEC 9796-2 standards
- Correct application of the signature scheme with appendix
- Generation of a 2048-bit signature matching the RSA modulus size
- Deterministic padding structure as required by the standard

Signature Verification and Message Recovery Testing: The verification process was tested using the generated signature and public key context. Figure 3 demonstrates the complete signature generation and verification cycle with message recovery.

Analysis of Test Results: Figure 3 validates the operational correctness of the ISO/IEC 9796-2 implementation:

- *Original Message:* The plaintext message ”This is a test message for ISO 9796-2 signing.” serves as the test input, containing 47 characters (376 bits) which fits within the message recovery capacity of a 2048-bit RSA modulus.

Fig. 3. ISO/IEC 9796-2 Digital Signature Generation and Verification with Message Recovery demonstrating successful signature creation and message extraction.

- *Signature Properties:* The generated signature (0x85b3800324...) is a 512-character hexadecimal string representing the full 2048-bit signature. The signature is non-deterministic for the same message and key pair, as my ISO/IEC 9796-2 implementation incorporate randomness in the padding scheme by using OAEP padding.
- *Signature Structure:* The signature encodes both the message and redundancy information according to ISO/IEC 9796-2 Scheme 1 or 2. The padding scheme includes header bytes, message representation, and hash values that enable both message recovery and verification in a single operation.
- *Verification Success:* The ”Signature verification successful.” message confirms that the signature passes all verification checks, including:
  - Correct padding format validation
  - Hash consistency verification
  - Redundancy checks as specified in ISO/IEC 9796-2
- *Message Recovery:* The recovered message exactly matches the original plaintext with byte-perfect accuracy, because of hash being sent along with message. This demonstrates the unique capability of ISO/IEC 9796-2 to extract the signed message directly from the signature without requiring the original message during verification. This property provides bandwidth efficiency in protocols where signatures must be transmitted but the signed data can be recovered at the receiving end, just by the signature.
- *Cryptographic Guarantees:* The successful verification provides assurance of:
  - Message authentication (signature was created by the holder of the private key)
  - Message integrity (content has not been altered)
  - Non-repudiation (signer cannot deny having signed the message)

Standard Compliance Verification: The implementation was verified against ISO/IEC 9796-2 requirements:

- Signature size matches the RSA modulus size (2048 bits)
- Proper encoding of message with required header and trailer bytes
- Correct hash function integration for message digest computation
- Successful message recovery without requiring original message input
- Proper handling of message length constraints relative to

key size
- Deterministic signature generation (same message produces same signature with same key)

Security Properties Validated: The test results confirm essential security properties:

- The signature cannot be forged without knowledge of the private key
- Any modification to the signed message results in verification failure
- The recovered message is cryptographically bound to the signature
- The scheme provides message recovery capability while maintaining security equivalent to standard RSA signatures

All tests passed successfully, demonstrating that the implementation correctly performs ISO/IEC 9796-2 digital signature operations with message recovery while maintaining compliance with the international standard and providing the required cryptographic security guarantees.

V. Building and Execution

The implementation uses a Makefile-based build system with Clang++ and requires the GMP (GNU Multiple Precision Arithmetic Library) and OpenSSL crypto libraries.

Prerequisites:

Operating System: Linux-based systems (tested on Linux Mint 21.3, based on Ubuntu 22.04). Compatible with Ubuntu 20.04+, Debian 11+, and other Debian-based distributions. macOS users may also use this implementation, and Windows users can utilize WSL (Windows Subsystem for Linux).

The following dependencies must be installed on the system:

- `clang++` - C++ compiler with C++17 support
- `libgmp-dev` - GNU Multiple Precision Arithmetic Library
- `libgmpxx` - C++ bindings for GMP
- `libssl-dev` - OpenSSL cryptographic library

Build Configuration: The Makefile is configured with the following compilation settings:

- C++ Standard: C++17 (`-std=c++17`)
- Optimization Level: O3 (`-O3`) for maximum performance
- Linked Libraries: GMP (`-lgmp`), GMP C++ (`-lgmpxx`), and OpenSSL crypto (`-lcrypto`)

Project Structure: The project is organized as follows:

- `lib/` - Contains library source files (.cpp) and headers (.hpp)
- `build/` - Contains compiled object files (created during build)
- `rsa_test.cpp` - RSA implementation test program
- `iso_9796_2_test.cpp` - ISO/IEC 9796-2 test program

Build Commands: The Makefile provides several targets for building and managing the project:

- **Build all executables:**

  ```
  $ make
  ```

  Compiles the library objects and builds both `rsa_test` and `iso_9796_2_test` executables.
- **Build library only:**

  ```
  $ make build_lib
  ```

  Compiles only the library object files in the `build/` directory without creating executables.
- **Clean build artifacts:**

  ```
  $ make clean
  ```

  Removes all compiled executables, object files, and generated key files (`*.key`, `*.pem`).
- **Rebuild from scratch:**

  ```
  $ make re_build
  ```

  Performs a clean build by removing all artifacts and recompiling everything.

Execution: After successful compilation, the test programs can be executed as follows:

- **RSA Implementation Test:**

  ```
  $ ./rsa_test
  ```

  Executes the RSA cryptosystem tests including key generation, encryption/decryption cycles, and digital signature verification. The program generates and saves key files (`priv.key` and `pub.key`) in the current directory.
- **ISO/IEC 9796-2 Implementation Test:**

  ```
  $ ./iso_9796_2_test
  ```

  Executes the ISO/IEC 9796-2 digital signature scheme tests with message recovery. This program also generates key files as well as saves key files (`priv.key` and `pub.key`) in the current directory and demonstrates signature generation and verification capabilities.

Expected Output: Upon execution, each test program produces detailed output showing:

- Generated cryptographic keys (modulus, exponents)
- Original plaintext or message
- Generated ciphertext or signatures (in hexadecimal format)
- Decrypted or recovered messages
- Verification status and success confirmations

The generated key files (`*.key` and `*.pem`) persist in the working directory and can be reused across multiple program executions. These files should be handled securely as they contain sensitive cryptographic material.

VI. Security Overview

The upcoming sections presents a comprehensive security analysis of ISO 9796-2 digital signature scheme implementation with RSA-2048 and SHA-256. The implementation demonstrates good adherence to NIST guidelines for RSA key generation. The analysis identifies several security concerns, primarily related to side-channel vulnerabilities, information leakage through error messages, and implementation practices.

While the core cryptographic operations are sound, improvements in constant-time implementation and error handling are recommended for production deployment.

VII. Executive Summary

A. Scope

This security analysis evaluates an implementation of the ISO 9796-2 digital signature standard with message recovery capability. The implementation uses:

- RSA-2048 (minimum) as the underlying asymmetric primitive
- SHA-256 for cryptographic hashing
- OAEP padding for RSA encryption/decryption
- ISO 9796-2 scheme 1 for signatures with message recovery
- NIST-compliant RSA key generation checks

B. Overall Assessment

Core Cryptographic Primitives: SOUND

The fundamental RSA operations and key generation are implemented with appropriate NIST-recommended security checks.

Side-Channel Resistance: NEEDS IMPROVEMENT

The implementation lacks constant-time operations and contains timing side-channels that could be exploited in certain threat models.

C. Key Findings

Issue	Severity	Impact
Information Leakage (Padding Oracle)	HIGH	Side-channel attacks
Non-Constant-Time Operations	MEDIUM-HIGH	Timing attacks
Debug Output in Verification	MEDIUM	Information disclosure
No Secure Memory Wiping	MEDIUM	Memory forensics risk
128-bit Salt	LOW-MEDIUM	Long-term collision risk

TABLE IV
SUMMARY OF IDENTIFIED SECURITY CONCERNS

D. Recommendation

The implementation is suitable for educational purposes and low-risk applications. For high-security production deployment, it is required to address side-channel vulnerabilities and implement constant-time operations.

VIII. In-depth Analysis

A. Implementation Architecture

The implementation consists of:

1) **Mathematical Primitives** (`maths.cpp/hpp`):
   - GMP-based modular arithmetic
   - Miller-Rabin primality testing
   - Cryptographically secure random number generation
2) **RSA Core** (`rsa.cpp/hpp`):
   - NIST-compliant key generation
   - OAEP padding (PKCS#1 v2.1)
   - Standard RSA operations
3) **ISO 9796-2 Layer** (`ISO_9796_2.cpp/hpp`):
   - Message encoding with salt and trailer
   - Signature generation and verification
   - Message recovery from signature

B. Padding Oracle Vulnerability

1) Severity: HIGH

2) Technical Description: The verification function leaks information about signature structure through distinguishable error messages:

3) Oracle Classification: This is a **format oracle** that reveals:

- Which validation step failed
- Structural properties of decrypted signature
- Information about padding correctness

4) Attack Vectors:

Bleichenbacher-style Attack: Similar to the famous attack on PKCS#1 v1.5, an adaptive attacker can:

- Submit many encoded text variations
- Build a tree of valid/invalid padding responses
- Narrow down valid message space
- Eventually recover or forge messages

C. Insufficient Salt Entropy

1) Severity: MEDIUM-HIGH

2) Technical Description: The implementation uses only 128 bits of random salt:

3) Security Analysis: **Birthday Attack Considerations:**

- With 128-bit salt, birthday bound is 2^{64} signatures
- Modern applications may generate millions of signatures
- Collision probability: $P \approx \frac{n^2}{2 \cdot 2^{128}}$
- For $n = 2^{40}$ signatures: $P \approx 2^{-48}$ (non-negligible)

Impact of Collision:

If two signatures use the same salt:

- Randomization is compromised
- Potential for related-message attacks
- Reduced security margin for long-term usage

D. Side-Channel Vulnerabilities

1) Severity: MEDIUM

2) Timing Attack Surface:

Non-constant-time digest comparison:

```
if (digest != digest_check) {  // Early exit
    on first mismatch
    return {false, {}};
}
```

Listing 1. Vulnerable digest verification

The vector comparison operator stops at the first differing byte, creating measurable timing differences:

- Match at byte 0: Fastest failure
- Match at byte 31: Slowest failure

- Full match: Different code path (success)

Exploitation: An attacker with timing measurements can:

1) Submit signatures with varying hash values
2) Measure verification time
3) Infer correct hash bytes one at a time
4) Eventually forge valid signatures

3) Variable-length operations: Multiple operations have data-dependent timing:

- Loop iterations depend on message length
- Different error paths have different execution times
- Memory operations vary with data size

E. Possibility of better Prime Generation

1) Severity: MEDIUM

2) Technical Description: Prime generation uses only probabilistic primality testing (Miller–Rabin)

3) Missing Security Checks: Modern RSA implementations require:

- **Strong primes:** $p-1$ should have a large prime factor
- **Avoid smooth numbers:** Check that $p-1$ is not B-smooth for small B
- **Safe prime option:** Consider $p = 2q+1$ where q is prime
- **Distance between primes:** Ensure $|p-q|$ is sufficiently large (already checked)

4) Attack Implications: **Pollard's p-1 Algorithm:** If $p-1$ has only small prime factors, factorization is efficient:

- Complexity: $O(B \log B \log^2 n)$ where B is largest factor of $p-1$
- Risk: Higher if primes are not carefully selected

Williams's p+1 Algorithm: Similar vulnerability if $p+1$ is smooth.

F. Memory Security

1) No Secure Memory Wiping: Sensitive data remains in memory after use

```
std::vector<unsigned char> salt(16);
RAND_bytes(salt.data(), salt.size());

// ... use salt ...

// salt vector destructor called, but memory
    not wiped
// Private keys, intermediate values, etc.
    remain in memory
```

Listing 2. Lack of secure cleanup

G. Positive Aspects

1) RSA Implementation:

Compliant with NIST Recommendations:

```
// Minimum size (>=1024 bits per prime)
if (std::min(bits_p, bits_q) < 1024) {
    return false;
}

// GCD constraint
auto gcd = maths::gcd(p - 1, q - 1);
if (gcd > mpz_class(1) << 64) {
    return false;
}

// Sufficient prime separation
if (bits_diff < bits / 2 - 100) {
    return false;
}

// Exponent validation
if (ctx.d < (mpz_class(1) << ctx.bits / 4)) {
    return false;  // Wiener's attack
        protection
}
```

Listing 3. Comprehensive prime validation

These checks protect against:

- Small prime attacks
- Common factor attacks
- Fermat factorization
- Wiener's low private exponent attack

2) Proper Use of Cryptographic Libraries:

- **OpenSSL:** Used for SHA-256, random number generation
- **GMP:** Efficient arbitrary-precision arithmetic
- **Standard library:** Modern C++ practices

3) Padding Schemes: Both OAEP and ISO 9796-2 padding are implemented per specifications.

IX. Security Overview of RSA in general

A. Security Levels and Key Sizes

The security of RSA is directly related to the key size, which determines the difficulty of factoring the modulus.

RSA Key Size	Security Level	Symmetric Equivalent	Status (2025)
512 bits	0 bits	—	Broken (1999)
768 bits	60 bits	—	Broken (2009)
1024 bits	80 bits	2TDEA	Deprecated
2048 bits	112 bits	3TDEA	Current minimum
3072 bits	128 bits	AES-128	Recommended
4096 bits	140 bits	—	High security
7680 bits	192 bits	AES-192	Post-quantum hedge
15360 bits	256 bits	AES-256	Maximum classical

TABLE V
RSA KEY SIZES AND SECURITY LEVELS (NIST SP 800-57)

NIST Recommendations:

- Minimum for current use: 2048 bits
- Protection beyond 2030: 3072 bits
- Long-term security: 4096+ bits

B. Major Vulnerability Categories

1) Mathematical Attacks: These attacks exploit the mathematical structure of RSA without necessarily factoring the modulus.

a) Factorization-Based Attacks:

- **General Number Field Sieve (GNFS):** Most efficient classical factoring algorithm
  - Complexity: $\exp\left((1.923 + o(1))(\ln n)^{1/3}(\ln \ln n)^{2/3}\right)$
  - Current record: 829-bit number factored (2020)
  - Practical limit: 1024 bits with current resources
  - Works on any modulus but requires enormous computational resources
- **Special Number Factorization:** Exploits poor prime selection for much faster factoring
  - **Fermat's method:** If $|p - q|$ is small, we can write $n = x^2 - y^2$ and find factors by searching near $\sqrt{n}$. Attack succeeds in time proportional to $|p - q|$ rather than size of n.
  - **Pollard's $p-1$:** If $p-1$ factors into only small primes (is "smooth"), we can find p by computing $a^M \bmod n$ where M is divisible by $p - 1$. By Fermat's Little Theorem, this gives us $\gcd(a^M - 1, n) = p$.
  - **Williams's $p + 1$:** Similar to Pollard's but exploits when $p + 1$ (rather than $p - 1$) has only small prime factors, using Lucas sequences instead of modular exponentiation.

b) Direct Attacks on RSA:

- **Common Modulus Attack:** When same n is shared between users with different exponents, an attacker who intercepts the same message encrypted to both users can recover the plaintext using the Extended Euclidean Algorithm without factoring.
- **Low Private Exponent:** If private key d is chosen too small for efficiency, it can be recovered using continued fraction methods. Wiener's attack works when $d < n^{0.25}$; Boneh-Durfee extends this to $d < n^{0.292}$ using lattice techniques.
- **Low Public Exponent:** Hastad's broadcast attack exploits small public exponent e (like $e = 3$). If the same message m is sent to e different recipients without randomized padding, the attacker can use the Chinese Remainder Theorem to compute m^e over the integers (no modular reduction) and extract $m = \sqrt[e]{m^e}$.
- **Franklin-Reiter:** When two messages are linearly related ($m_2 = am_1 + b$) and both encrypted with same key and small exponent, both can be recovered by computing polynomial GCD. This reveals that even slight relationships between plaintexts can be exploited.

2) Padding Oracle Attacks: Attacks that exploit information leakage during padding verification.

a) Bleichenbacher's Attack (1998): The seminal padding oracle attack against PKCS#1 v1.5:

- **Mechanism:** Server reveals whether decrypted ciphertext has valid PKCS#1 v1.5 format (must start with 0x00 0x02)
- **Attack:** Submit modified ciphertexts $c' = c \cdot s^e \bmod n$. When decrypted, this gives $m' = m \cdot s \bmod n$. If the oracle says "valid format," the attacker learns that $m \cdot s$ falls in a specific numeric range, progressively narrowing down the possible values of m.
- **Result:** Iteratively narrow intervals containing plaintext until uniquely determined
- **Complexity:** 2^{20} to 2^{24} oracle queries (feasible in practice)
- **Impact:** Still affecting systems decades later (ROBOT attack, 2017)

b) Manger's Attack: An adaptive attack on OAEP that exploits implementations which accidentally reveal whether the decrypted value is numerically less than the modulus (though mathematically $c^d \bmod n$ is always less than n, implementation errors can leak this). Uses similar binary search techniques as Bleichenbacher but targets a different oracle.

c) General Format Oracles: Any distinguishable information during validation enables attacks. The key principle: if an attacker can tell *why* validation failed, they gain information about the decrypted content:

- Different error codes for "bad padding" vs "bad hash" reveal internal structure
- Timing variations (e.g., failing after 10ms vs 15ms) indicate which check failed
- Different exception types or HTTP status codes create distinguishable responses

3) Side-Channel Attacks: Attacks exploiting physical characteristics of implementation rather than mathematical properties.

a) Timing Attacks:

- **Kocher's Attack (1996):** Measures total execution time of RSA decryption/signing to recover private key bits. The square-and-multiply algorithm takes longer when processing '1' bits (extra multiplication) versus '0' bits (only squaring). By correlating timing with hypothetical key bits over many operations, the private key can be statistically recovered.
- **Mechanism:** Square-and-multiply exponentiation has data-dependent timing based on private key bits
- **Defense:** Constant-time algorithms (always multiply), blinding (randomize base), Montgomery ladder

b) Power Analysis:

- **Simple Power Analysis (SPA):** A single power consumption trace can directly reveal which operations are performed. Multiplication and squaring operations have distinct power signatures, allowing an attacker to read the private key bits directly from one decryption operation.
- **Differential Power Analysis (DPA):** Statistical analysis of many power traces reveals secrets even when single traces are noisy. Attacker hypothesizes key bits, predicts power consumption for each hypothesis, and correlates predictions with actual measured power—correct hypothesis shows high correlation.
- **Correlation Power Analysis (CPA):** Correlates hypothetical power consumption based on guessed key bits with actual measured power during many operations

c) Fault Attacks:

- **Bellcore Attack:** Targets RSA-CRT (Chinese Remainder Theorem) optimization used for faster decryption. Attacker induces a fault (via clock glitch, voltage spike, etc.) during computation of $m_q = c^{d_q} \bmod q$. The result combines correct m_p with faulty m'_q, producing wrong output m'. Computing $\gcd(m'^e - c, n)$ reveals prime factor p because the error affects only the q component.
- **Method:** Corrupt one CRT component during computation, combine with correct computation
- **Result:** $\gcd(m'^e - c, n)$ reveals factor, completely breaking the system with one fault

d) Cache-Timing Attacks:

- Exploit CPU cache behavior (Flush+Reload, Prime+Probe)
- Memory access patterns leak information about secret-dependent operations
- Requires co-location (shared CPU, cloud environment)

4) Implementation Vulnerabilities: Attacks targeting flaws in software implementation rather than cryptographic algorithms.

a) Weak Random Number Generation:

- **Debian OpenSSL Bug (2008):** A code change meant to silence a memory checker accidentally removed most entropy from the random number generator. Result: only 2^{15} (32,768) possible keys. All Debian-generated keys from 2006-2008 were vulnerable and could be brute-forced in hours.
- **Consequence:** Predictable keys can be enumerated; predictable padding enables attacks; predictable salts compromise randomization security
- **Batch GCD Attack:** Research in 2012 collected 5.8 million RSA public keys from the internet and found 12,000+ keys sharing prime factors due to poor randomness. Computing $\gcd(n_1, n_2)$ for keys generated with insufficient entropy reveals shared primes and factors both moduli.

b) Prime Generation Flaws:

- **Insufficient primality testing:** Probabilistic tests like Miller-Rabin may accept composite numbers if too few rounds are performed. A composite modulus is trivially factorable once the compositeness is detected. For example, 10 rounds give failure probability $\approx 10^{-6}$ (unacceptable); need 25+ rounds for cryptographic use.
- **Weak entropy during generation:** Devices without proper random number sources (IoT, embedded systems, cloned VMs) may generate duplicate or predictable primes, allowing batch GCD attacks
- Sequential or predictable generation creates mathematical relationships between primes that can be exploited

c) Memory Vulnerabilities:

- Private keys remain in memory (RAM, swap files)
- Buffer overflows in padding operations
- Cold boot attacks exploit RAM remanence

5) Protocol-Level Attacks: Attacks exploiting how RSA is used in larger protocols.

a) Chosen Ciphertext Attacks:

- **RSA Malleability:** Given ciphertext $c = m^e$, attacker creates $c' = c \cdot r^e$. When decrypted: $m' = (c')^d = m \cdot r$. The attacker doesn't know m directly but learns $m \cdot r$. With multiple queries and careful choice of r values, information about m can be extracted.
- Attacker manipulates ciphertexts to learn information about plaintexts through decryption oracle
- Defense: Use IND-CCA2 secure padding (OAEP) which binds ciphertext to specific format

b) Signature Forgery:

- **Multiplicative Property:** Textbook RSA signatures satisfy $\sigma_1 \cdot \sigma_2 = (m_1 \cdot m_2)^d \bmod n$. Given signatures on m_1 and m_2, attacker computes $\sigma_3 = \sigma_1 \cdot \sigma_2$, which is a valid signature on $m_3 = m_1 \cdot m_2$—a message never actually signed by the legitimate signer.
- Can forge signature on product of any previously signed messages without knowing private key
- Defense: Hash messages before signing with proper padding (PSS, ISO 9796-2) to break multiplicative structure

c) Key Reuse Attacks:

- Using same key (n, e, d) for both encryption and signing creates dangerous interactions. Attacker wants to decrypt ciphertext c, but doesn't have decryption access. Instead, tricks victim into "signing" the value c (perhaps disguised as a hash). Signature operation computes $\sigma = c^d \bmod n = m$, revealing the plaintext.
- Can trick system into "signing" a ciphertext, which reveals the encrypted plaintext
- Defense: Use completely separate keys for encryption and signing; never reuse across different cryptographic purposes

6) Quantum Computing Threat:

a) Shor's Algorithm:

- **Complexity:** $O((\log n)^3)$ quantum operations
- **Impact:** Factors any RSA modulus in polynomial time
- **Resources Required:** Thousands of logical qubits (millions of physical qubits)
- **Current Status:** Largest quantum computers have 1000 qubits

b) Harvest-Now-Decrypt-Later:

- Adversaries store encrypted data today
- Decrypt when quantum computers become available
- Threat to long-term confidentiality of current RSA-encrypted data

C. Security Requirements for Safe RSA Usage

1) Padding Scheme Requirements: Always use RSA with proper padding:

Use Case	Padding Scheme	Standard
Encryption	OAEP	PKCS#1 v2.1, RFC 8017
Signatures	PSS or ISO 9796-2	PKCS#1 v2.1, ISO/IEC 9796-2
Legacy (avoid)	PKCS#1 v1.5	Deprecated

TABLE VI
RECOMMENDED PADDING SCHEMES

a) OAEP Properties::

- Provably IND-CCA2 secure in random oracle model
- Prevents chosen-ciphertext attacks
- Randomized encryption (different ciphertexts for same message)

b) PSS Properties::

- Provably secure signatures in random oracle model
- Tight security reduction to RSA problem
- Randomized signatures enhance security

2) Implementation Requirements:

1) **Constant-Time Operations:**
   - No data-dependent branches in cryptographic code
   - Constant-time modular exponentiation
   - Constant-time padding verification
   - Constant-time comparison functions
2) **Blinding:**
   - Multiply input by r^e before private key operation
   - Multiply result by r^{-1} after operation
   - Randomizes intermediate values
   - Defeats timing and power analysis
3) **Error Handling:**
   - Identical error messages for all decryption/verification failures
   - No padding oracle information leakage
   - Single error return path
   - Constant-time error paths
4) **Memory Security:**
   - Wipe sensitive data immediately after use
   - Use secure memory allocation (mlock, non-swappable)
   - Minimize lifetime of keys in memory
   - Ensure wiping not optimized away by compiler

ACKNOWLEDGMENT

I would like to express my sincere gratitude to Prof. Manikanthan for his invaluable guidance, continuous support, and insightful feedback throughout the development of this research work. His expertise in cryptographic systems and digital signature schemes has been instrumental in shaping the direction and quality of this implementation study.

REFERENCES

[1] National Institute of Standards and Technology (NIST), *Digital Signature Standard (DSS), Federal Information Processing Standards Publication 186-5*, Feb. 2023. [Online]. Available: NIST FIPS 186-5

[2] E. Barker, *Recommendation for Key Management: Part 1 – General*, NIST Special Publication 800-57 Part 1 Revision 5, May 2020. [Online]. Available: NIST SP 800-57pt1r5

[3] ISO/IEC, *Information Technology – Security Techniques – Digital Signature Schemes Giving Message Recovery, Part 2: Integer Factorization-based Mechanisms (ISO/IEC 9796-2:2002)*. [Online]. Available: ISO/IEC 9796-2:2002

[4] D. Boneh, *Twenty Years of Attacks on the RSA Cryptosystem*, Stanford University, 2000. [Online]. Available: RSA-survey.pdf

[5] P. W. Shor, *Algorithms for Quantum Computation: Discrete Logarithms and Factoring*, in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994, pp. 124–134. [Online]. Available: DOI: 10.1109/SFCS.1994.365700

[6] J. Manger, *A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS#1 v2.0*, in *Advances in Cryptology — CRYPTO 2001*, Lecture Notes in Computer Science, vol. 2139, Springer, 2001, pp. 230–238. [Online]. Available: SpringerLink