Đại học quốc gia TP.HCM
Trường đại học công nghệ thông tin

Ngành An toàn thông tin

Môn học: Mật mã học

Cryptanalysis on Asymmetric Ciphers: RSA & RSA-Based Signatures

Học viên:
Lê Trí Đức – 24520009
Phạm Nguyễn Thành Long - 24521011

Giảng viên:
Nguyễn Ngọc Tự

Contents

1 Overview

- Scenario: Secure service company uses RSA-based algorithms for securing transactions and digital signatures. They want to ensure the robustness of their RSA implementation against potential attacks.
- Gaps: While RSA is a widely accepted and used in public-key cryptosystem, improper implementations or usage of weak parameters can lead to vulnerabilities.
- Motivations: To ensure the integrity and confidentiality of financial transactions and to maintain the trust of clients and stakeholders.

2 Preliminaries

2.1 Key Generation

The keys for the RSA algorithm are generated in the following way:

1. Choose two large prime numbers p and q.
   (a) To make factoring infeasible, p and q must be chosen at random from a large space of possibilites, such as all prime number between 2^{1024} and 2^{1024}. It is recommend that we should generate two primes p, q with size 1024 bits or 1536 bits.
   (b) p and q are kept secret.
2. Compute $n = pq$
   (a) n is used as the modulus for both the public and private keys. Its length, usually express in bits, is the key length
   (b) n is released as part of the public key
3. Compute $\lambda(n)$, where λ is Carmichael's totient function. Since $n = pq, \lambda = lcm(\lambda(p), \lambda(q))$, and since p, q are prime, $\lambda(p) = \phi(p) = p - 1, \lambda(q) = \phi(q) = q - 1$. Hence $\lambda(n) = lcm(p - 1, q - 1)$
   (a) The lcm can be calculated through the Euclidean algorithm, since $lcm(a, b) = \dfrac{|ab|}{\gcd(a, b)}$.

(b) $\lambda(n)$ is kept secret.

4. Choose an integer e such that $1 < e < \lambda(n)$ and $\gcd(e, \lambda(n)) = 1$; that is, e and $\lambda(n)$ are coprime.

   (a) e having a short bit-length and small Hamming weight results in more efficient encryption - the most commonly chosen value for e is $2^{16} + 1 = 65537$. The smallest possible value for e is 3, but such a small value for e may expose vulnerabilities in insecure padding schemes.

   (b) e is released as part of the public key.

5. Determine d as $d \equiv e^{-1} (\mathrm{mod} \lambda(n))$; that is, d is the modular multiplicative inverse of e modulo $\lambda(n)$.

   (a) Thus, in order to find d, we need to solve for d the equation $de \equiv 1 \bmod \lambda(n)$; d can be computed efficiently by using the extended Euclidean algorithm, since, e and $\lambda(n)$ being coprime, the equation is a form of Bezout Identity, where d is one of the coefficients.

   (b) d is kept secret as the private key exponent.

The public key pk consists of the modulus n and the public exponent e. The private key sk consists of the private exponent d, and also the modulus n. In the orignial RSA paper, the Euler totient function $\phi(n) = (p-1)(q-1)$ is used instead of $\lambda(n)$ for calculating the private exponent d. Both is working fine but in many practical RSA implementations, Carmichael's totient function is used instead of Euler's totient function, since the Carmichael function typically yields a smaller modulus for defining the private exponent.

Suppose that Bob wants to send a secret message to Alice, or verify a message from Alice. If they decide to use RSA, Bob must know Alice's public key to encrypt his secret messages or verify Alice's messages, and Alice must use her private key to decrypt Bob's secret messages or sign her own messages. Alice first have to transmits her public key (n, e) to Bob via a reliable, but not necessarily secret, route. Alice's private key (d) is never distributed.

2.2 RSA - Textbook

Textbook RSA is the simplest cryptographic scheme based on the RSA problem. It consists of two main algorithms: encryption and decryption

1. Encryption
   (a) Input: A message/plaintext m and the public key (n, e)
   (b) Message m is then converted to an integer number and calculated by $c = m^e (\mathrm{mod} n)$
   (c) Output: A ciphertext c.
2. Decryption
   (a) Input: A ciphertext c and private key d associated with the public key (n, e).
   (b) Output: A plaintext $m' = c^d (\mathrm{mod} n)$

The message m after converting to integer must satisfy $m < n$. If it is greater than n then we can not use RSA to encrypt it.

This scheme is not secure in practice for two reasons:

- It is deterministic - encrypting the same message always gives the same ciphertext.

- It does not provide semantic security, and is vulnerable to chosen-plaintext and chosen-ciphertext attacks.

Real-world systems therefore never use textbook RSA; they always wrap it with a padding/encoding scheme.

2.3 The padding scheme for RSA

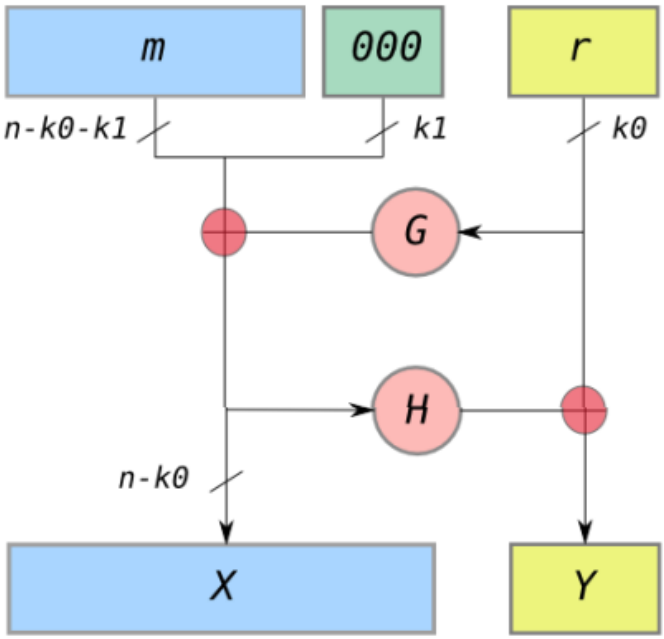
We use the OAEP padding scheme together with the RSA algorithm to improve security, rather than relying on textbook RSA.

Optimal Asymmetric Encryption Padding (OAEP) was invented by Mihir Bellare and Phillip Rogaway in 1994 and enchanted by Don Johnson and Stephen Matyas in 1996. It was standardized as RSAES - OAEP in PKCS#1 Version 2 and lately republished as RFC 2437. OAEP combined with RSA is good at performance and provides good security espcially against adaptive chosen ciphertext attack.

There are two aims of OAEP:

1. Adding random padding to plaintext can convert RSA from a deterministic scheme into a probabilistic one.
2. Prevent leaking any encryption structure information caused by chosen plaintext attack.

The padding process of OAEP is shown as below:



Notation.

- n: the bit-length of the RSA modulus.
- k_0 and k_1: parameters defined by the OAEP scheme.
- m: the plaintext message with length $n - k_0 - k_1$ bits.
- G and H: cryptographic hash functions.
- $\oplus$: the bitwise XOR operation.
- r: a randomly generated bit string of length k_0.

OAEP Encoding.

1. The plaintext m is padded with k_1 zero bits, producing message m' of length $n - k_0$ bits.
2. A random string $r \in \{0,1\}^{k_0}$ is generated.
3. Compute
$$X = m' \oplus G(r).$$
4. Compute
$$Y = r \oplus H(X).$$
5. The OAEP-encoded message is the concatenation (X, Y).

OAEP Decoding.

1. Recover the random string by
$$r = Y \oplus H(X).$$
2. Recover the padded message by
$$m' = X \oplus G(r),$$
and remove the zero padding to obtain the original plaintext m.

Security of OAEP. OAEP provides semantic security against chosen-ciphertext attacks. Although Victor Shoup initially raised concerns regarding whether OAEP could achieve such security, Fujisaki *et al.* proved in 2001 that RSA-OAEP is semantically secure in the random oracle model.

2.4 RSA Signature Scheme

RSA signatures use the same key pair (N, e), (N, d) but reverse the roles of encryption and decryption:

- - Signing (with the private key): $s = m^d \bmod N$
- - Verification (with the public key): check $m \stackrel{?}{=} s^e \bmod N$

In practice, m is not the raw message but a padded/hash-encoded value. Different signature schemes define various ways to encode the hash of a message before applying RSA.

We focus on two main signature schemes:

2.4.1 PKCS#1 v1.5 (RSASSA-PKCS1-v1.5)

PKCS#1 v1.5 has long been the most widely deployed RSA signature scheme in practice. The original definition of the RSASSA-PKCS1-v1_5 algorithm is given in RFC 3447, available at RFC3447.

Algorithm 1 RSA-PKCS#1 v1.5 Signature Scheme

procedure KeyGen(1^λ)
  Choose primes $P, Q \xleftarrow{R} \mathcal{P}(\lambda/2)$
  $N \leftarrow PQ, \quad \varphi(N) \leftarrow (P-1)(Q-1)$
  Choose e such that $\gcd(e, \varphi(N)) = 1$
  $d \leftarrow e^{-1} \bmod \varphi(N)$
  Select hash function $H : \{0,1\}^* \to \{0,1\}^\ell$
  Determine hash identifier ID_H
  return $pk \leftarrow (N, e), \quad sk \leftarrow (N, d)$
end procedure
procedure Sign(sk, m)
  $z \leftarrow H(m)$
  Construct the encoded message y:
  $y \leftarrow \texttt{0x00} \parallel \texttt{0x01} \parallel \texttt{FF} \ldots \texttt{FF} \parallel \texttt{0x00} \parallel \mathsf{ID}_H \parallel z$
  $\sigma \leftarrow y^d \bmod N$
  return σ
end procedure
procedure Verify(pk, m, σ)
  $y' \leftarrow \sigma^e \bmod N$
  $z \leftarrow H(m)$
  if y' has valid PKCS#1 v1.5 format and contains $\mathsf{ID}_H \parallel z$ **then**
    return 1
  else
    return 0
  end if
end procedure

Properties.

- **Deterministic:** signing the same message always produces the same signature.

- **Widely deployed:** PKCS#1 v1.5 signatures are extensively supported in practice, including in TLS, X.509 certificates, code signing infrastructures, and legacy JWT configurations.
- **Sensitive to implementation flaws:** security critically depends on strict and correct verification. Lenient parsing, acceptance of malformed paddings, or improper handling of the ASN.1 `DigestInfo` structure can lead to signature forgery attacks.
- **Limited theoretical guarantees:** due to its deterministic and highly structured padding, PKCS#1 v1.5 signatures are difficult to analyze under tight security reductions.

2.4.2 RSASSA-PSS

RSASSA-PSS is a modern RSA signature scheme standardized in RFC 8017. At a high level, the scheme operates as follows:

1. Given a message M, compute its hash
$$mHash = \text{Hash}(M).$$
2. Generate a random salt and construct an encoded message (EM) using a randomized, mask-based padding scheme. The encoding combines $mHash$, the salt, and fixed bits through a mask generation function (MGF).
3. Interpret the encoded message EM as an integer m_{EM} and compute the signature
$$s = m_{\text{EM}}^d \bmod N.$$
4. For verification, compute
$$m'_{\text{EM}} = s^e \bmod N,$$
and run the EMSA-PSS verification procedure to check whether the recovered encoding is consistent with the message M.

Unlike PKCS#1 v1.5, RSASSA-PSS introduces randomness into the signing process, enabling probabilistic signatures with strong provable security guarantees.

Overall, RSA-PSS represents the modern and recommended RSA signature scheme, offering probabilistic signatures and strong provable security guarantees, in contrast to the deterministic and more fragile PKCS#1 v1.5 construction.

2.5 Summary.

- RSA defines a core mathematical primitive parameterized by (N, e, d); encryption and signature schemes are higher-level constructions built on top of this primitive.
- In practice, the security of RSA-based systems critically depends on the choice of padding and encoding schemes, rather than solely on the hardness of factoring the modulus N.
- **PKCS#1 v1.5 signatures** are deterministic and widely deployed, but are sensitive to implementation flaws, particularly lenient parsing and incomplete padding verification.
- **RSASSA-PSS** introduces randomized, mask-based encoding and offers stronger theoretical security guarantees, and is therefore recommended for modern designs and system migrations.

These constructions and their differences form the foundation for the subsequent analysis of practical attacks on RSA, including padding oracle attacks (e.g., Bleichenbacher-style attacks), timing attacks, and fault attacks, as well as a broader evaluation of deployment-level weaknesses.

2.6 The Factoring Problem in RSA

2.6.1 Context

In our setting, a trusted key-generation algorithm $\mathsf{GenModulus}(1^n)$ outputs a triple (N, p, q), where p and q are independently and uniformly sampled n-bit prime numbers and $N = pq$.

The RSA public key contains the modulus N (together with a public exponent e), while the prime factors p and q are kept secret and constitute the core of the private key.

In practice, such a modulus N is embedded in various real-world systems, including:

1. server certificates used in TLS connections (e.g., for online payments or secure APIs),
2. digital-signature keys used to sign transactions, contracts, or software updates,
3. hardware tokens or hardware security modules (HSMs) operated by service providers.

An external adversary $\mathcal{A}$ can freely observe N, as it is public by design, and may obtain arbitrarily many ciphertexts or signatures generated under this modulus. The most direct way for $\mathcal{A}$ to break RSA is therefore to solve the *factoring problem* for N, that is, to find non-trivial factors $p, q > 1$ such that $N = pq$.

The *factoring assumption* states that for moduli generated by GenModulus, no efficient (polynomial-time) adversary can factor N with non-negligible probability in the security parameter n.

2.6.2 Risks

If the factoring assumption fails for a deployed RSA modulus, the consequences for the system are severe:

- **Private-key recovery:** once $\mathcal{A}$ computes p and q, it can derive $\varphi(N)$ and recover the private exponent d, thereby obtaining a full copy of the RSA private key.
- **Loss of confidentiality:** the adversary can decrypt any ciphertext encrypted under the public key (N, e), including previously recorded traffic, leading to retrospective disclosure of sensitive data such as credentials, financial information, or session keys.
- **Loss of integrity, authenticity, and non-repudiation:** with access to the private key, the adversary can forge valid RSA signatures that

are indistinguishable from those produced by the legitimate key owner. This enables server impersonation, fraudulent transaction approvals, or the signing of malicious software updates.

- **System-wide impact due to key reuse:** in many deployments, a single RSA key pair is reused across multiple services (e.g., TLS, VPN authentication, or code signing). Factoring a single modulus N may therefore compromise multiple security domains simultaneously.

In summary, breaking the factoring assumption directly compromises the security of any RSA-based encryption or signature scheme built on the affected modulus.

2.6.3 Goals

To rely on RSA securely, system design must ensure that the factoring assumption remains realistic for all deployed moduli.

Goal 1: Hard-to-factor moduli (sufficient key sizes). The modulus size N should be large enough that the best known classical factoring algorithms, such as the Number Field Sieve, remain computationally infeasible in practice. Current recommendations require at least 2048-bit moduli, with 3072-bit or larger moduli preferred for long-term security.

Goal 2: High-quality modulus generation. The algorithm GenModulus must use cryptographically secure randomness to generate primes p and q that are:

- of appropriate size,
- not too close to each other,
- not drawn from small or structured subsets.

This prevents the creation of weak RSA keys that may be vulnerable to specialized factoring attacks, such as shared-prime or low-entropy key attacks.

Goal 3: Key isolation and minimal reuse. RSA moduli should not be reused across independent security domains (e.g., combining TLS, code signing, and document signing under a single key). If a modulus is compromised, the resulting damage should be contained.

Goal 4: Forward-looking protection against advances in factoring. System operators must monitor cryptanalytic progress and adjust key sizes and lifetimes accordingly. In addition, migration plans toward post-quantum cryptographic schemes should be considered, given the potential impact of quantum algorithms such as Shor's algorithm on integer factorization.

Together, these goals formalize what it means, at the system level, to assume that factoring is hard: parameters, algorithms, and operational practices must be chosen so that any realistic adversary's probability of factoring N (as produced by GenModulus) remains negligible.

2.7 Proposed Solution

In this project, we propose two complementary approaches for generating RSA parameters and for performing RSA-based encryption and digital signature operations on messages or files.

The first approach is based on the Python `PyCryptodome` library, which provides a high-level and well-maintained implementation of modern cryptographic primitives. The second approach relies on `OpenSSL` version 3.6.0, which is the latest stable release at the time of writing. In this case, OpenSSL command-line utilities are wrapped using Python 3 through the `subprocess` and `argparse` modules, allowing programmatic control while relying on a widely deployed reference implementation.

For key generation, the default configuration generates two primes p and q of 1536 bits each, resulting in an RSA modulus of 3072 bits. This key size is chosen to provide a strong security margin against current classical factoring algorithms.

For encryption, the default scheme is RSA-OAEP, which ensures probabilistic encryption and semantic security for plaintexts and ciphertexts. For digital signatures, we recommend the use of RSASSA-PSS, as it provides randomized signatures and stronger theoretical security guarantees compared to legacy schemes.

All demonstration code corresponding to key generation, encryption, and signature operations can be found at: GitHub/KeyGeneration.

3 Attack models

3.1 Cryptanalysis Tools

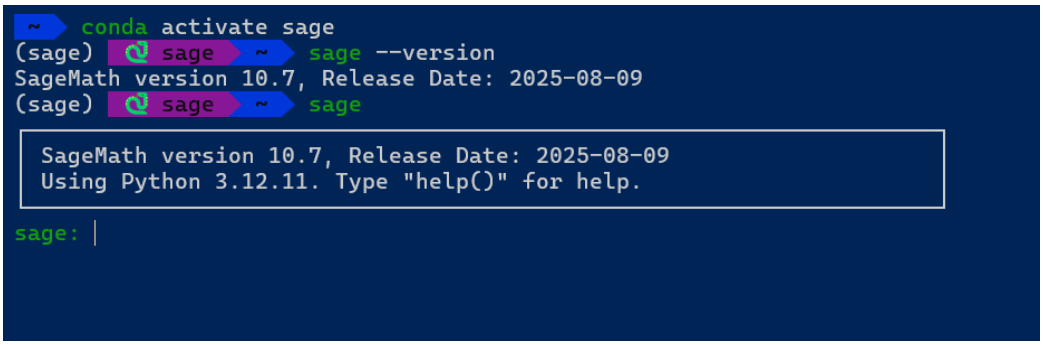
To support the implementation and analysis of RSA cryptographic attacks, we use the following tools and environments.

- **Python 3.x**
  Python is used as the primary scripting language for implementing cryptographic algorithms, experiments, and automation scripts.
  Installation guide: python.org/downloads

- **SageMath**
  SageMath is used for symbolic computation and number-theoretic operations required in cryptanalysis, such as lattice-based attacks, polynomial manipulation, and modular arithmetic.
  Installation guide: SageMath Installation Guide. For convenience, one should install SageMath via `conda-forge`.

When using Visual Studio Code, it is important to select the correct Python interpreter corresponding to the SageMath environment. This can be done by pressing `Ctrl+Shift+P` and choosing the appropriate interpreter before writing or executing code.

Acivate the Sage cell by the following command:



In this project we will be using `Python` version 3.12 and `SageMath` version 10.7

3.2 Factoring Attacks

In the factoring attack model, the adversary is given only the public RSA key (N, e) and attempts to break RSA by factoring the modulus

$$N = pq.$$

If the adversary successfully recovers the two prime factors p and q, it can:

- compute $\varphi(N) = (p-1)(q-1)$,
- derive the private exponent

$$d = e^{-1} \bmod \varphi(N),$$

- and thus obtain the full private key (N, d).

At this point, RSA is completely compromised: the adversary can decrypt any ciphertext c by computing $m = c^d \bmod N$, and can forge valid signatures $s = m^d \bmod N$ that verify under the public key (N, e). This directly violates the *factoring assumption* underlying the GenModulus experiment.

In practice, factoring large integers is performed using specialized integer-factorization algorithms. For general-purpose, randomly generated RSA moduli, the fastest known classical algorithm is the *General Number Field Sieve (GNFS)*. In this project, we treat GNFS as a conceptual black-box factoring oracle:

- **Input:** the RSA modulus N,
- **Output:** (if successful) two primes p, q such that $N = pq$.

Practical feasibility. With current classical computing technology, factoring a well-generated RSA modulus of 3072 bits is considered computationally infeasible. No known classical algorithm can factor such a modulus in polynomial or even practical super-polynomial time. As a result, factoring-based attacks are not considered viable against correctly generated RSA-3072 keys in real-world deployments.

Factoring attacks become feasible primarily when RSA parameters fail to satisfy the security requirements of GenModulus, for example:

- the modulus N is too small (e.g., toy key sizes used for testing or laboratory exercises),
- the key-generation process is flawed (e.g., insufficient randomness, repeated primes, or special structure in p or q).

Quantum considerations. From a long-term perspective, large-scale quantum computers running Shor's algorithm would be able to factor RSA moduli efficiently. However, despite significant progress in quantum computing research, such machines are not yet available at the required scale. Current expert consensus suggests that cryptographically relevant quantum computers capable of breaking RSA-3072 are still many years away, and possibly decades from practical deployment.

Nevertheless, this anticipated threat motivates ongoing research into post-quantum cryptography and long-term migration strategies away from factoring-based public-key schemes.

Attack outline. Given the public key (N, e), a factoring-based attack proceeds as follows:

1. Run a factoring algorithm (conceptually, GNFS) on N to obtain p and q such that $N = pq$.
2. Compute $\varphi(N) = (p-1)(q-1)$.
3. Compute the private exponent $d = e^{-1} \bmod \varphi(N)$.
4. Use (N, d) to:
   - decrypt ciphertexts c as $m = c^d \bmod N$,
   - or forge signatures $s = m^d \bmod N$.

3.3 Wiener's Attack

Wiener's Theorem. Let $n = pq$ with $q < p < 2q$, and let the private exponent satisfy

$$d < \frac{1}{3} n^{1/4}.$$

Given the public key (n, e) such that $ed \equiv 1 \pmod{\varphi(n)}$, the private exponent d can be efficiently recovered.

Attack Description. Assume that the attacker is given the RSA public key (n, e). Wiener's attack attempts to recover the private exponent d when it is unusually small.

Algorithm 2 Wiener's Attack on RSA with Small Private Exponent

Require: RSA public key (n, e)
Ensure: Private exponent d, if vulnerable

Compute the continued fraction expansion of $\frac{e}{n}$:

$$\frac{e}{n} = [a_0; a_1, a_2, \ldots, a_k].$$

for each convergent $\frac{k_i}{d_i}$ of the continued fraction **do**
    if d_i is even **then**
        continue
    end if
    if $ed_i \not\equiv 1 \pmod{k_i}$ **then**
        continue
    end if
    Compute $\varphi(n) \leftarrow \frac{ed_i - 1}{k_i}$
    Solve the quadratic equation

$$x^2 - (n - \varphi(n) + 1)x + n = 0$$

    if the equation has integer roots **then**
        return $d \leftarrow d_i$
    end if
end for
return failure (RSA parameters are not vulnerable)

3.4 Low-exponent attacks

In many practical environments, RSA encryption is performed on resource-constrained devices such as smart cards or embedded systems. In such settings, exponentiation with a large public exponent e may be costly in terms of computation time, energy consumption, or hardware complexity.

To optimize performance, it is tempting to choose a very small public exponent, such as $e = 3$. With this choice, encryption reduces to computing

$$c = m^3 \bmod N,$$

which requires only two modular multiplications.

At first glance, this modification appears harmless, as computing modular cube roots without knowing the factorization of N still seems difficult. However, as we show below, using a low public exponent can lead to severe vulnerabilities in certain communication scenarios.

Lemma (Chinese Remainder Theorem). Let $m_1, \ldots, m_r$ be pairwise coprime integers, and suppose we are given a system of congruences

$$x \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, r.$$

Then there exists a unique solution x modulo $M = m_1 m_2 \cdots m_r$, and this solution can be computed efficiently.

Suppose Alice wishes to send the same plaintext message m to three different recipients: Bob, Charlie, and Dave. Each recipient has an RSA public key with public exponent $e = 3$:

$$(N_B, 3), \quad (N_C, 3), \quad (N_D, 3).$$

Alice encrypts the same message m under each public key, producing the ciphertexts:

$$c_B = m^3 \bmod N_B, \quad c_C = m^3 \bmod N_C, \quad c_D = m^3 \bmod N_D.$$

We assume without loss of generality that N_B, N_C, and N_D are pairwise coprime. (If this is not the case, an attacker can immediately factor at least one modulus and recover the plaintext.)

An eavesdropper who intercepts (c_B, c_C, c_D) can apply the Chinese Remainder Theorem to compute a value c such that

$$c \equiv m^3 \pmod{N_B N_C N_D}.$$

Since the plaintext satisfies $m < N_B, N_C, N_D$, it follows that

$$m^3 < N_B N_C N_D.$$

Therefore, the congruence above holds as an equality over the integers:

$$c = m^3.$$

The attacker can then recover the original message m by computing the integer cube root of c.

3.4.1 Discussion and Limitations

This attack demonstrates that RSA with a low public exponent is vulnerable when the same message is encrypted under multiple public keys without proper randomization or padding.

One might attempt to mitigate this issue by ensuring that different recipients never receive the same plaintext. For example, each recipient could be assigned a unique identifier ID, and the sender could encrypt

$$(m + 2^k \cdot \mathsf{ID})^3 \bmod N,$$

where k is the bit-length of m.

However, as shown in subsequent sections, this countermeasure is insufficient. In fact, a much more general class of attacks exists, known as *Håstad's Broadcast Attack*, which applies whenever RSA encrypts a low-degree polynomial function of the message under multiple moduli.

In the example above, the encryption function corresponds to the polynomial

$$g(m) = (m + 2^k \cdot \mathsf{ID})^3,$$

which still allows efficient recovery of m under suitable conditions.

3.4.2 Coppersmith's Theorem

Many of the most powerful attacks against RSA with a low public exponent rely on a fundamental result due to Coppersmith. Coppersmith's theorem has numerous applications in cryptanalysis; in this project, we focus on its role in breaking RSA when partial information about the plaintext is known or when the plaintext satisfies a low-degree polynomial relation modulo the RSA modulus.

Theorem (Coppersmith). Let N be an integer and let $f(x) \in Z[x]$ be a monic polynomial of degree d. Fix a parameter $\varepsilon > 0$ and define

$$X = N^{\frac{1}{d} - \varepsilon}.$$

Given $\langle N, f \rangle$, there exists an efficient algorithm that finds all integers x_0 satisfying

$$|x_0| < X \quad \text{and} \quad f(x_0) \equiv 0 \pmod{N}.$$

The running time of the algorithm is dominated by the execution of the LLL lattice basis reduction algorithm on a lattice of dimension

$$w = \min\bigl(1/\varepsilon, \log_2 N\bigr).$$

Interpretation. Coppersmith's theorem provides an explicit algorithm for finding all sufficiently small roots of a polynomial congruence modulo a composite integer N. In particular, when ε is small, the algorithm can recover all roots x_0 with

$$|x_0| < N^{1/d}.$$

As the bound on $|x_0|$ decreases, the lattice dimension required by the algorithm becomes smaller, leading to improved practical performance.

Relevance to RSA. The strength of Coppersmith's theorem lies in its applicability to composite moduli. When working modulo a prime, classical root-finding algorithms are typically more efficient, and Coppersmith's method is unnecessary.

In the context of RSA, Coppersmith's theorem enables attacks in which the plaintext (or part of it) is small or satisfies a low-degree polynomial equation modulo N. This includes, for example:

- low public exponent attacks,
- Håstad's broadcast attack,
- partial key exposure attacks.

These attacks do not require factoring N and instead exploit algebraic structure combined with lattice-based techniques.

3.5 Fault attacks on RSA-CRT

To sign a message m, the signer first applies an encoding function μ to m, and then computes the signature $\sigma = \mu(m)^d \bmod N$. To verify the signature σ, the receiver check that $\sigma^e = \mu(m) \mod N$. The Chinese Remainder Theorem (CRT) is often used to speed up signature generationby a factor of about 4. This is done by computing:

$$\sigma_p = \mu(m)^{d \bmod p-1} \bmod p,\ \sigma_q = \mu(m)^{d \bmod q-1} \bmod q$$

and deriving σ from (σ_p, σ_q) using the CRT.

The first attack on this scheme is Fault attacks discovered by Boneh, DeMillo and Lipton in 1997. Assuming that the attacker can induce a fault when σ_q is computed while keeping the computation of σ_p correct, one get:

$$\sigma_p = \mu(m)^{d \bmod p-1} \bmod p,\ \sigma_q \neq \mu(m)^{d \bmod q-1} \bmod q$$

hence:

$$\sigma^e = \mu(m) \mod p,\ \sigma^e \neq \mu(m) \mod q$$

which allows the attacker to factor N by computing

$$\gcd\left(\sigma^e - \mu(m) \mod N, N\right) = p$$

.

This attack applies to any deterministic padding function µ, such as RSA PKCS#1 v1.5 or Full-Domain Hash, or probabilisticsignatures where the randomizer used to generate the signature is sent along with the signature, such as PFDH . Only probabilistic signature schemes such that the randomness remains unknown to the attacker may be safe, though some particular cases have been attacked as well.

3.5.1 Inject on the modulus

An alternative way to mount fault attacks on RSA-CRT is to inject faults into the public modulus, instead of targeting one of the two CRT sub-exponentiations. In this section, we demonstrate such an attack using the orthogonal lattices technique introduced by Nguyen and Stern.

Let the signer compute a signature σ on a message m padded as $\mu(m)$. First, the signer computes the CRT components

$$\sigma_p \equiv \mu(m)^{\,d \bmod (p-1)} \pmod p, \qquad \sigma_q \equiv \mu(m)^{\,d \bmod (q-1)} \pmod q,$$

and then recombines them to output

$$\sigma \equiv \sigma_p \alpha + \sigma_q \beta \pmod N,$$

where

$$\alpha = q\,(q^{-1} \bmod p), \qquad \beta = p\,(p^{-1} \bmod q).$$

Assume that an adversary can obtain the correct signature σ, and also a faulty signature σ' on the same padded message $\mu(m)$ after corrupting the

modulus N into a faulty modulus $N' \neq N$ right before the CRT recombination step. Thus, the faulty output satisfies

$$\sigma' \equiv \sigma_p \alpha + \sigma_q \beta \pmod{N'}.$$

For the sake of our experiment, we further assume that the adversary is able to recover the faulty modulus N'. Then, by applying CRT to the pair (σ, σ'), the adversary can compute an integer

$$v = \mathrm{CRT}_{N,\,N'}(\sigma, \sigma') \in Z,$$

which is congruent to $\sigma_p \alpha + \sigma_q \beta$ modulo NN'.

Since $|N| \approx 2^n$, we have $|NN'| \approx 2^{2n}$, whereas α, β are n-bit values and σ_p, σ_q are roughly $(n/2)$-bit values. Hence, v can be viewed (heuristically) as a *small* integer linear combination of α and β. Based on this observation, we will construct an orthogonal lattice to recover a nontrivial factor of N.

Attack Outline for the attacks: Assume that, for $\ell \geq 5$ padded messages $\mu(m_i)$, the adversary obtains a correct signature σ_i and a faulty signature σ_i' computed with a faulty modulus $N_i' \neq N$ (and that N_i' is known/recovered). Then, one can heuristically recover the factorization of N as follows.

1. **CRT lifting.** For each i, compute

   $$v_i = \mathrm{CRT}_{N,\,N_i'}(\sigma_i, \sigma_i') \in Z.$$

   Let $\mathbf{v} = (v_1, \ldots, v_\ell) \in Z^\ell$.

2. **First orthogonal lattice.** Compute an LLL-reduced basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_{\ell-1}\}$ of the orthogonal lattice

   $$\mathbf{v}^\perp = \big\{\mathbf{b} \in Z^\ell : \langle \mathbf{b}, \mathbf{v} \rangle = 0\big\}.$$

   In practice, this can be done by applying LLL to an embedding lattice in $Z^{\ell+1}$. Concretely, build the lattice generated by the rows of

   $$\begin{pmatrix} \kappa v_1 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ \kappa v_\ell & 0 & \cdots & 0 & 1 \end{pmatrix},$$

   where κ is a sufficiently large constant, run LLL, and discard the first coordinate of the resulting short vectors to obtain vectors in $\mathbf{v}^\perp$.

3. **Second orthogonal lattice (rank 2).** Let $L' \subset Z^\ell$ be the sublattice generated by the first $\ell - 2$ vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{\ell-2}$ (so $\text{rank}(L') = \ell - 2$). Compute an LLL-reduced basis $\{\mathbf{x}', \mathbf{y}'\}$ of the rank-2 lattice $(L')^\perp$. Again, this can be achieved by applying LLL to an embedding lattice in $Z^{\ell+2}$ generated by the rows of

$$\begin{pmatrix} \kappa' b_{1,1} & \cdots & \kappa' b_{1,\ell-2} & 1 & 0 \\ \vdots & & \vdots & \vdots & \vdots \\ \kappa' b_{\ell,1} & \cdots & \kappa' b_{\ell,\ell-2} & 0 & 1 \end{pmatrix},$$

where κ' is a suitably large constant. After LLL, keep the last ℓ coordinates of the short vectors to obtain $\mathbf{x}', \mathbf{y}' \in (L')^\perp$.

4. **Enumeration of short vectors.** Enumerate vectors of the form

$$\mathbf{z} = a\mathbf{x}' + b\mathbf{y}' \in (L')^\perp$$

with Euclidean norm bounded by $\|\mathbf{z}\| \leq \sqrt{\ell N}$.

5. **GCD extraction.** For each enumerated $\mathbf{z}$, try to extract a nontrivial factor of N by computing

$$\gcd(v_i - z_i,\ N)$$

over the components $i = 1, \ldots, \ell$, and output any nontrivial gcd found (which reveals a factor of N).

3.5.2 Countermeasures and Further Research

A first line of defense is to prevent the adversary from obtaining *two* signatures on the *same padded message* $\mu(m)$. In particular, *probabilistic* and *stateful* signature schemes are usually secure against this attack because they make repeated signatures on the same padded value hard to obtain. In contrast, *deterministic* schemes are typically vulnerable, including settings where the attacker does not fully know $\mu(m)$, as long as the target device can be forced to compute the same signature twice.

A natural algorithmic countermeasure is to use a CRT recombination method that does *not* require the public modulus N during interpolation. One common choice is *Garner's formula*, which recombines directly from

(σ_p, σ_q) using only p, q and the CRT coefficient $\gamma = q^{-1} \bmod p$. Assuming $p > q$, Garner recombination can be written as:

$t \leftarrow \sigma_p - \sigma_q$
if $t < 0$ **then** $t \leftarrow t + p$
$\sigma \leftarrow \sigma_q + (t \cdot \gamma \bmod p) \cdot q$
return σ

Moreover, the final modular reduction can be avoided since:

$$\sigma = \sigma_q + (t \cdot \gamma \bmod p) \cdot q \leq (q-1) + (p-1)q < N. \quad (\text{hence } \sigma < N)$$

This removes the explicit dependence on N in the recombination step and thwarts the modulus-fault attack described earlier.

Besides switching the recombination formula, a practical defense is to *check signatures before release* (e.g., verify $\sigma^e \equiv \mu(m) \pmod N$ internally, or use an infective variant that randomizes the output upon detection), Although this may incur extra cost.

Further research. Beyond these generic defenses, it would be valuable to devise countermeasures that specifically protect the CRT recombination step (either Formula (1) or Garner's formula) while taking into account that *all* intermediate values involved (e.g., $\sigma_p, \sigma_q, p, q, \gamma$) may be corrupted by faults. Finally, in special cases and particular implementation settings, other fault attacks targeting CRT recombination may exist; a systematic analysis of such scenarios remains an interesting direction.

3.6 Timing attack

The core idea is that many modular exponentiation routines (e.g., square-and-multiply or sliding-window) execute a data-dependent sequence of operations: a squaring is performed at every step, while an extra multiplication is performed only when the processed bit/window is non-zero. If the implementation is not constant-time, this conditional behavior can create a measurable timing bias that correlates with the secret exponent bits.

Assume we can query the device on many inputs and record a dataset of pairs

$$(\mu(m_i),\ \sigma_i,\ T_i),$$

where T_i is the measured signing (or exponentiation) time, $\mu(m_i)$ is the padded message and σ_i is the signature of our message . The attack proceeds *bit by bit* (more precisely, prefix by prefix) as follows.

Let $d_{[t]}$ denote the already recovered prefix of the secret exponent up to bit position $t-1$. To decide the next bit $b \in \{0,1\}$, the attacker tests two hypotheses for the extended prefix $d_{[t+1]} = d_{[t]} \| b$. For each hypothesis, the attacker partitions the recorded samples into two sets: a set that is predicted to execute the "slower" branch at the next step, and a set predicted to execute the "faster" branch (based on the assumed prefix).

For each hypothesis b, the attacker computes the difference of average timings

$$\Delta_b = \overline{T}_{\text{slow},b} - \overline{T}_{\text{fast},b}.$$

If the hypothesis is correct, the partition tends to separate samples according to the real execution path, and Δ_b becomes noticeably positive (above a chosen threshold). If the hypothesis is wrong, the partition mixes both behaviors, and the timing gap largely cancels out, yielding a small Δ_b. The attacker then selects the bit b that produces the most significant gap (or the only gap exceeding the threshold), appends it to the known prefix, and repeats the procedure for the next bit.

This procedure is essentially a statistical classification step repeated iteratively: each recovered bit is the one that maximizes the consistency between the predicted fast/slow behavior and the observed timing measurements.

3.6.1 Countermeasures against timing attacks

Although timing attacks are a serious threat, there are several practical countermeasures that can be deployed in RSA implementations. Below we summarize three common approaches.

1) Constant exponentiation time (constant-time implementation). The most robust mitigation is to ensure that the private-key operation (modular exponentiation) runs in *constant time*, i.e., its control flow and memory accesses do not depend on secret data. Conceptually, this can be achieved by making every iteration execute the same sequence of operations (e.g., using a ladder-style exponentiation), and by avoiding secret-dependent table lookups in window methods. This is simple to state, but it can degrade performance compared to variable-time implementations.

2) Random delay (noise injection). A lightweight mitigation is to add a random delay before returning the result in order to decorrelate the observed runtime from the secret-dependent behavior. However, this is not a principled defense: if the added noise is not large enough, an attacker can average it out by collecting more measurements. Hence, random delays are generally considered weaker than constant-time techniques.

3) Blinding. Blinding is one of the most effective countermeasures in practice. The idea is to randomize the input to the private-key operation so that an attacker cannot meaningfully correlate timing variations with the structure of the *original* ciphertext/message.

For RSA decryption/signing, where we compute

$$M \equiv C^d \pmod{n},$$

message blinding can be implemented as follows:

1. Sample a secret random $r \in \{1, \ldots, n-1\}$ such that $\gcd(r, n) = 1$.
2. Compute the blinded ciphertext
$$C' \equiv C \cdot r^e \pmod{n},$$
where e is the public exponent.
3. Compute the blinded plaintext/signature using the ordinary private-key operation:
$$M' \equiv (C')^d \pmod{n}.$$
4. Unblind:
$$M \equiv M' \cdot r^{-1} \pmod{n},$$
where r^{-1} denotes the multiplicative inverse of r modulo n.

Correctness follows from $M' \equiv (C \cdot r^e)^d \equiv C^d \cdot r^{ed} \pmod{n}$, and since $ed \equiv 1 \pmod{\varphi(n)}$, we have $r^{ed} \equiv r \pmod{n}$, hence $M \equiv C^d \pmod{n}$.

In practice, enabling blinding typically incurs a modest performance overhead (often reported on the order of a few percent, depending on implementation and platform).

Remark. In a production-grade RSA library, blinding is usually combined with a constant-time exponentiation routine (and, for RSA-CRT, integrity checks on CRT recombination) to defend simultaneously against timing attacks and fault attacks.

4 Implementation and Testing

We use SageMath together with Python/C++ to implement and demonstrate the attacks (server/client scripts) and RSA primitives. All source code and experiment logs are available at github/ken3k06/RSA.

Software environment. All experiments were conducted in a controlled environment:

- OS: `Ubuntu 24.04.3 LTS (WSL2)`
- Python: `3.12`
- SageMath: `10.7`
- OpenSSL: `3.6.0`

5 Conclusion

We categorized attacks on RSA into four categories: (1) elementary attacks that exploit blatant misuse of the system, (2) low private exponent attacks serious enough that a low private exponent should never be used, (3) low public exponent attacks, (4) and attacks on the implementation. Although RSA has been studied extensively, there is no devastating attack has ever been found. The attacks discovered so far mainly illustrate the pitfalls to be avoided when implementing RSA. At the moment, it appears that proper implementations can be trusted to provide security in the digital world.

References

[1] Dan Boneh. Twenty years of attacks on the rsa cryptosystem. https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf, 1999.

[2] Eric Brier, David Naccache, Phong Q. Nguyen, and Mehdi Tibouchi. Modulus fault attacks against RSA-CRT signatures. Cryptology ePrint Archive, Paper 2011/388, 2011.

[3] Nairen Cao, Adam O'Neill, and Mohammad Zaheri. Toward RSA-OAEP without random oracles. Cryptology ePrint Archive, Paper 2018/1170, 2018.

[4] Michael Perry. Notes on bleichenbacher's attack. https://fog.misty.com/perry/Bleichenbacher/notes.html.

[5] Wing H. Wong. Timing attacks on rsa: Revealing your secrets through the fourth dimension. https://www.cs.sjsu.edu/faculty/stamp/students/article.html, 2005.