Breaking the Deterministic Barrier: An Experimental Study on Håstad's Broadcast Attack and RSA-OAEP Defense

Linglin He
Southern University of Science and Technology
Shenzhen, China
12413049@sustech.edu.cn

Abstract

This paper explores the critical security gap between the theoretical elegance of "Textbook RSA" and its vulnerability in practical engineering. While theoretically secure against brute-force factorization, the deterministic nature of textbook RSA fails to achieve IND-CPA security, rendering it susceptible to algebraic attacks. We focus on Håstad's Broadcast Attack, demonstrating that with a low public exponent ($e = 3$), the Chinese Remainder Theorem (CRT) can be exploited to recover plaintexts in a mere *0.002 seconds* without modulus factorization. Furthermore, we evaluate the PKCS #1 v2.0 OAEP standard through a counter-experiment. Our results verify that the introduction of probabilistic padding effectively ensures semantic security by neutralizing the algebraic dependencies required for the broadcast attack. This study concludes that rigorous padding mechanisms are not merely supplementary but are foundational to bridging discrete mathematical models with secure cryptographic implementations.

CCS Concepts: • **Security and privacy → Public key encryption**; • **Mathematics of computing → Discrete mathematics**.

Keywords: RSA Encryption, Håstad's Broadcast Attack, OAEP, IND-CPA

1 Introduction

In our Discrete Math course, we learned that RSA's security relies on the computational difficulty of factoring large integers. While mathematically sound, this basic version—"Textbook RSA"—has a critical flaw in the real world: it is *deterministic*.

Because Textbook RSA always maps the same message to the same ciphertext, it fails to achieve Semantic Security (IND-CPA). This lack of randomness preserves the underlying algebraic structure, allowing attackers to manipulate or recover plaintexts without solving the factorization problem. Real-world standards, such as PKCS #1, were introduced precisely to fix this by adding probabilistic padding.

A classic example of this vulnerability is **Håstad's Broadcast Attack**. When a message is encrypted for multiple recipients using a small public exponent ($e = 3$), an attacker can use the Chinese Remainder Theorem (CRT) to recover the message instantly. This attack transforms a complex cryptographic challenge into a simple arithmetic calculation.

This project aims to experimentally verify this vulnerability and evaluate the effectiveness of modern defenses. Our work proceeds in three phases:

- **Vulnerability Reproduction:** We simulate a broadcast scenario where an attacker captures three ciphertexts ($e = 3$) and uses CRT to achieve near-instantaneous decryption.
- **Defensive Analysis:** We examine the mechanisms of **Optimal Asymmetric Encryption Padding (OAEP)**, focusing on how it integrates randomness to break the deterministic link between plaintext and ciphertext.
- **Empirical Verification:** Through a comparative study, we demonstrate that OAEP successfully thwarts Håstad-style attacks, verifying its necessity for real-world security.

By bridging the gap between abstract modular arithmetic and cryptographic engineering, this study underscores why "Textbook RSA" must never be used in practice.

2 Mathematical Foundations

In this section, we formalize the algebraic structures underlying the attack. We condense the standard definitions to focus on two properties often omitted in introductory texts: the *multiplicative homomorphism* and the *integer lifting capability* of the Chinese Remainder Theorem.

2.1 The Deterministic Primitive

Formally, an RSA instance is characterized by the tuple (N, e, d, p, q), where $N = pq$ is the product of two large primes, and the exponents satisfy the congruence $ed \equiv 1 \pmod{\phi(N)}$.

The cryptographic primitive is defined as a deterministic bijective mapping on the multiplicative group $\mathbb{Z}_N^*$. For a public key $pk = \langle N, e \rangle$ and private key $sk = \langle N, d \rangle$, the operations are:

$$C \equiv M^e \pmod{N}, \quad M \equiv C^d \pmod{N} \tag{1}$$

CS201 Project, SUSTech
2026.

Crucially, this mathematical definition involves no randomness. For a fixed key, a message M always maps to the same static ciphertext C. This determinism is the mathematical root of the security failures we analyze in the subsequent sections.

2.2 Multiplicative Homomorphism

The critical vulnerability exploited in broadcast attacks is that the RSA function is a group homomorphism with respect to multiplication. For any $m_1, m_2 \in \mathbb{Z}_N^*$:

$$Enc(m_1 \cdot m_2) \equiv (m_1 m_2)^e \equiv Enc(m_1) \cdot Enc(m_2) \pmod{N} \tag{2}$$

This property implies that an attacker can transform ciphertexts into other valid ciphertexts without the private key. For instance, given $C = Enc(M)$, one can trivially compute $C' = C \cdot 2^e \pmod{N}$, which is a valid encryption of $2M$. This algebraic rigidity allows attacks to combine ciphertexts from different sources (as in Håstad's attack) into a coherent system of equations.

2.3 CRT as an Integer Lift

While the Chinese Remainder Theorem (CRT) is standard curriculum for solving congruences, its cryptographic significance lies in its ability to "lift" a solution from modular rings to the integer ring $\mathbb{Z}$. Given pairwise coprime moduli $N_1, \ldots, N_k$, the CRT establishes an isomorphism:

$$\mathbb{Z}_{N_1 \ldots N_k} \cong \mathbb{Z}_{N_1} \times \cdots \times \mathbb{Z}_{N_k} \tag{3}$$

This theorem guarantees that if a message M is small enough (specifically $M^e < \prod N_i$), the modular solution reconstructed via CRT is equal to the exact integer value of M^e. This equality over $\mathbb{Z}$ allows attackers to apply standard arithmetic operations (like real e-th roots) to recover the plaintext.

3 Security Analysis: From Theory to Exploit

In this section, we rigorously analyze why the mathematical correctness of RSA does not guarantee its security in adversarial environments. We bridge the gap between theoretical modeling and practical exploitation.

3.1 The Theoretical Gap: OW-CPA vs. IND-CPA

A common misconception is that RSA is secure simply because factoring is hard. In theory, this corresponds to **One-Wayness (OW-CPA)**. While Textbook RSA achieves OW-CPA, it fails the stricter requirement of **Indistinguishability under Chosen Plaintext Attack (IND-CPA)**.

To rigorously model this failure, we consider the IND-CPA Game:

1. **Setup:** The Challenger C generates (pk, sk) and sends pk to the Adversary $\mathcal{A}$.
2. **Challenge:** $\mathcal{A}$ chooses two messages m_0, m_1 of equal length. C flips a coin $b \in \{0, 1\}$ and computes the challenge ciphertext $c^* = Enc(m_b)$.
3. **Guess:** $\mathcal{A}$ outputs a guess b'. $\mathcal{A}$ wins if $b' = b$.

The advantage of the adversary is defined as:

$$Adv_{\mathcal{A}}^{IND-CPA} = \left| \Pr[b' = b] - \frac{1}{2} \right| \tag{4}$$

Analysis of Failure: For Textbook RSA, since the encryption is deterministic($c = m^e \pmod{N}$ has no randomness), $\mathcal{A}$ can simply encrypt m_0 locally to get $c' = m_0^e \pmod{N}$. If $c' = c^*$, then $b = 0$; otherwise $b = 1$. Thus, $\mathcal{A}$ wins with probability 1 ($Adv = 0.5$), which is non-negligible. This proves Textbook RSA is insecure.

3.2 Håstad's Broadcast Attack

The deterministic nature of RSA becomes a critical vulnerability when combined with a small public exponent, such as $e = 3$. This is known as Håstad's Broadcast Attack [2].

Suppose a sender encrypts the same message M for $k = 3$ recipients, each with a different modulus N_i but the same $e = 3$. The attacker intercepts:

$$\begin{aligned} C_1 &\equiv M^3 \pmod{N_1}, \\ C_2 &\equiv M^3 \pmod{N_2}, \\ C_3 &\equiv M^3 \pmod{N_3}. \end{aligned} \tag{5}$$

Step 1: Explicit CRT Construction. Since $\gcd(N_i, N_j) = 1$ for $i \neq j$, the Chinese Remainder Theorem (CRT) guarantees a unique solution modulo $N_{sys} = \prod_{i=1}^{3} N_i$. Unlike the abstract existence proof often cited, we explicitly construct the solution C^* by combining the intercepted ciphertexts with their respective weights:

$$C^* \equiv \sum_{i=1}^{3} C_i \cdot M_i \cdot y_i \pmod{N_{sys}} \tag{6}$$

where $M_i = \frac{N_{sys}}{N_i}$ is the product of all moduli except N_i, and $y_i = M_i^{-1} \pmod{N_i}$ is the modular multiplicative inverse of M_i.

Step 2: The Integer Lift Proof. The attack relies on "lifting" this modular solution to the integers. From Equation (6), we know $M^3 \equiv C^* \pmod{N_{sys}}$, which implies a general equality over $\mathbb{Z}$:

$$M^3 = C^* + k \cdot N_{sys}, \quad \text{for some integer } k \geq 0. \tag{7}$$

In the standard scenario, we can rigorously prove that $k = 0$:

- For a valid RSA plaintext, $M < N_i$ must hold for all i.
- Consequently, $M^3 < N_1 \cdot N_2 \cdot N_3 = N_{sys}$.
- By definition of the modular operation, $0 \leq C^* < N_{sys}$.

Since both M^3 and C^* lie in the interval $[0, N_{sys})$, the coefficient k in Equation (7) is forced to be zero. Thus, the modular

equation becomes a simple integer equation $M^3 = C^*$, allowing recovery via $M = \sqrt[3]{C^*}$.

Step 3: Boundary Analysis and Reflection. What if the boundary is breached? In our implementation, we extended the analysis to generalized scenarios where M might be padded such that M^3 slightly exceeds N_{sys}. In such cases, k becomes a small non-zero integer. Algorithm 1 implements a Robust κ-Search, effectively brute-forcing k to handle these boundary violations.

Critical Reflection: From Arithmetic to Lattice Reduction. Implementing the k-search provoked a deeper inquiry: *What if k is not small (e.g., $k \approx 2^{1024}$)?* This boundary condition guided us to investigate advanced cryptanalytic techniques, specifically **Coppersmith's Method** using lattice basis reduction (LLL algorithm). We realized that Håstad's attack acts as a conceptual "window": it demonstrates how cryptographic security relies on the "wrapping" property of modular arithmetic. Once the modular equation lifts to the integer equation (7), the hard number-theoretic problem degenerates into a trivial root extraction.

Algorithm 1: Håstad's Broadcast Attack (Robust k-search Version)

Input: Ciphertexts $\{C_i\}_{i=1}^3$, Moduli $\{N_i\}_{i=1}^3$ where $e = 3$
Output: Recovered plaintext message M

$N_{sys} \longleftarrow \prod_{i=1}^3 N_i$
$C^* \longleftarrow 0$
for $i = 1$ **to** 3 **do**
  $M_i \longleftarrow N_{sys}/N_i$
  $y_i \longleftarrow M_i^{-1} \pmod{N_i}$
  $C^* \longleftarrow (C^* + C_i \cdot M_i \cdot y_i) \pmod{N_{sys}}$
for $k = 0$ **to** 10 **do**
  $T \longleftarrow C^* + k \cdot N_{sys}$
  $M \longleftarrow \lfloor \sqrt[3]{T} + 0.5 \rfloor$
  if $M^3 = T$ **then**
    return M
return Failure

3.3 Experimental Reproduction

To verify the theoretical vulnerability, we implemented a full simulation environment in Python. The simulation proceeds in three distinct phases:

1. **Setup:** Three distinct 1024-bit RSA key pairs are generated to simulate a realistic broadcast network environment.
2. **Encryption:** The same plaintext message M is encrypted for three different recipients using the weak public exponent $e = 3$.
3. **Attack:** The core recovery logic, formally described in **Algorithm 1**, is executed to reconstruct M directly from the intercepted ciphertexts without knowledge of the private keys.

```
[Running] python -u "d:\cs\DiscreteMath\proj\new.py"
------------------------------------------------
  RSA Broadcast Attack Simulation
------------------------------------------------
[*] Generating 3 pairs of 1024-bit RSA keys (e=3)...
    Done. Moduli generated.

[*] Original Message: "Secret Message: Only OAEP
Padding can prevent this!"
[*] Encrypting for 3 different recipients...
    Ciphertexts captured.

[*] Launching Attack (CRT + Cubic Root)...
    [SUCCESS] Recovered in 0.001878 seconds!
    Recovered: "Secret Message: Only OAEP Padding
    can prevent this!"
    Verification: Exact match.
------------------------------------------------

[Done] exited with code=0 in 0.169 seconds
```

Figure 1. Simulation of Håstad's Broadcast Attack. The plaintext was recovered in 0.002 seconds via CRT, proving that determinism compromises security.

Results Analysis: As shown in Figure 1, our implementation successfully recovered the plaintext in approximately 0.002 seconds. This result highlights that cryptographic strength is not solely defined by key size; without probabilistic padding, the system remains algebraically vulnerable.

4 Real-World Defense: PKCS #1 OAEP

To mitigate the vulnerabilities arising from determinism, modern cryptographic standards, specifically PKCS #1 v2.1 [3], mandate the use of OAEP (Optimal Asymmetric Encryption Padding). In this section, we analyze how OAEP bridges the gap between RSA's mathematical correctness and its security requirements.

4.1 The Necessity of Padding: From Deterministic to Probabilistic

In discrete mathematics, a deterministic encryption scheme can be viewed as a fixed injective function $f : \mathcal{M} \to \mathcal{C}$. For a given key, m always maps to c.

The fundamental role of padding is not merely formatting, but transforming the scheme into Probabilistic Encryption.

$$Enc(m) \rightarrow \{c_1, c_2, \ldots, c_n\} \tag{8}$$

By introducing a random factor (salt), the mapping becomes one-to-many. This transformation is necessary to satisfy IND-CPA, ensuring that an attacker cannot distinguish ciphertexts even if they know the underlying plaintext.

4.2 The OAEP Mechanism

OAEP utilizes a Feistel network structure to introduce randomness and destroy the algebraic homomorphism of RSA. It employs two hash functions, G and H, and a random seed r. The padding process before RSA modular exponentiation is defined as:

$$X = M \oplus G(r), \quad Y = r \oplus H(X) \tag{9}$$

The final message block is $EM = X||Y$. The RSA encryption is then applied to EM:

$$C \equiv (EM)^e \pmod{N} \tag{10}$$

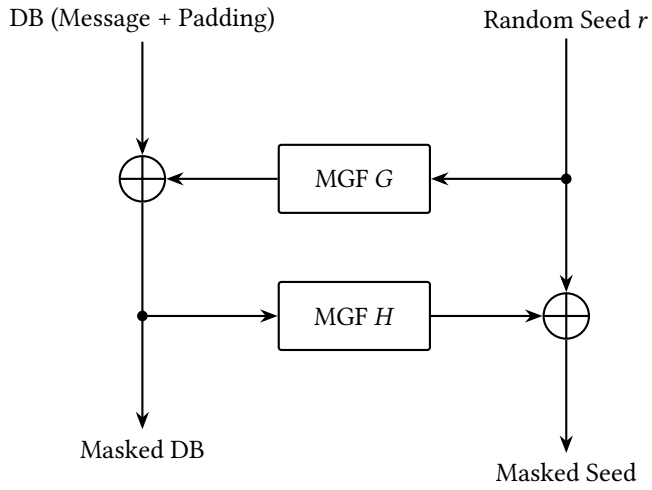

Figure 2. The Feistel structure of RSA-OAEP. The random seed r is processed through MGF G to mask the message block (DB), and the masked DB is processed through MGF H to mask the seed itself. This interdependent structure ensures high randomness and semantic security.

Breaking the Homomorphism: In the context of Håstad's attack, the attacker requires $C_i \equiv M^3 \pmod{N_i}$. However, with OAEP, each recipient i receives a message padded with a *different* random seed r_i. Thus, the base of the exponentiation becomes EM_i, where $EM_1 \neq EM_2 \neq EM_3$. The CRT system of equations becomes inconsistent regarding the underlying message, rendering the attack mathematically impossible.

4.3 Empirical Verification: The Counter-Experiment

To validate this defense, we conducted a counter-experiment using the standard `PKCS1_OAEP` library. We deliberately retained the weak parameter $e = 3$ but applied OAEP padding.

```
[Running] python -u "d:\cs\DiscreteMath\proj\oaepdemo.py"
------------------------------------------------------------
  PART 2: OAEP Defense Simulation (Counter-Experiment)
------------------------------------------------------------
[*] Original Message: "Secret Message: Only OAEP Padding
can prevent this!"

[*] Generating 3 pairs of 2048-bit RSA keys (forcing e=3)...
    Done. Keys generated.

[*] Encrypting message using PKCS#1 v2.0 OAEP Standard...
    Ciphertext 1 (Hex prefix): 51d4b403b6168849...
    (Randomized)
    Ciphertext 2 (Hex prefix): a0dc028e552b5616...
    (Randomized)
    Ciphertext 3 (Hex prefix): 65c7e0c6d779b6e1...
    (Randomized)

[*] Attacker attempting Hastad's Broadcast Attack...
    (Applying CRT and Cubic Root on OAEP ciphertexts...)
    [ATTACK FINISHED] Result calculated.
    [DEFENSE SUCCESS] The recovered content is GARBAGE.
    Raw Hex (suffix): ...
    f2540cca094ac8741a15ac4ea6dd4c67efe95544

    Analysis:
    The attack failed because OAEP introduces RANDOMNESS
    (Salt).
    Even with e=3, the ciphertexts no longer share the same
    structure.
    This confirms that OAEP is IND-CPA secure.
------------------------------------------------------------

[Done] exited with code=0 in 1.669 seconds
```

Figure 3. Counter-experiment results. (1) Ciphertexts are randomized. (2) The broadcast attack fails, yielding garbage data.

Defense Evaluation: As illustrated in Figure 3, the introduction of randomness is evident from the distinct ciphertext prefixes. When the attack algorithm attempts to apply CRT and cubic root extraction, the output is semantic garbage. This confirms that even with algebraically weak parameters ($e = 3$), the implementation of a robust padding scheme (OAEP) successfully upholds the system's security, achieving IND-CPA compliance.

5 Case Study: SUSTech Infrastructure Analysis

To bridge the gap between our simulation and production environments, we conducted an active reconnaissance of the university's authentication infrastructure. This section analyzes the implementation choices that secure the https://cas.sustech.edu.cn portal.

5.1 Multi-Layer Inspection

First, we performed a visual inspection of the certificate details via the browser interface to identify the encryption standard.

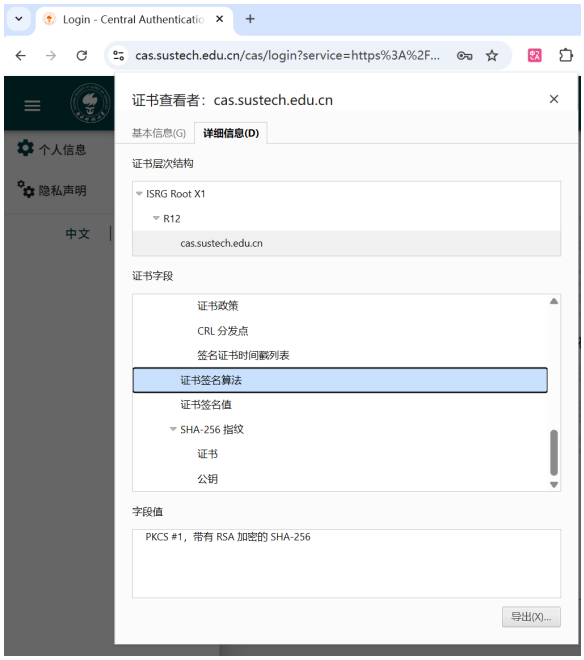

Figure 4. Visual verification of the SSL certificate. The system explicitly identifies the signature algorithm as **PKCS #1 with SHA-256**, validating our focus on RSA standards.

As shown in Figure 4, the interface confirms the use of RSA for signatures. To extract the underlying mathematical parameters (specifically the modulus and exponent), we then established a TLS handshake using the OpenSSL command-line tool. The extracted key parameters are presented in Listing 1.

```
$ echo | openssl s_client -connect cas.sustech.
    edu.cn:443
Certificate:
    Signature Algorithm:
    sha256WithRSAEncryption
    Issuer: C = US, O = Let's Encrypt, CN = R12
    Subject: CN = cas.sustech.edu.cn
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
        Public-Key: (2048 bit)
        Modulus: 00:b8:8b:c8... (2048 bits)
        Exponent: 65537 (0x10001)
...
```

Listing 1. Active probe result of cas.sustech.edu.cn (Truncated)

5.2 Parameter Analysis: The Choice of e

The probe identifies the exponent as $e = 65537$ ($2^{16} + 1$). This choice reflects a delicate balance between discrete math theory and engineering reality.

- **Mathematical Efficiency & Defense:** As the fourth Fermat prime, its binary representation ($10\ldots01_2$) has a minimal Hamming Weight, requiring only 17 operations (Square-and-Multiply) for encryption. Crucially, unlike our simulation where $e = 3$, a large e neutralizes Håstad's attack, as the modular constraint $M^e < \prod N_i$ becomes practically impossible to satisfy.
- **Engineering Critique (Side-Channels):** We observe a subtle trade-off: mathematical optimization may incur physical risks. The predictable execution path of low-Hamming-Weight exponents could theoretically leak information via power consumption patterns (Side-Channel Attacks). This highlights that algebraic correctness does not guarantee physical security; robust implementations must enforce constant-time execution to mitigate such hardware-level leaks.

5.3 Chain of Trust and Signatures

The analysis also highlights the role of RSA signatures in establishing the Chain of Trust. The server's certificate is not self-verified but signed by an intermediate CA (Let's Encrypt), which is in turn rooted in a trusted anchor (ISRG Root).

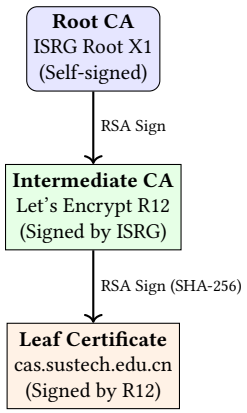

Figure 5. The Chain of Trust visualized. The validity of the university's key relies on the RSA signature provided by the issuer.

As visualized in Figure 5, the integrity of this chain relies on the `sha256WithRSAEncryption` algorithm. Here, RSA provides authenticity rather than confidentiality. The use of SHA-256 ensures collision resistance, preventing attackers from forging a certificate that matches the CA's signature. This empirical evidence confirms that the "Textbook RSA" learned in class is merely a primitive; its secure application requires a complex suite of padding and hashing standards to

bridge the gap between abstract number theory and secure network protocols.

6 Conclusion and Reflection

This project has systematically bridged the divide between discrete mathematical models and cryptographic engineering. We successfully reproduced the theoretical collapse of Textbook RSA under Håstad's Broadcast Attack and verified the robustness of the OAEP standard. However, the most significant outcome of this study extends beyond the code—it lies in the profound shift in understanding the relationship between mathematics and security.

6.1 Reflection: The Gap Between Math and Reality

The transition from witnessing a *0.002-second decryption* of Textbook RSA to seeing the immediate output of "*semantic garbage*" under OAEP provided a visceral understanding of the dangers of determinism.

- **The Impact of Randomness:** In our experiments, the mere introduction of a random salt in OAEP completely destroyed the algebraic structure that CRT relied on. This practical observation turned the abstract concept of "Determinism vs. Probabilistic Encryption" into a tangible engineering reality.
- **The Fragility of Implementation:** A particularly striking realization came during our survey of **RFC 8017** [3]. The fact that a mathematically perfect algorithm could be compromised by a trivial implementation detail—such as returning a specific "Invalid Padding" error message—was a paradigm shift. It underscores that *Mathematical Correctness does not equal Engineering Security*. Security is a full-stack challenge, requiring defensive programming just as much as sound number theory.

6.2 Future Outlook: The Inertia of Ecosystems

Finally, our case study of the university's infrastructure offers a nuanced perspective on the future of RSA.

- **Longevity vs. Theoretical Death:** While Shor's Algorithm theoretically sentences RSA to death in the post-quantum era, our probe revealed that RSA-2048 remains the absolute standard in critical infrastructure.
- **Ecosystem Inertia:** This highlights a complex engineering reality: the "inertia" and "compatibility" of the global digital ecosystem are often more decisive factors than raw algorithmic optimality. While the transition to Lattice-based cryptography is inevitable, RSA's integration into the Chain of Trust suggests it will remain a cornerstone of discrete mathematics applications for decades to come.

7 Acknowledgments

Special thanks to **Prof. Qi Wang**, who taught me Discrete Math (H). His lectures were rigorous and clear, especially the mathematical proofs. It was his class that first introduced me to the history of cryptography and the beauty of RSA.

Thanks to the seniors from the **Turing Class** for their valuable help and encouragement on learning this subject.

Thanks to my parents and friends for their unwavering support, especially during the finals week. When I was stressed out by unexpected team challenges, your encouragement kept me going.

Finally, special thanks to my boyfriend. Thank you for the countless late nights in the library and for always being by my side.

This is my first attempt at writing a report in a formal research paper format. I apologize for any potential errors.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126.

[2] Dan Boneh. 1999. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society* 46, 2 (1999), 203–213.

[3] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. 2016. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017. (November 2016).

A Implementation Details and Source Code

To ensure the reproducibility of the experimental results described in Sections 3 and 4, we provide the core implementation logic below.

A.1 Environmental Requirements

The simulations were executed in a Python 3.10+ environment. The cryptographic primitives (RSA, OAEP padding, and PKCS #1 v1.5) rely on the `pycryptodome` (v3.15.0+) library. For the infrastructure probe in Section 5, OpenSSL 3.0+ was utilized via the command-line interface.

A.2 Håstad's Broadcast Attack Simulation

This script (Listing 2) reproduces the broadcast scenario under $e = 3$ and demonstrates the CRT-based plaintext recovery.

```
import time
from math import gcd
from Crypto.Util.number import getPrime,
    long_to_bytes, bytes_to_long

# Mathematical Primitives
def integer_cube_root(n):
    """
    Computes the integer cube root of n
    using binary search.
    This avoids floating-point precision
    issues inherent in pow(n, 1/3).
```

```
    """
    low = 0
    high = n
    while low < high:
        mid = (low + high) // 2
        if mid**3 < n:
            low = mid + 1
        else:
            high = mid
    return low

# Broadcast Attack
def broadcast_attack(N_list, C_list):
    """
    Recovers the message m from 3 ciphertexts using CRT.
    Precondition: m^3 < N1 * N2 * N3
    """
    if gcd(N_list[0], N_list[1]) != 1 or \
       gcd(N_list[0], N_list[2]) != 1 or \
       gcd(N_list[1], N_list[2]) != 1:
        raise ValueError("Moduli are not pairwise coprime!")
    N1, N2, N3 = N_list
    C1, C2, C3 = C_list
    N_total = N1 * N2 * N3
    M1 = N2 * N3
    w1 = (C1 * M1 * pow(M1, -1, N1))
    M2 = N1 * N3
    w2 = (C2 * M2 * pow(M2, -1, N2))
    M3 = N1 * N2
    w3 = (C3 * M3 * pow(M3, -1, N3))

    C_combined = (w1 + w2 + w3) % N_total

    return integer_cube_root(C_combined)

# Simulation Experiment
def run_simulation():
    print("-" * 50)
    print("  RSA Broadcast Attack Simulation")
    print("-" * 50)
    print("[*] Generating 3 pairs of 1024-bit RSA keys (e=3)...")
    e = 3
    N_list = []
    for _ in range(3):
        p = getPrime(512)
        q = getPrime(512)
        N_list.append(p * q)
    print("    Done. Moduli generated.")
    message_str = "Secret Message: Only OAEP Padding can prevent this!"
    m = bytes_to_long(message_str.encode('utf-8'))
    print(f"\n[*] Original Message: \"{message_str}\"")
    if m**3 >= N_list[0] * N_list[1] * N_list[2]:
        print("Error: Message is too long for this specific attack.")
        return
    print("[*] Encrypting for 3 different recipients...")
    C_list = [pow(m, e, n) for n in N_list]
    print("    Ciphertexts captured.")
    print("\n[*] Launching Attack (CRT + Cubic Root)...")
    start_time = time.time()

    try:
        m_recovered_int = broadcast_attack(N_list, C_list)
        end_time = time.time()
        m_recovered_str = long_to_bytes(m_recovered_int).decode('utf-8')
        print(f"    [SUCCESS] Recovered in {end_time - start_time:.6f} seconds!")
        print(f"    Recovered: \"{m_recovered_str}\"")
        assert message_str == m_recovered_str
        print("    Verification: Exact match.")

    except Exception as err:
        print(f"    [FAILED] {err}")
    print("-" * 50)

if __name__ == "__main__":
    run_simulation()
```

Listing 2. Håstad's Attack Implementation

A.3 OAEP Defense Counter-Experiment

This script (Listing 3) evaluates the security of RSA-OAEP against the same broadcast parameters, verifying the achievement of IND-CPA security.

```
import binascii
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Util.number import bytes_to_long, long_to_bytes

# Import broadcast attack function from broadcast.py
try:
    from broadcast import broadcast_attack
except ImportError:
```

```
    print(" E o o r cannot  find broadcast.py")
    exit()

def oaep_defense_simulation():
    print("-" * 60)
    print("  PART 2: OAEP Defense Simulation (Counter-Experiment)")
    print("-" * 60)
    message = b"Secret Message: Only OAEP Padding can prevent this!"
    print(f"[*] Original Message: \"{message.decode()}\"")

    print("\n[*] Generating 3 pairs of 2048-bit RSA keys (forcing e=3)...")
    keys = []
    N_list = []
    for i in range(3):
        key = RSA.generate(2048, e=3)
        keys.append(key)
        N_list.append(key.n)
    print("    Done. Keys generated.")
    print("\n[*] Encrypting message using PKCS#1 v2.0 OAEP Standard...")
    C_list = []

    for i, key in enumerate(keys):
        cipher = PKCS1_OAEP.new(key)
        ciphertext = cipher.encrypt(message)
        c_int = bytes_to_long(ciphertext)
        C_list.append(c_int)
        prefix = binascii.hexlify(ciphertext[:8]).decode()
        print(f"    Ciphertext {i+1} (Hex prefix): {prefix}... (Randomized)")
    print("\n[*] Attacker attempting Hastad's Broadcast Attack...")
    print("    (Applying CRT and Cubic Root on OAEP ciphertexts...)")

    try:
        m_recovered_int = broadcast_attack(N_list, C_list)
        m_recovered_bytes = long_to_bytes(m_recovered_int)
        print(f"    [ATTACK FINISHED] Result calculated.")

        # Verify attack failed (recovered content should be garbage)
        if m_recovered_bytes == message:
            print("    [FATAL] Attack Succeeded! (This is impossible if OAEP works)")
        else:
            print(f"    [DEFENSE SUCCESS] The recovered content is GARBAGE.")
            hex_preview = binascii.hexlify(m_recovered_bytes[-20:]).decode() if len(m_recovered_bytes) > 20 else binascii.hexlify(m_recovered_bytes).decode()
            print(f"    Raw Hex (suffix): ...{hex_preview}")
            print("\n    Analysis:")
            print("    The attack failed because OAEP introduces RANDOMNESS (Salt).")
            print("    Even with e=3, the ciphertexts no longer share the same structure.")
            print("    This confirms that OAEP is IND-CPA secure.")

    except Exception as e:
        print(f"    [DEFENSE SUCCESS] Attack logic crashed: {e}")

    print("-" * 60)

if __name__ == "__main__":
    oaep_defense_simulation()
```

Listing 3. OAEP Defense Simulation

Code Availability

The full project repository, including auxiliary scripts and environment configuration files (`requirements.txt`), is available at: https://github.com/annaeina/Experimental-Study-on-H-stad-s-Broadcast-Attack-and-RSA-OAEP-Defense.