Improvement of RSA security with Optimal Asymmentric Encryption Padding

(Tarit Goswami,Tanurima Halder,Athina Saha and Oindrila Ray)
Department Of Computer Science & Engineering

1.1 Introduction to RSA Algorithm

The RSA algorithm is the basis of a cryptosystem -- a suite of cryptographic algorithms that are used for specific security services or purposes -- which enables public key encryption and is widely used to secure sensitive data, particularly when it is being sent over an insecure network such as the internet.RSA was first publicly described in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman of the Massachusetts Institute of Technology, though the 1973 creation of a public key algorithm by British mathematician Clifford Cocks was kept classified by the U.K.'s GCHQ until 1997.In RSA cryptography, both the public and the private keys can encrypt a message; the opposite key from the one used to encrypt a message is used to decrypt it. This attribute is one reason why RSA has become the most widely used asymmetric algorithm: It provides a method to assure the confidentiality, integrity, authenticity, and non-repudiation of electronic communications and data storage.

1.2 Why is RSA Algorithm used?

RSA derives its security from the difficulty of factoring large integers that are the product of two large prime numbers. Multiplying these two numbers is easy, but determining the original prime numbers from the total -- or factoring -- is considered infeasible due to the time it would take using even today's supercomputers.
The public and private key generation algorithm is the most complex part of RSA cryptography. Two large prime numbers, p and q, are generated using the Rabin-Miller primality test algorithm. A modulus, n, is calculated by multiplying p and q. This number is used by both the public and private keys and provides the link between them. Its length, usually expressed in bits, is called the key length.
Alan Way explains how RSA public key

encryption works
The public key consists of the modulus n and a public exponent, e, which is normally set at 65537, as it's a prime number that is not too large. The e figure doesn't have to be a secretly selected prime number, as the public key is shared with everyone.
The private key consists of the modulus n and the private exponent d, which is calculated using the Extended Euclidean algorithm to find the multiplicative inverse with respect to the totient of n.

1.3 RSA Security

RSA security relies on the computational difficulty of factoring large integers. As computing power increases and more efficient factoring algorithms are discovered, the ability to factor larger and larger numbers also increases.

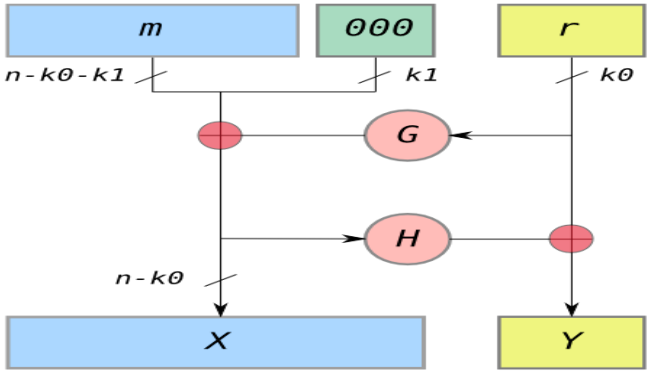
2.1 OPTIMAL ASYMMETRIC ENCRYPTION PADDING

The OAEP algorithm is a form of Feistel network which uses a pair of random oracles G and H to process the plaintext prior to asymmetric encryption. When combined with any secure trapdoor one-way permutation ‘f’ this processing is proved in the random oracle model to result in a combined scheme which is semantically secure under chosen plaintext attack (IND-CPA). When implemented with certain trapdoor permutations (e.g., RSA), OAEP is also proved secure against chosen ciphertext attack. OAEP can be used to build an all-or-nothing transform.
OAEP satisfies the following two goals:

1. Add an element of randomness which can be used to convert a deterministic encryption scheme (e.g., traditional RSA) into a probabilistic scheme.
2. Prevent partial decryption of ciphertexts (or other information leakage) by ensuring that an adversary cannot recover any portion of the plaintext without being able to invert the trapdoor one-way permutation ‘f’

The original version of OAEP (Bellare/Rogaway, 1994) showed a form of "plaintext awareness" (which they claimed implies security against chosen ciphertext attack) in the random oracle model when OAEP is used with any trapdoor permutation. Subsequent results contradicted

this claim, showing that OAEP was only IND-CCA1 secure. However, the original scheme was proved in the random oracle model to be IND-CCA2 secure when OAEP is used with the RSA permutation using standard encryption exponents, as in the case of RSA-OAEP.An improved scheme (called OAEP+) that works with any trapdoor one-way permutation was offered by Victor Shoup to solve this problem. More recent work has shown that in the standard model (that is, when hash functions are not modeled as random oracles) it is impossible to prove the IND-CCA2 security of RSA-OAEP under the assumed hardness of the RSA problem.



OAEP is a Feistel network(In cryptography, a **Feistel cipher** is a symmetric structure used in the construction of block ciphers, named after the German-born physicist and cryptographer Horst Feistel who did pioneering research while working for IBM (USA); it is also commonly known as a **Feistel network**. A large proportion of block ciphers use the scheme, including the Data Encryption Standard (DES).)

In the diagram,

- n is the number of bits in the RSA modulus.
- k_0 and k_1 are integers fixed by the protocol.
- m is the plaintext message, an $(n - k_0 - k_1)$-bit string
- G and H are random oracles such as cryptographic hash functions.
- $\oplus$ is an xor operation.

To encode,

1. messages are padded with k_1 zeros to be $n - k_0$ bits in length.
2. r is a randomly generated k_0-bit string
3. G expands the k_0 bits of r to $n - k_0$ bits.
4. X = m00..0 $\oplus$ G(r)
5. H reduces the $n - k_0$ bits of X to k_0 bits.
6. Y = r $\oplus$ H(X)

7. The output is X || Y where X is shown in the diagram as the leftmost block and Y as the rightmost block.

To decode,

1. recover the random string as r = Y ⊕ H(X)
2. recover the message as m00..0 = X ⊕ G(r)

3.1 How RSA Algorithm works?

The security of the algorithm is based on the hardness of factoring a large composite number and computing e th roots modulo a composite number for a specified odd integer e. An RSA public key consists of a pair (n, e) of integers, where n is the modulus and e is the public exponent. The modulus n is a large composite number (a bit length of at least 1024 is the current recommended size), while the public exponent e is normally a small prime such as 3, 17, or 65537. In this specification, the modulus is the product of two distinct primes. For a discussion of the case of three or more prime factors (so-called ''multi-prime'' RSA), see Appendix A at the end of this document. An RSA private key may have one of two different representations. Both representations contain information about an integer d satisfying (x e) d ≡ x (mod n) for all integers x. This means that the private key can be used to solve the equation x e ≡ c (mod n) in x, i.e., compute the e th root of c modulo n. The RSA encryption primitive RSAEP takes as input the public key (n, e) and a positive integer m < n (a message representative) and returns the integer c = me mod n. The RSA decryption primitive RSADP takes as input the private key and an integer c < n (a ciphertext representative) to return the integer m = c d mod n, where d is an integer with the above specified properties.

3.1.1 RSA Public Key

For the purposes of this document, an RSA public key consists of two components: n the modulus, a nonnegative integer e the public exponent, a nonnegative integer In a valid RSA public key, the modulus n is a product of two distinct odd primes p and q, and the public exponent e is an integer between 3 and n − 1 satisfying GCD(e, p − 1) = GCD(e, q − 1) = 1.

3.1.2 RSA Private Key

The representation consists of the pair (n, d), where the components have the following meanings: n the modulus, a nonnegative integer d the private exponent, a nonnegative integer In a valid RSA private

key with this representation, the modulus n is the same as in the corresponding public key and is the product of two odd primes p and q, and the private exponent d is a positive integer less than n satisfying e · d ≡ 1 (mod LCM(p − 1, q − 1)), where e is the corresponding public exponent.

Alice generates her RSA keys by selecting two primes: p=11 and q=13. The modulus is n=p×q=143. The totient is n ϕ(n)=(p−1)x(q−1)=120. She chooses 7 for her RSA public key e and calculates her RSA private key using the Extended Euclidean algorithm, which gives her 103.
Bob wants to send Alice an encrypted message, M, so he obtains her RSA public key (n, e) which, in this example, is (143, 7). His plaintext message is just the number 9 and is encrypted into ciphertext, C, as follows:
Me mod n = 97 mod 143 = 48 = C
When Alice receives Bob's message, she decrypts it by using her RSA private key (d, n) as follows:
Cd mod n = 48103 mod 143 = 9 = M
To use RSA keys to digitally sign a message, Alice would need to create a hash -- a message digest of her message to Bob -- encrypt the hash value with her RSA private key, and add the key to the message. Bob can then verify that the message has been sent by Alice and has not been altered by decrypting the hash value with her public key. If this value matches the hash of the original message, then only Alice could have sent it -- authentication and non-repudiation -- and the message is exactly as she wrote it -- integrity.

3.2 Java Implementation Of RSA Algorithm without padding

```
package org.oaep;
import org.apache.axiom.om.util.Base64;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import javax.crypto.Cipher;
import java.io.FileInputStream;
import java.security.*;
import java.security.cert.Certificate;

public class TestOAEP {
  public static void main(String [] args) throws Exception {
    Security.insertProviderAt(new BouncyCastleProvider(), 1);
    String plaintext = "admin";
    String ciphertext = Base64.encode(encrypt(plaintext));
```

```
    System.out.println("ciphertext " + ciphertext);
    String recoveredPlaintext = decrypt(Base64.decode(ciphertext));
   System.out.println("recoveredPlaintext   "+recoveredPlaintext);
}
private static byte [] encrypt(String plaintext) throws Exception {
    KeyStore keyStore = getKeyStore();
    Certificate[] certs = keyStore.getCertificateChain("oaep");
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, certs[0].getPublicKey());
    return cipher.doFinal(plaintext.getBytes());}
private static String decrypt(byte [] ciphertext) throws Exception {
    KeyStore keyStore = getKeyStore();
    PrivateKey privateKey = (PrivateKey) keyStore.getKey("oaep",
          "oaep".toCharArray());
    Cipher cipher = Cipher.getInstance("RSA");
    System.out.println(privateKey);
    cipher.init(Cipher.DECRYPT_MODE, privateKey);
    byte[] cipherbyte=cipher.doFinal(ciphertext);
    return new String(cipherbyte);}
public static KeyStore getKeyStore() throws Exception {
    String file ="newkeystore.jks";
    KeyStore keyStore = KeyStore
          .getInstance("JKS");
    String password = "oaep";
    FileInputStream in = null;
    try {
        in = new FileInputStream(file);
       keyStore.load(in, password.toCharArray());
    } finally {
       if (in != null) {
          in.close();
       }}
    return keyStore;
}}
```

3.3 Loop Holes Of RSA Algorithm

The operation at the core of RSA is a modular exponentiation: given input m, compute m^e modulo n. Although *in general* this is a one-way permutation of integers modulo n, it does not fulfill all the characteristics needed for generic asymmetric encryption:

- If e is small and m is small, then m^e could be smaller than n, at which point the modular exponentiation is no longer modular, and can be reverted efficiently.
- The operation is deterministic, which allows for exhaustive search *on the message*: the attacker encrypts possible messages until a match is found with the actual encrypted message.
- The modular exponentiation is malleable: given the "encryption" of m_1 and m_2, a simple multiplication yields the encryption of $m_1 m_2$. This is akin to homomorphic encryption, which can be a good property, or not, depending on the context.

For these reason, the integer m which is subject to RSA must not be the data to encrypt alone, but should be the result of a transform which ensures that m is "not small", contains some random bytes, and deters malleability.

3.4 How OAEP improves the security of RSA (with padding)

Here's how OAEP works; you take the plaintext message m to send, you pick a random value r, and compute the function:

Padded=OAEP(m,r)
Padded=OAEP(m,r)
(with H and G as parts of OAEP). Then, once we have that, we encrypt it using the base RSA public operation (using the public key):
Ciphertext=RSA(PublicKey,Padded)
Ciphertext=RSA(PublicKey,Padded)
The
Ciphertext
Ciphertext is what's actually sent; because it is encrypted using RSA, the attacker cannot recover
Padded
Padded.
So, if the OAEP operation doesn't prevent an attacker from decrypted the ciphertext in this straightforward manner, why is it there at all?
Well, raw RSA has some nastly properties (called an homeomorphic property), namely:

RSA(k,A)×RSA(k,B)=RSA(k,A×B)
RSA(k,A)×RSA(k,B)=RSA(k,A×B)
(where
×
× is implicitly modulo the public key modulus).
Because of this property, using raw RSA to directly encrypt plaintexts is almost always the wrong thing to do; there are clever ways to use this homeomorphic property to recover plaintext.

To elaborate this in the functional level, this padded value will be a part of the cipher text, this will be added to the Ciphertext after doing a ⊕ operation with a cryptographic hash function as below.

X=(m+some zeros for fixed length)⊕ G(r)
Y = r ⊕ H(X), Here G and H are cryptographic hash functions.
Ciphertext will be X || Y here
During the decryption process this can be reversed again with the same hash function as below
r = Y ⊕ H(X)
m+some zeros for fixed length= X ⊕ G(r)

m can be retrieved after removing the zeros. This strengthen the padding mechanism and work well against the chosen ciphertext attacks.

4. Conclusion

With **RSA** the **padding** is essential for its core function. **RSA** has a lot of mathematical structure, which leads to weaknesses. Using correct **padding** prevents those weaknesses.

We can fix a few issues by introducing padding.

1. Malleability: If we have a strict format for messages, i.e. that the first or last bytes contain a specific value, simply multiplying both message and ciphertext will decrease the probability of creating a valid (in terms of padding) message.

2. Semantical Security: Add randomness such that RSA is not deterministic anymore (a deterministic encryption scheme yields

always the same x for each instance of x=encpubkey(m) for constant m and pubkey). See OAEP as an example on how to achieve this.

What the OAEP function does is try to mimic a random function between plaintexts and padded versions; because we present the raw RSA operation with effectively random texts, these homeomorphic properties are no concern.