RSA

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 22 Dec 2021,
rsa.tex, r1945

Contents

RSA Public Key Algorithm

- Created Ron Rivest, Adi Shamir and Len Adleman in 1978
- Formed RSA Security (company) in 1982 to commercialise products
- Most widely used public-key algorithm
- RSA is a block cipher: plaintext and ciphertext are integers

As we will see, the plaintext and ciphertext are integers. Any data can be represented in binary, and then split into blocks, where each block is taken as an input to RSA.

More information about Rivest, Shamir and Adleman is given in Chapter **??**.

The RSA Algorithm for Encryption

- Step 1: Users generated RSA key pairs using RSA Key Generation Algorithm
- Step 2: Users exchange public key
- Step 3: Sender encrypts plaintext using RSA Encryption Algorithm
- Step 4: Receiver decrypts ciphertext using RSA Decryption Algorithm

The following will show the algorithms used in steps 1, 3 and 4. For now we assume the users can exchange public keys, noting that public keys do not need to be kept secret. For example, one method to exchange public keys over a network is to simply email the public key, unencrypted. It doesn't matter if an attacker intercepts the public key, since, by definition, it is public to everyone.

Later we will see that the exchange of public keys is in fact harder than it seems.

RSA Key Generation (algorithm)

Each user generates their own key pair

1. Choose primes p and q
2. Calculate $n = pq$
3. Select e: $gcd(\phi(n), e) = 1, 1 < e < \phi(n)$
4. Find $d \equiv e^{-1} \pmod{\phi(n)}$

The user keeps p, q and d private. The values of e and n can be made public.

- Public key of user, $PU = \{e, n\}$
- Private key of user $PR = \{d, n\}$

Note that the private key includes both d and n, however the same n is also included in the public key. So while n is included in the private key, it is not actually private. This describes the conceptual view of the RSA public and private key. Implementations of RSA may store additional information in the keys, especially the private key.

RSA Key Generation (exercise)

Assume user A chose the primes $p = 17$ and $q = 11$. Find the public and private keys of user A.

RSA Encryption and Decryption (algorithm)

Encryption of plaintext M, where $M < n$:

$$C = M^e \bmod n$$

Decryption of ciphertext C:

$$M = C^d \bmod n$$

Note the conceptual simplicity of the encryption and decryption algorithms, compared to DES and AES. Also note that the decryption algorithm is in fact identical to encryption—it is only the variable names that have changed.

Requirements of the RSA Algorithm

1. Successful decryption: Possible to find values of e, d, n such that $M^{ed} \bmod n = M$ for all $M < n$
2. Successful decryption: Encryption with one key of a key pair (e.g. PU) can only be successfully decrypted with the other key of the key pair (e.g. PR)
3. Computational efficiency: Easy to calculate $M^e \bmod n$ and $C^d \bmod n$ for all values of $M < n$
4. Secure: Infeasible to determine d or M from known information e, n and C
5. Secure: Infeasible to determine d or M given known plaintext, e.g. (M_1, C_1)

We will not show how RSA meets these requirements yet (it is covered in more depth later), but RSA does indeed meet these requirements.

The 1st requirement is that if a message is encrypted, then the decryption of the resulting ciphertext will produce the original message.

The 2nd requirement is that you can only use keys in the same key pair; using the wrong key will produce incorrect results.

The 3rd requirement is that users can easily perform the encrypt and decrypt operations. By "easily" we mean within reasonable time (i.e. seconds, not thousands of years).

The 4th requirement is that an attacker cannot find the private value d or the message.

The 5th requirement is that, even if the attacker knows old plaintext values and the corresponding ciphertext (which was obtained using the same key pair), they should not be able to find d or M.

Looking at the algorithms it is not immediately obvious how the security requirements are met. That is because, for example, the encryption algorithm is an equation with 4 variables (C, M, e, n), of which 3 are known to the attacker. Why can't the attacker re-arrange the equation and find the value of the unknown variable C? We will see some analysis of the security later.

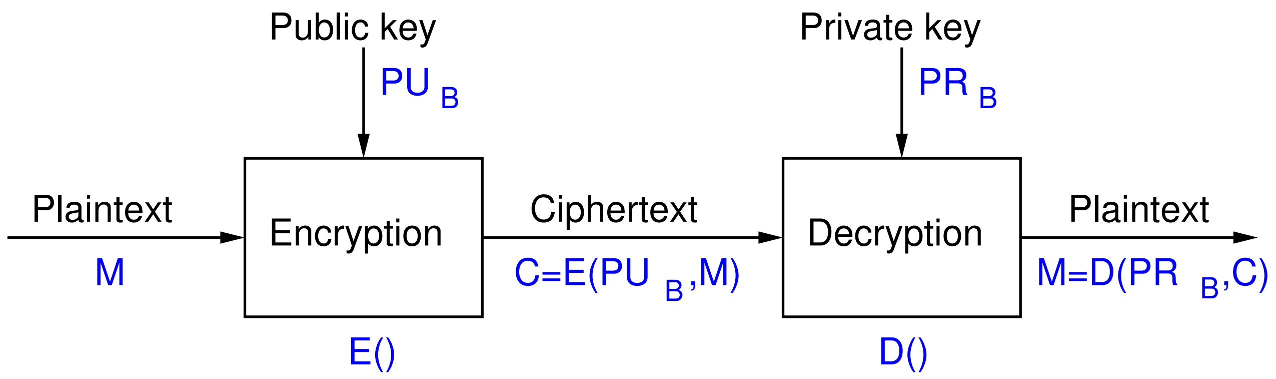
Ordering of RSA Keys

- RSA encryption uses one key of a key pair, while decryption must use the other key of that same key pair
- RSA works no matter the order of the keys
- RSA for confidentiality of messages
  - Encrypt using the public key of receiver
  - Decrypt using the private key of receiver
- RSA for authentication of messages
  - Encrypt using the private key of the sender (called signing)
  - Decrypt using the public key of the sender (called verification)
- In practice, RSA is primarily used for authentication, i.e. sign and verifying messages

Why does confidentiality work? Since the receiver is the only user that knows their private key, then they are the only user that can decrypt the ciphertext.

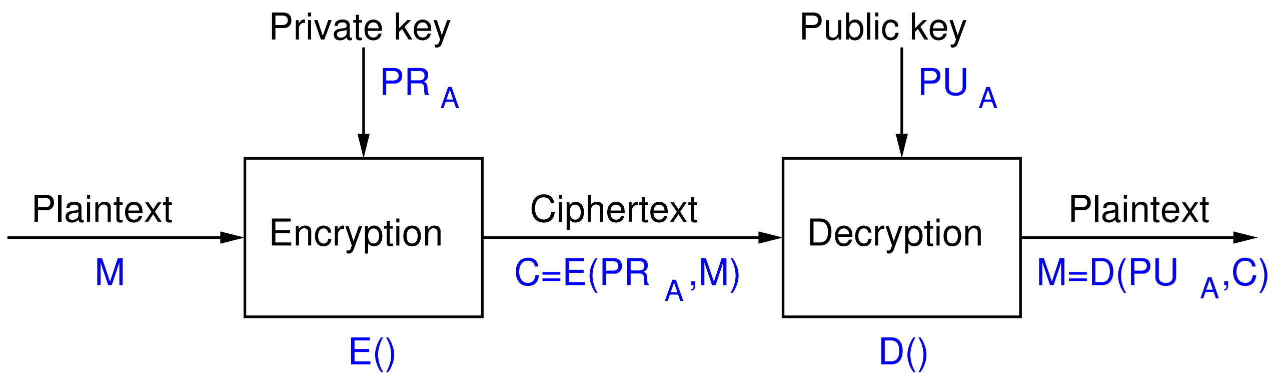
Why does authentication work? Since the sender is the only user that knows their private key, then they are the only user that can sign the message/plaintext. And the receiver can verify it came from that user if the signature decrypts successful with the sender's public key.

RSA used for Confidentiality



The figure on slide 10 shows RSA used to provide confidentiality of the message M. User A is on the left and user B is on the right. The operations E() and D() correspond to the encrypt and decrypt algorithms of RSA, respectively. User A encrypts the message using user B's public key, PU_B. The ciphertext is sent to user B. User B then decrypts using their own private key, PR_B.

RSA used for Authentication



The figure on slide 11 shows RSA used to provide authentication of the message M. The operations E() and D() correspond to the encrypt and decrypt algorithms of RSA, respectively, however they are more commonly referred to as signing and verification operations, respectively. User A encrypts/signs the message using their own private key, PR_A. The ciphertext/signed message is sent to user B. User B then decrypts/verifies using user A's public key, PU_A.

RSA Encryption for Confidentiality (exercise)

Assume user B wants to send a confidential message to user A, where that message, M is 8. Find the ciphertext that B will send A.

RSA Decryption for Confidentiality (exercise)

Show that user A successfully decrypts the ciphertext.

Contents

Why Does RSA Decryption Work?

- Encryption involves taking plaintext and raise to power e
- Decryption involves taking previous value and raise to a different power d
- Decryption must produce the original plaintext, that is:

$$(M^e)^d \bmod n = M \text{ for all } M < n$$

- This is true of if e and d are relatively prime
- Choose primes p and q, and calculate:

$$\begin{aligned}&n = pq\\&1 < e < \phi(n)\\&ed \equiv 1 \pmod{\phi(n)} \text{ or } d \equiv e^{-1} \pmod{\phi(n)}\end{aligned}$$

Here we see why the key generation algorithm is designed as it is. Decryption will only work (that is, produce the original plaintext) if the top equation is true. Note that $M^{e^d} = M^{ed}$. So the condition is that if you take the plaintext M and raise it to the power ed then the answer must be the original M (in mod n). For this to be true, e and d must be chosen appropriately—it will not work for just any value of e and d. Using Euler's theorem it can be shown that it will be true if e and d are multiplicative inverses of each other in mod $\phi(n)$.

Parameter Selection in RSA Key Generation

- Note: modular exponentiation is slow when using large values
- Choosing e
  - Values such as 3, 17 and 65537 are popular: make exponentiation faster
  - Small e vulnerable to attack; solution is to add random padding to each M
- Choosing d
  - Small d vulnerable to attack
  - But large d makes decryption slow
- Choosing p and q
  - p and q must be very large primes
  - Choose random odd number and test if its prime (probabilistic test)

As we saw in the exercise, key generation involves selecting values for p, q and e (where e influences the value of d as it is the multiplicative inverse).

As e is a public value, a small value can be selected (since a brute force is not relevant; the attacker already knows it) and in fact, many users can use the same value as each other. For example, OpenSSL defaults to using $e = 2^{16} + 1 = 65537$ for all keypairs generated. That is, by default everyone using OpenSSL to generate keypairs will have the same value of e. This value is small, meaning encryption is reasonable fast.

As d is the multiplicative inverse of e, a small e means d will be large. This is good, because d must be kept private; large values are not subject to brute force attack. But it makes decryption slow, since it involves M^d, which is often taking one very large number M and raising to the power of another very large number d. We will see later there are algorithms that can speed up the decryption process.

The primes p and q should be chosen randomly (again, they are private, so should be hard for an attacker to guess). A common approach is to choose a large odd number and then check if it is prime. There are primality testing algorithms that can either prove the number selected is prime, or give high confidence that it is prime (i.e. probabilistic test). When RSA is used for signatures—it's most common use—probabilistic testing is sufficient (it is faster than testing for provable primes).

Security of RSA

- Brute-Force attack: choose large d (but makes algorithm slower)
- Mathematical attacks:
  1. Factor n into its two prime factors
  2. Determine $\phi(n)$ directly, without determining p or q
  3. Determine d directly, without determining $\phi(n)$
- Factoring n is considered fastest approach; hence used as measure of RSA security
- Timing attacks: practical, but countermeasures easy to add (e.g. random delay). 2 to 10% performance penalty
- Chosen ciphertext attack: countermeasure is to use padding (Optimal Asymmetric Encryption Padding)

The three mathematical attacks require the attacker to solve computationally hard problems. That is, when large values are used,

Progress in Factorisation

- Factoring n into primes p and q is considered the easiest attack
- Some records by length of n:
  - 1991: 330 bits (100 digits)
  - 2003: 576 bits (174 digits)
  - 2005: 640 bits (193 digits)
  - 2009: 768 bits (232 digits), 10^{20} operations, 2000 years on single core 2.2 GHz computer
  - 2019: 795 bits (240 digits), 900 core years
- Improving at rate of 5–20 bits per year
- Typical length of n: 1024 bits, 2048 bits, 4096 bits

In the 1990's and 2000's, the RSA Challenge tasked researchers with factoring integers of various sizes. The numbers reported on this slide are mainly from successful attempts at the RSA Challenge.

The rate of improvement of integer factorisation, varies depending on where you consider the starting year. In any case, RSA keys of 2048 bits are considered secure for the near future.

We don't cover quantum computers and cryptography here. While it is important for the future, in 2018 the largest reported integer factored into primes using a quantum computer was 4088459, that is 22 bits. While in theory quantum computers will be able to make integer factorisation much easier (make RSA insecure), in practice there is a long way to go.

Contents

Recommended or Typical RSA Parameters

- RSA Key length: 1024, 2048, 3072 or 4096 bits
  - Refers to the length of n
  - 2048 and above are recommended
- p and q are chosen randomly; about half as many bits as n
- e is small, often constant; e.g. 65537
- d is calculated; about same length as n
- For detailed recommendations see NIST FIPS 186 Digital Signature Standard

As an example, with a RSA 1024 bit key, length of p and q will be about 512 bits, and the length of n will be 1024 bits. e could be 65537 which is 17 bits, and d will be approximately 1024 bits.

FIPS 186 provides details of the implementation of RSA to meet US government standards. It includes specific algorithms to use and some recommended values. It also sets requirements for selecting random primes.

Decryption with Large d is Slow

- Modular arithmetic, especially exponentiation, can be slow with very large numbers (1000's of bits)
- Use properties of modular arithmetic to simplify calculations, e.g.

$$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$$

- Also Euler's theorem and Chinese Remainder Theorem can simplify calculations
- Decryption is significantly slower than encryption since d is very large
- Implementations of RSA often store and use intermediate values to speed up decryption

While there are methods to speed up decryption in RSA (see the next slide), it is still significantly slower than encryption in practice.

RSA Implementation Example

- Encryption:
$$C = M^e \bmod n$$
- Decryption:
$$M = C^d \bmod n$$
- Modulus, n of length b bits
- Public exponent, e
- Private exponent, d
- Prime1, p, and Prime2, q
- Exponent1, $d_p = d \pmod{p-1}$
- Exponent2, $d_q = d \pmod{q-1}$
- Coefficient, $q_{inv} = q^{-1} \pmod{p}$
- Private values: $PR = \{n, e, d, p, q, d_p, d_q, q_{inv}\}$
- Public values: $PU = \{n, e\}$

We see the parameters used within OpenSSL. p, q, n, e and d are normal. However d_p, d_q and q_{inv} are intermediate values introduced and stored as part of the private key. They are used to speed up the decryption calculation. The decryption algorithm is split into multiple steps using these intermediate values, such that it is significant faster than if using a single step. However the end result is still the same.

While you don't need to know what the intermediate steps are, it is useful to know that these intermediate values exist, as you will see them when using RSA in practice (e.g. generating keys with OpenSSL).

Contents

RSA Key Generation (exercise)

Generate your own RSA key pair using the OpenSSL `genpkey` command. Extract your public key and then exchange public key's with another person (or if you want to do it on your own, generate a second key pair).

RSA Signing (exercise)

Create a message in a file, sign that message using the `dgst` command, and then send the message and signature to another person.

RSA Verification (exercise)

Verify the message you received.

RSA Performance Test (exercise)

Using the OpenSSL `speed` command, compare the performance of RSA encrypt/sign operation against the RSA decrypt/verify operation.

Contents

RSA in Python Cryptography Library

- `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/`