

# RSA

## Cryptography

School of Engineering and Technology  
CQUniversity Australia

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

# Contents

## RSA Algorithm

## Analysis of RSA

## Implementations of RSA

## RSA in OpenSSL

## RSA in Python

# 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

# 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

# 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  $PR = \{d, n\}$

# 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  $C$ :

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

# Requirements of the RSA Algorithm

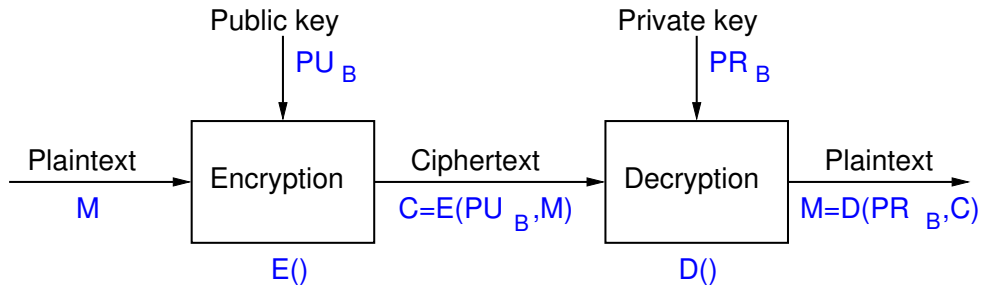
1. Successful decryption: Possible to find values of  $e$ ,  $d$ ,  $n$  such that  $M^{ed} \bmod n = M$  for all  $M < n$
2. Successful decryption: Encryption with one key of a key pair (e.g. PU) can only be successfully decrypted with the other key of the key pair (e.g. PR)
3. Computational efficiency: 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.  $(M_1, C_1)$



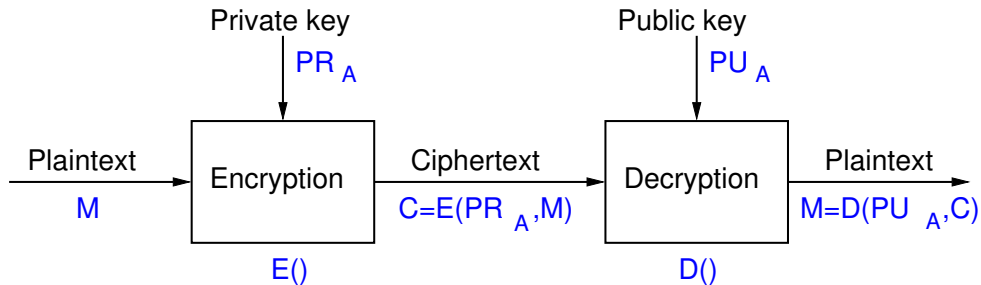
# Ordering of RSA Keys

- ▶ 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

# RSA used for Confidentiality



# RSA used for Authentication



# 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

RSA Algorithm

Analysis of RSA

Implementations of RSA

RSA in OpenSSL

RSA in Python

# 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:

$$n = pq$$

$$1 < e < \phi(n)$$

$$ed \equiv 1 \pmod{\phi(n)} \text{ or } d \equiv e^{-1} \pmod{\phi(n)}$$

# 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)



# 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)

# 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

# Contents

RSA Algorithm

Analysis of RSA

Implementations of RSA

RSA in OpenSSL

RSA in Python

# 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

# 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

# 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,  $d_p = d \bmod (p-1)$
- ▶ Exponent2,  $d_q = d \bmod (q-1)$
- ▶ Coefficient,  $q_{inv} = q^{-1} \bmod p$
- ▶ Private values:  $PR = \{n, e, d, p, q, d_p, d_q, q_{inv}\}$
- ▶ Public values:  $PU = \{n, e\}$

# Contents

RSA Algorithm

Analysis of RSA

Implementations of RSA

RSA in OpenSSL

RSA in Python

# 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 Algorithm

Analysis of RSA

Implementations of RSA

RSA in OpenSSL

RSA in Python

# RSA in Python Cryptography Library

- ▶ <https://cryptography.io/en/latest/hazmat/primitives/asymmetric/>