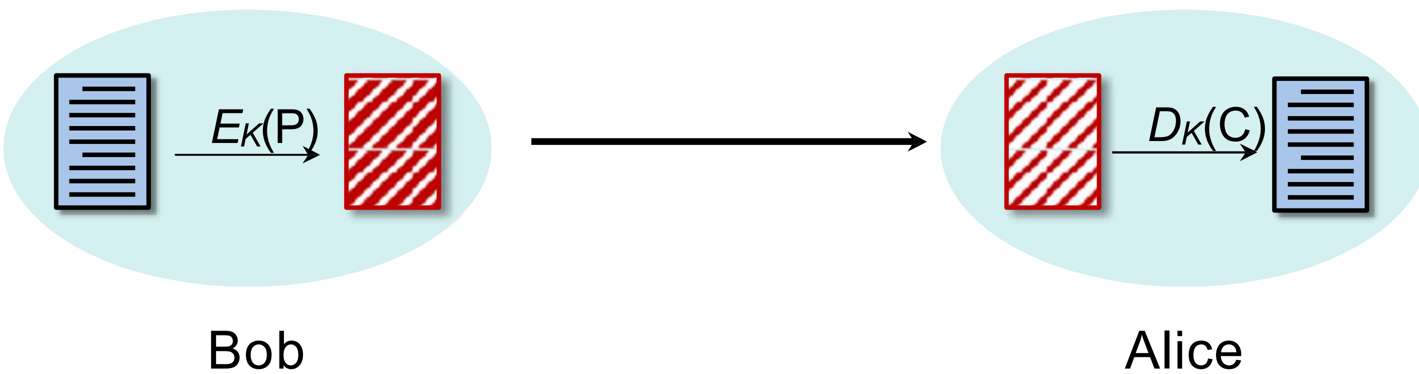
Asymmetric Cryptography and Integrity

Key Distribution

Communicating with symmetric cryptography

- Both parties must agree on a secret key, *K*
- Message is encrypted, sent, decrypted at other side



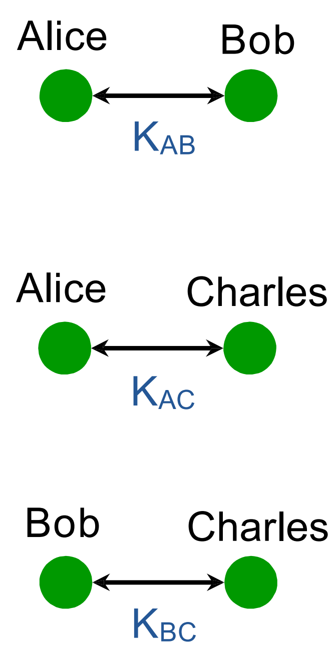
Key distribution must be secret. Otherwise

- Messages can be decrypted by the adversary
- Users can be impersonated

Problems With Keys In Symmetric Cryptography

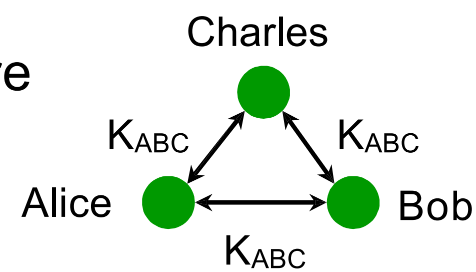
Key Management

- Potentially a lot of keys to track
- Every group of users needs a key



Key Distribution

- How do you communicate with someone you've never met?
- You cannot send them the secret key if the communication line is not secure



Key Distribution

Secure key distribution is the biggest problem with symmetric cryptography

Public Key Cryptography

Public-key algorithm

Two related keys:

$C = E_{K1}(P) \quad P = D_{K2}(C)$

$C' = E_{K2}(P) \quad P = D_{K1}(C')$

K_1 is a public key

K_2 is a private key

Examples:

- RSA, Elliptic curve algorithms
- DSS (digital signature standard)

Trapdoor functions

- Public key cryptography relies on trapdoor functions
- Trapdoor function
  - Easy to compute in one direction
  - Inverse is difficult to compute without extra information

- Example:

  96171919154952919 is the product of two prime #s. What are they?

  But if you're told that one of them is 100225441

  … then it's easy to compute the other: 959555959

RSA Public Key Cryptography

Ron Rivest, Adi Shamir, Leonard Adleman created the first public key encryption algorithm in 1977

Each user generates two keys:

- Private key (kept secret)
- Public key (can be shared with anyone)

Difficulty of algorithm based on the difficulty of factoring large numbers

- Keys are functions of a pair of large (~300 digits) prime numbers

RSA algorithm: key generation

1. Choose two random large prime numbers p, q
2. Compute the product $n = pq$ and $\phi = (p - 1)(q - 1)$

   n will be presented with the public & private keys. Length(n) is the key length
3. Choose the public exponent, e, such that:

   $1 < e < \phi$ and $\gcd(e, \phi) = 1$ [e and $(p - 1)(q - 1)$ are relatively prime]
4. Compute the secret exponent, d such that:

   $ed = 1 \bmod \phi$

   $d = e^{-1} \bmod ((p - 1)(q - 1))$
5. Public key = (e, n)

   Private key = (d, n)

   Discard p, q, ϕ

See https://www.di-mgt.com.au/rsa_alg.html

RSA algorithm: key generation and encryption

- Choose $p = 3$ and $q = 11$
- Compute $n = p * q = 3 * 11 = 33$
- Compute $\varphi(n) = (p - 1) * (q - 1) = 2 * 10 = 20$
- Choose e such that $1 < e < \varphi(n)$ and e and φ (n) are coprime. Let $e = 7$
- Compute a value for d such that $(d * e) \% \varphi(n) = 1$. One solution is $d = 3$ [$(3 * 7) \% 20 = 1$]
- Public key is (e, n) => (7, 33)
- Private key is (d, n) => (3, 33)
- The encryption of $m = 2$ is $c = 2^7 \% 33 = 29$
- The decryption of $c = 29$ is $m = 29^3 \% 33 = 2$

See https://www.di-mgt.com.au/rsa_alg.html

RSA Encryption

Key pair: public key = (e, n)
private key = (d, n)

Encrypt

- Divide data into numerical blocks < n
- Encrypt each block:

$$c = m^e \bmod n$$

Decrypt

$$m = c^d \bmod n$$

RSA security

The security of RSA encryption rests on the difficulty of factoring a large integer

Public key = { *modulus*, *exponent* }, or $\{n, e\}$

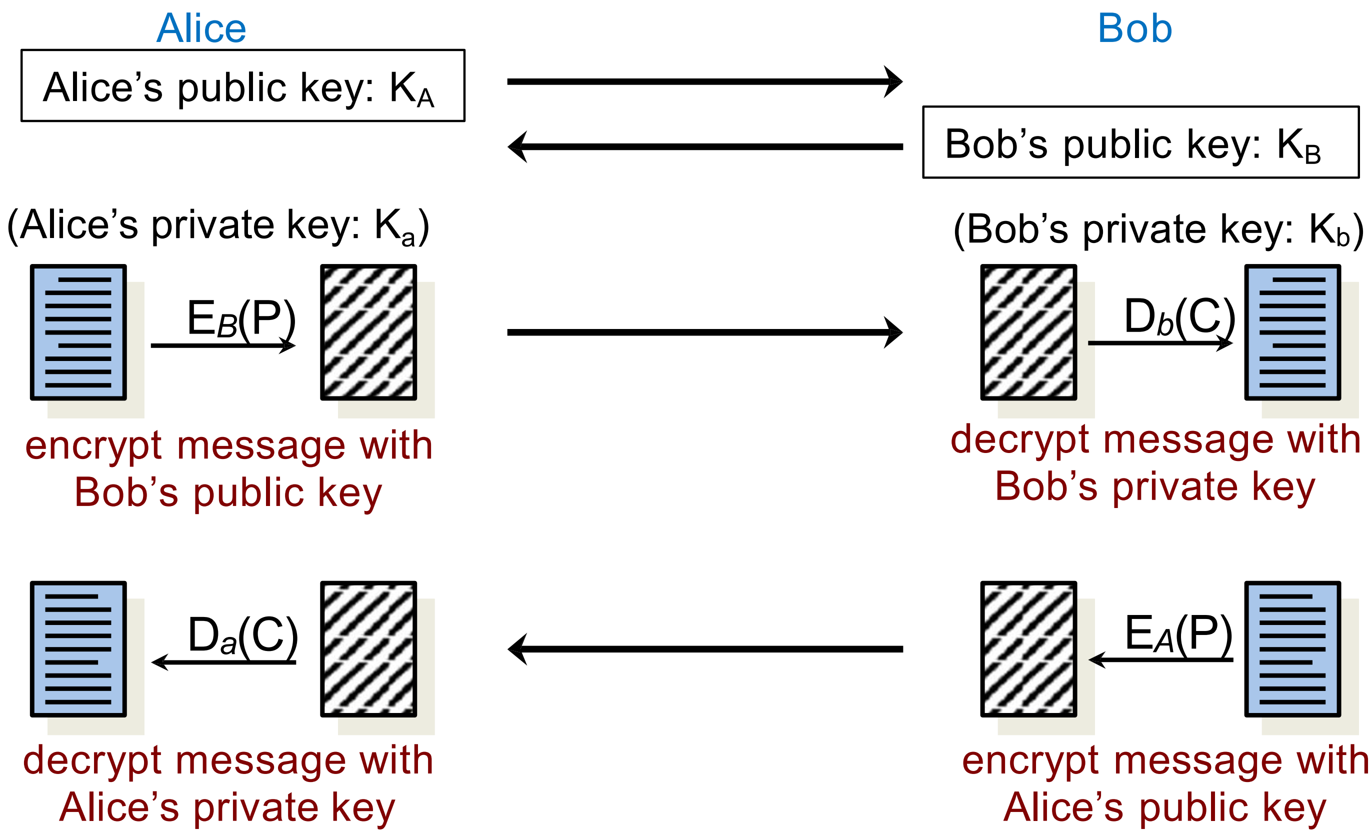
- The *modulus* is the product of two primes, p, q
- The private key is derived from the same two primes

Communication with public key algorithms

Different keys for encrypting and decrypting

- No need to worry about key distribution

Communication with public key algorithms



RSA isn't good for communication

Calculations are very expensive relative to symmetric algorithms

Common speeds:

Algorithm	Bytes/sec
AES-128-ECB	148,000,000
AES-128-CBC	153,000,000
AES-256-ECB	114,240,000
RSA-2048 encrypt	3,800,000
RSA-2048 decrypt	96,000

AES ~1500x faster to decrypt; 40x faster to encrypt than RSA

If anyone learns your private key, they can read all your messages

Key Exchange

Diffie-Hellman Key Exchange

Key distribution algorithm

- Allows two parties to share a secret key over a non-secure channel
- *Not* public key encryption
- Based on difficulty of computing discrete logarithms in a finite field compared with ease of calculating exponentiation

Allows us to negotiate a secret common key without fear of eavesdroppers

Diffie-Hellman Key Exchange

- All arithmetic performed in a field of integers modulo some large number

- Both parties agree on
  - a large prime number p
  - and a number $a < p$

- Each party generates a public/private key pair

Private key for user i: X_i

Public key for user i: $Y_i = a^{X_i} \bmod p$

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B

$$\boldsymbol{K = (\text{Bob's public key})^{(\text{Alice's private key})} \bmod p}$$

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B
- Bob computes

$$K = Y_A^{X_B} \bmod p$$

K′ = (Alice's public key) $^{\textbf{\textit{(Bob's private key)}}}$ ***mod p***

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- expanding:

$$\begin{aligned} K &= Y_B^{X_A} \bmod p \\ &= (a^{X_B} \bmod p)^{X_A} \bmod p \\ &= a^{X_B X_A} \bmod p \end{aligned}$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B
- Bob computes

$$K = Y_A^{X_B} \bmod p$$

- expanding:

$$\begin{aligned} K &= Y_B^{X_B} \bmod p \\ &= (a^{X_A} \bmod p)^{X_B} \bmod p \\ &= a^{X_A X_B} \bmod p \end{aligned}$$

K = K′

K is a *common key*, known *only* to Bob and Alice

Diffie-Hellman simple example

Assume p=1151, α=57

- Alice's secret key $X_A = 300$
- Alice's public key $Y_A = 57^{300} \bmod p = 282$
- Alice computes

$$K = Y_B^{X_A} \bmod p = 1046^{300} \bmod p$$

$$K = 105$$

- Bob's secret key $X_B = 25$
- Bob's public key $Y_B = 57^{25} \bmod p = 1046$
- Bob computes

$$K = Y_A^{X_B} \bmod p = 282^{25} \bmod p$$

$$K = 105$$

Given p=1151, α=57, Y_A=282, Y_B=1046, you cannot get 105

Message Integrity

One-way functions

- Easy to compute in one direction
- Difficult to compute in the other

Examples:

Factoring:

$pq = N$	EASY	Basis for RSA
find p,q given N	DIFFICULT	

Discrete Log:

$a^b \bmod c = N$	EASY	Basis for Diffie-Hellman & Elliptic Curve
find b given a, c, N	DIFFICULT	

Example of a one-way function: middle squares

Example with a 20-digit number

A = 18932442986094014771

A^2 = 3584373974217004547796075311891661824 41

Middle square, B = 42170045477960753118

Given A, it is easy to compute B

Given B, it is difficult to compute A

"Difficult" = no known short-cuts; requires an exhaustive search

Cryptographic hash functions

Cryptographic hash functions

Also called *digests* or *fingerprints*

Properties

- Arbitrary length input → fixed-length output
- Deterministic: you always get the same hash for the same message
- One-way function (pre-image resistance, or *hiding*)
  - Given *H*, it should be difficult to find *M* such that *H=hash(M)*
- Collision resistant
  - Infeasible to find any two different strings that hash to the same value:
    Find *M*, *M'* such that *hash(M) = hash(M')*
- Output should not give any information about any of the input
  - Like cryptographic algorithms, relies on *diffusion*
- Efficient
  - Computing a hash function should be computationally efficient

Hash functions are the basis of integrity

- Not encryption
- Can help us to detect:
  - Masquerading:
    - Insertion of message from a fraudulent source
  - Content modification:
    - Changing the content of a message
  - Sequence modification:
    - Inserting, deleting, or rearranging parts of a message
  - Replay attacks:
    - Replaying valid sessions

Hash Algorithms

Use iterative structure like block ciphers do … but use no key

- Example:
  - Secure Hash Algorithm, SHA-1
    - Designed by the NSA in 1993; revised in 1995
    - US standard for use with NIST Digital Signature Standard (DSS)
    - Produces 160-bit hash values
    - Chosen prefix collision attacks demonstrated May 2019
- Successors
  - SHA-2 (2001)
    - Produces 224, 256, 384, or 512-bit hashes
    - Approved for use with the NIST Digital Signature Standard (DSS)
  - SHA-3 (2015)
    - Can be substituted for SHA-2
    - Improved robustness

Message Integrity

How do we detect that a message has been tampered?

- A cryptographic hash acts as a checksum
- Associate a hash with a message
  - we're not encrypting the message
  - we're concerned with *integrity*, not *confidentiality*
- If two messages hash to different values, we know the messages are different

$$H(M) \neq H(M')$$

Tamperproof Integrity:
Message Authentication Codes and Digital Signatures

Message Integrity: MACs

We rely on hashes to assert the integrity of messages

But an attacker can create a new message & a new hash and replace *H(M)* with *H(M')*

So, let's create a checksum that <u>relies on a key for validation</u>

Message Authentication Code (MAC)

Two forms: hash-based & block cipher-based