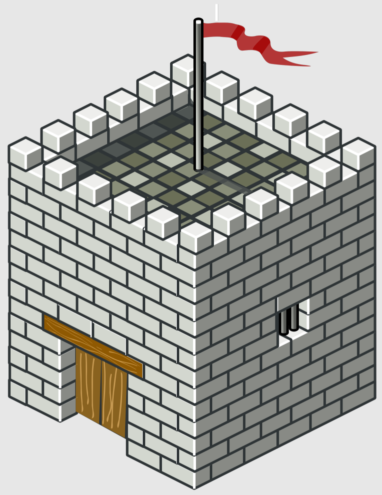
Foundations of Cybersecurity

IX-Public-Key Cryptography



Paweł Szałachowski

2017

Primes

- *a* divides *b* if you can divide b by a w/o leaving a reminder
    - *a* | *b*, e.g., *7* | *35*
- a number is a *prime* when it has two positive divisors (1 and itself)
    - otherwise the number is called a *composite*

Primes

- If $a \mid b$ and $b \mid c$ then $a \mid c$.
- Let $n > 1$ and $d > 1$ be the smallest divisor of n. Then d is prime.
- There are an infinite number of primes.
- Any integer > 1 can be written in exactly one way as the product of primes.

Modulo

- Operator modulo: *a mod N* returns remainder after division of *a* by *N*
  - Results are *0,1,...,N-1*   *e.g., 25 mod 7 = 4*
  - *to compute r = (a mod N),* find integers *q* and *r: a = qN + r*
    - *25 mod 7 = 4*   what if *a* is negative (e.g., -3) *?*
- In cryptography *N* is usually a prime
  - we use notation *mod p*

Computations Modulo

- Addition
  - (a + b) mod N
    - Compute and reduce modulo
  - (a + b + c + d) mod N
    - You can compute (a mod N + b mod N + ...) mod N
- Subtraction analogically

Computations Modulo

- $x*y \bmod N = y*x \bmod N$
- $\underbrace{x*x*\ldots*x}_{a} \bmod N = x^a \bmod N$
- $x^{ab} \bmod N = x^{ba} \bmod N$
- $(x^a)^b \bmod N = x^{ab} \bmod N$

Computations Modulo

- Division
  - $a/b \bmod N$ is the multiplication $ab^{-1} \bmod N$
    - (another notation of b^{-1} is $1/b$)
  - b^{-1} (a modular inverse of b) is a number such that $bb^{-1} = 1 \bmod N$
    - *What is* $5^{-1} \bmod 7$ *?*
  - How to compute modular inverses ?

The Greatest Common Divisor

- GCD of numbers *a* and *b* is the largest *k* such that $k \mid a$ and $k \mid b$

```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a – b;
        else
            b := b – a;
    return a;
```

Extended Euclidean Algorithm

For given (*a,b*) returns (*r,s,t*) such that *r=gcd(a,b)* and *sa + tb = r*

```
function egcd(a, b)
    s := 0;    old_s := 1
    t := 1;    old_t := 0
    r := b;    old_r := a
    while r ≠ 0
        quotient := old_r div r
        (old_r, r) := (r, old_r - quotient * r)
        (old_s, s) := (s, old_s - quotient * s)
        (old_t, t) := (t, old_t - quotient * t)
    return (old_r, old_s, old_t)
```

- How to compute $b^{-1} \bmod p$ (for $1 <= b < p$)?
    - Compute *r,s,t = egcd(b, p)*
    - *sb + tp = r* (r will be 1 as *p* is prime)
    - *sb = 1 - tp*, so *sb = 1 mod p*, so *s = 1/b mod p*

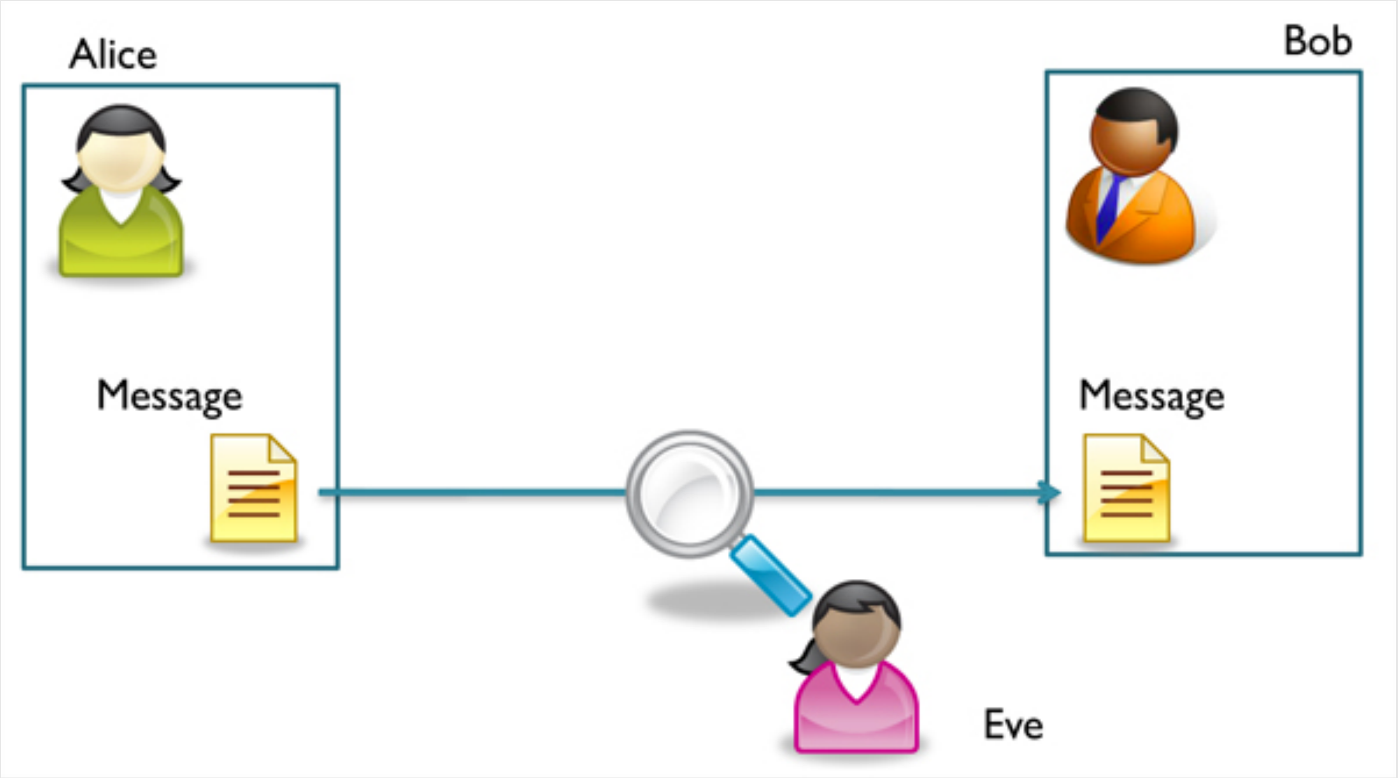
Generating Large Primes

- 2048-8192 bits long primes
- Probabilistic
    - Take a random number and check if it passes primality test(s)
    - The Rabin-Miller test

Diffie-Hellman (DH)

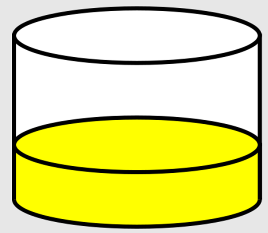
Problem definition

- Secure communication over insecure channel?
  - Two parties: **A**lice and **B**ob
  - Eavesdropping adversary: **E**ve



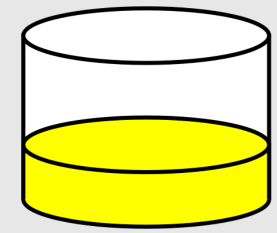
- How Alice and Bob can establish a shared secret?

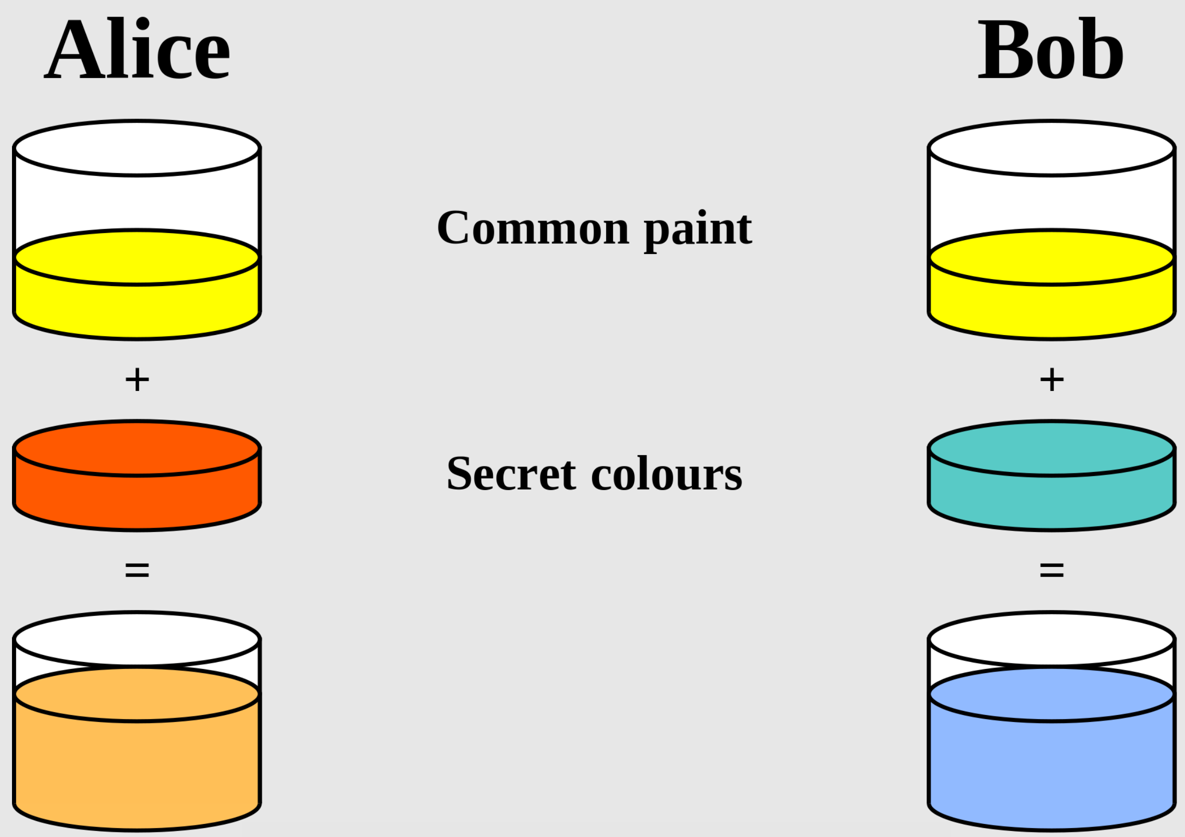
Alice

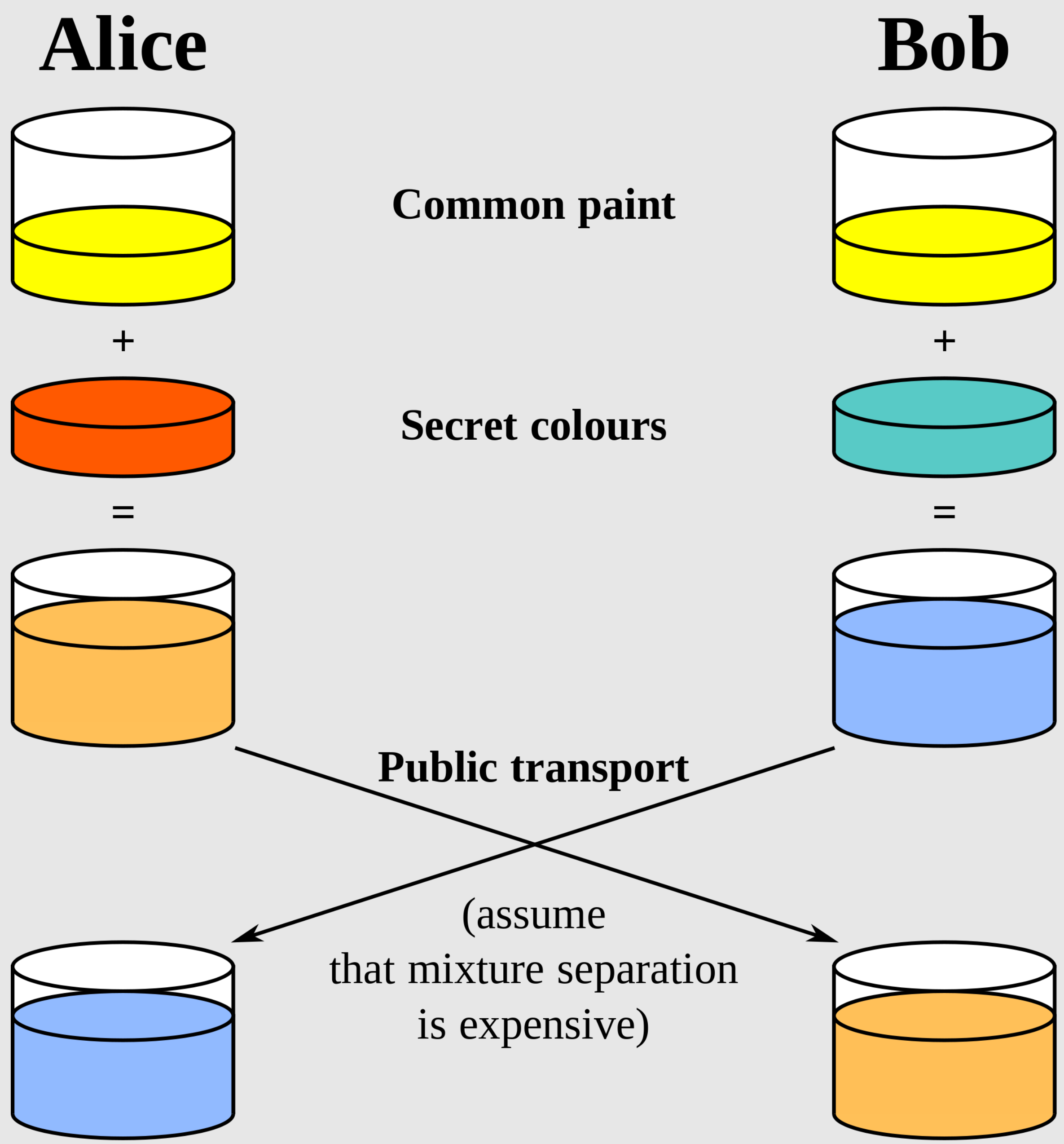


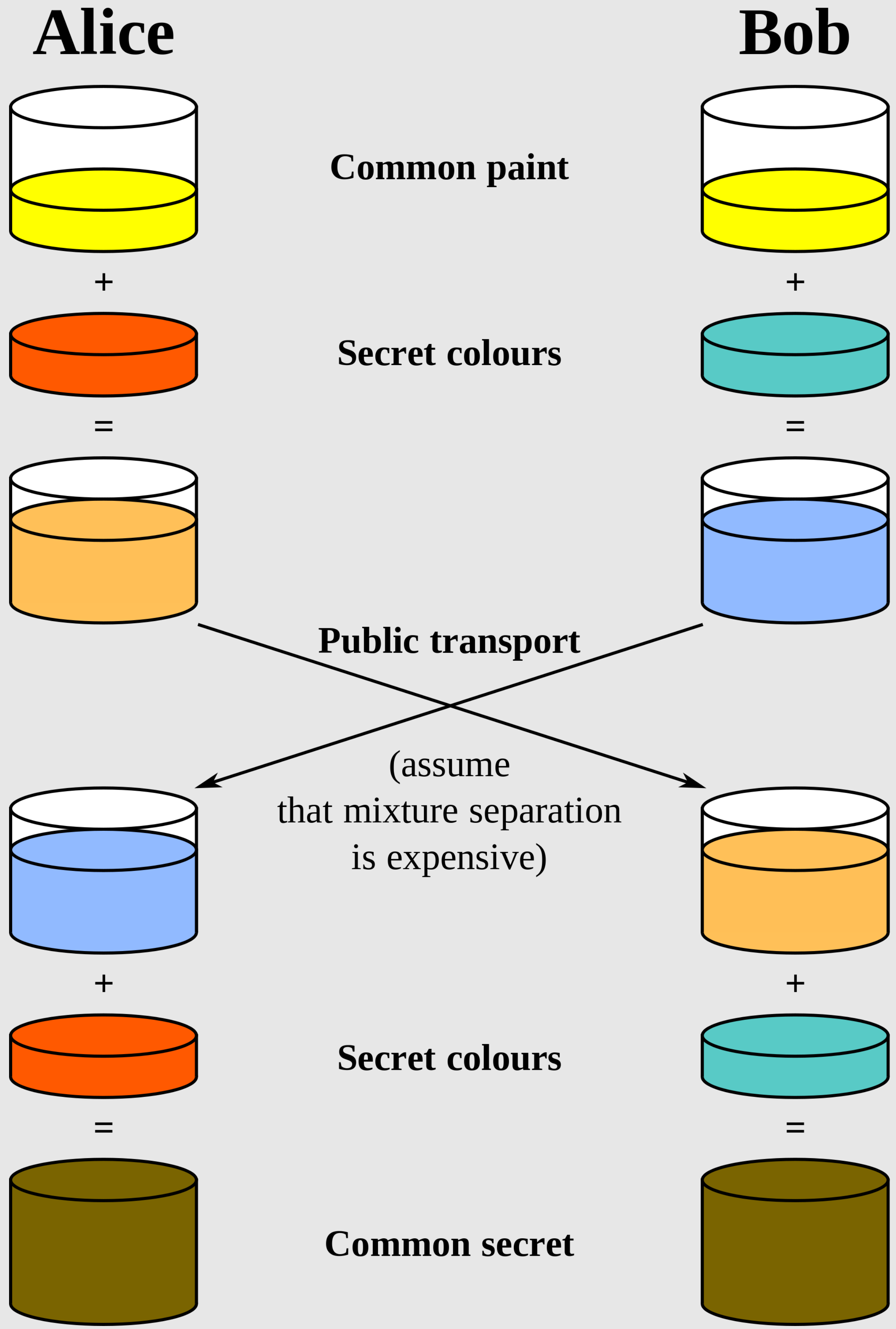
Common paint

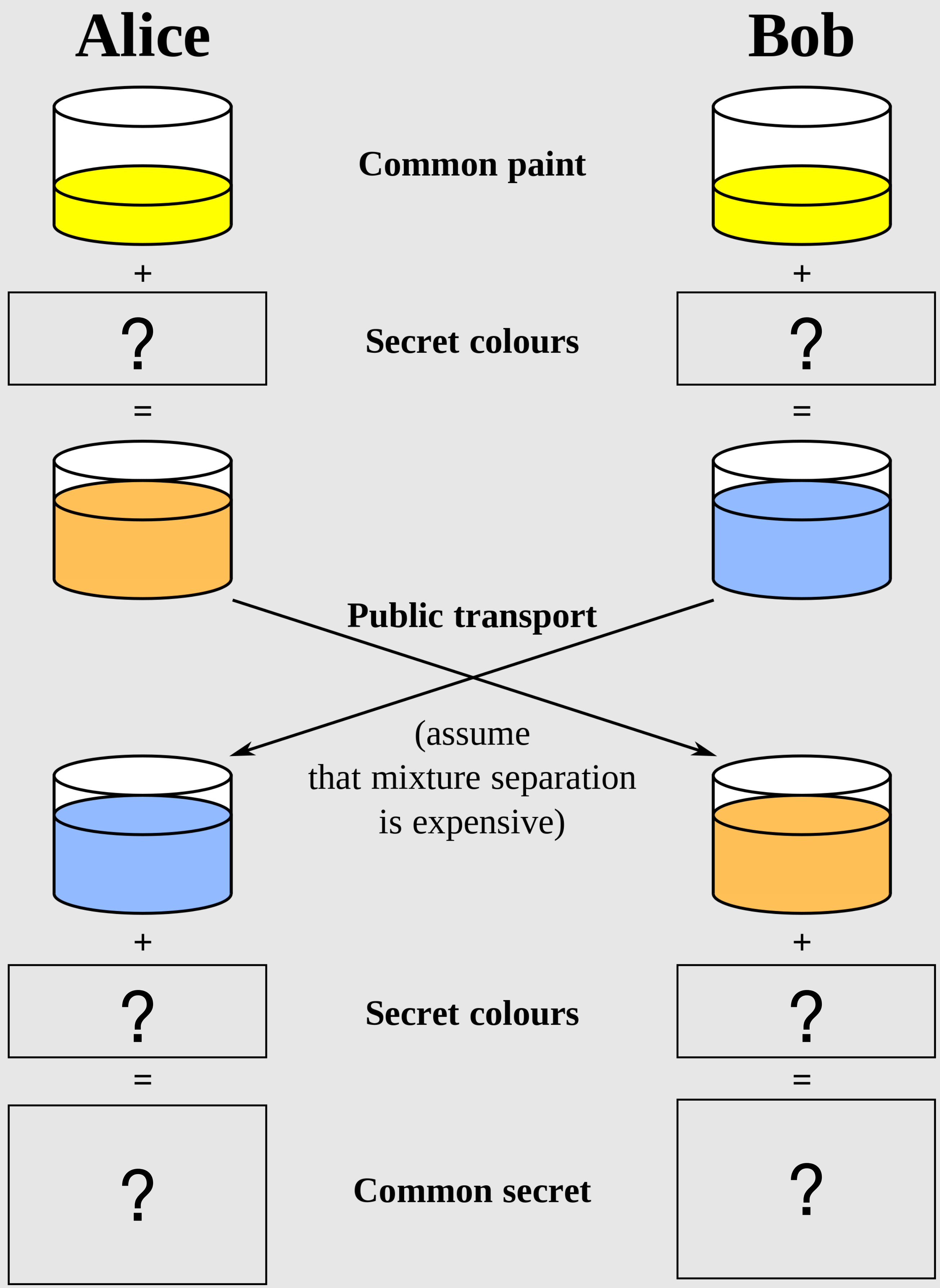
Bob




Alice
Bob
Common paint
+
+
Secret colours
=
=

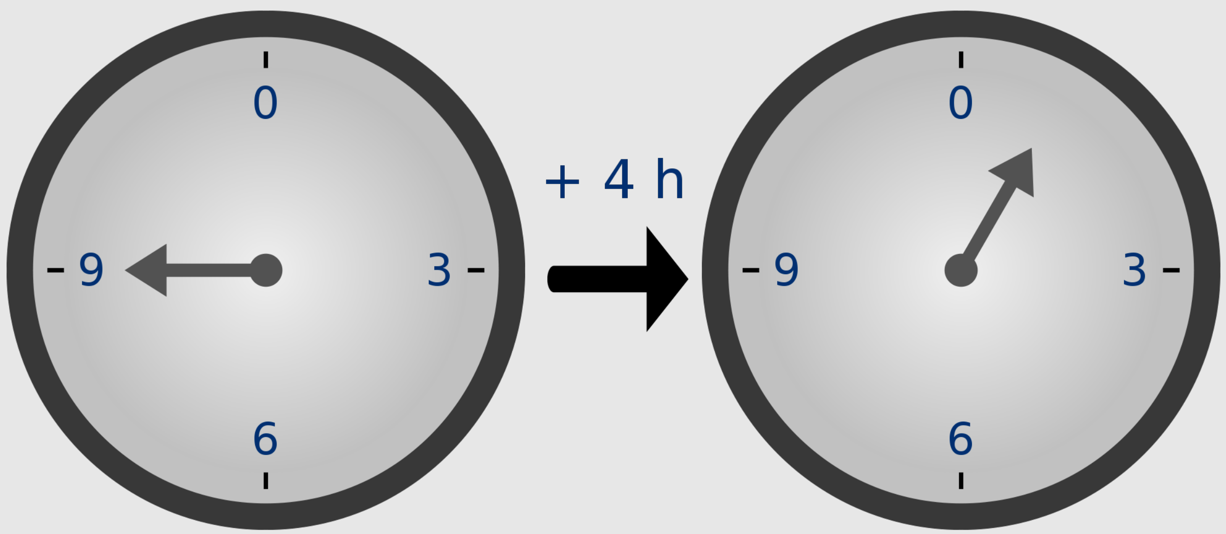

Alice
Bob
Common paint
+
+
Secret colours
=
=
Public transport
(assume
that mixture separation
is expensive)


Alice
Bob
Common paint
+
Secret colours
=
Public transport
(assume
that mixture separation
is expensive)
+
Secret colours
=
Common secret


Alice
Bob
Common paint
+
?
Secret colours
=
Public transport
(assume
that mixture separation
is expensive)
Secret colours
Common secret

Math Background: cyclic group

- Group (reminder)
  - Set and operation, for example [0, 11] and addition *mod* 12



- Multiplicative group of integers modulo N
  - set [1, N-1] and multiplication *mod* N

Math Background: cyclic group

- ***g*** is a <u>generator</u> of ***mod N*** if every element of **[1, *N-1*]** can be written as ***g^x mod N***
- Every number > 1 is a generator if *N* is prime

Example: *mod 11, g = 2*

2^0 *mod 11 = 1*	2^5 *mod 11 = 10*
2^1 *mod 11 = 2*	2^6 *mod 11 = 9*
2^2 *mod 11 = 4*	2^7 *mod 11 = 7*
2^3 *mod 11 = 8*	2^8 *mod 11 = 3*
2^4 *mod 11 = 5*	2^9 *mod 11 = 6*

Discrete Logarithm Problem

- Discrete Logarithm Problem (DLP):

  for known ***Y*,*g*, *N*** find ***X*** such that: $Y = g^X \bmod N$

- Examples: $g = 2, N = 13$

  $2 = 2^X \bmod 13 \quad X = 1$

  $3 = 2^X \bmod 13 \quad X = 4$

  $4 = 2^X \bmod 13 \quad X = 2$

  $5 = 2^X \bmod 13 \quad X = 9$

- Difficult (secure) when N is a large prime (e.g., 2048 bits)

2143512082772104306311491706279052757332819365350270236916619636267651473110852794594690121588759046304882342815119985428892042
604427608335711847366885192193296128232974167042736105925970485551575408786146057302507914866994805958463029863674231507767605
8654193185282927250356998785958415575881841411031093880658086633067469830081139764522105170108562855558139043580800539734898746
108361004674150661832306964399024263472249734260526991394535358856194229841900239384394337166360046344734779600165530865879362
144752939863330997697036578519527084377910216025745541416611237904706819511395029439640094554495074110424652379

Alice **Eve** **Bob**

Publicly known parameters: *g, p* (large prime)

Random secret *a*

$$\xrightarrow{g^a \bmod p}$$

Random secret *b*

$$\xleftarrow{g^b \bmod p}$$

$K = (g^b)^a \bmod p$ $K = (g^a)^b \bmod p$

Properties

- Parameters can be sent by Alice (don't have to be hardcoded)
- DH problem: Eve has to compute K with g^a *mod p* and g^b *mod p*
  - If she can solve DLP then it is trivial to compute K
  - At least as easy as DLP. Can it be easier than solving DLP?
- Efficiency
  - $g^{p-1} \bmod p = 1$, thus $g^a \bmod p = g^{(a \bmod p-1)} \bmod p$
  - easy for $g = 2$ (can express other generators as 2^x)

Security

- Key and parameters sizes

Date	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash	
2017 - 2022	128	2000	250	2000	250	SHA-256 SHA-512/256 SHA-384 SHA-512	SHA3-256 SHA3-384 SHA3-512
> 2022	128	3000	250	3000	250	SHA-256 SHA-512/256 SHA-384 SHA-512	SHA3-256 SHA3-384 SHA3-512

- The protocol is unauthenticated
    - Secure only against passive adversaries
    - Eve can impersonate Alice to Bob and Bob to Alice

RSA

Public-Key Encryption

- Gen()
  - returns a key pair (i.e., public and private key)
- Enc(pub_key, msg)
  - Encrypts a message using a public key. Returns a ciphertext.
- Dec(priv_key, ctxt)
  - Decrypts a ciphertext using a private key. Returns a message.

RSA Encryption

- Gen()
  - Select (large) random prime numbers *p, q* such that *p!=q*
  - Compute modulus $n = pq$
  - Compute $\Phi = (p-1)(q-1)$
  - Select public exponent $e = 1 < e < \Phi$ such that $gcd(e, \Phi) = 1$
  - Compute private exponent $d = e^{-1} \bmod \Phi$
  - Return public key *(n, e)*, and private key *d*
- Enc(*{n, e}*, *msg*)
  - return $msg^e \bmod n$
- Dec(*d*, *ctxt*)
  - return $ctxt^d \bmod n$

Digital Signatures

- Gen()
  - returns a key pair (i.e., public and private key)
- Sign(priv_key, msg)
  - Signs the message using the private key. Returns the signatures
- Verify(pub_key, msg, sign)
  - Verifies the signatures of the message, using the public key. Returns boolean (true/false).

RSA Signatures

- Gen()   (the same as in the encryption)
  - Select (large) random prime numbers *p, q* such that *p!=q*
  - Compute modulus $n = pq$
  - Compute $\Phi = (p-1)(q-1)$
  - Select public exponent $e = 1 < e < \Phi$ such that $gcd(e, \Phi) = 1$
  - Compute private exponent $d = e^{-1} \bmod \Phi$
  - Return public key *(n, e)*, and private key *d*
- Sign(*d, msg*)
  - return $H(msg)^d \bmod n$
- Verify(*{n, e}, msg, sign*)
  - return $sign^e \bmod n == H(msg)$

Properties

- RSA Problem
  - Compute *P* given (*n,e*) and $C = P^e \bmod n$
  - At least as easy as integer factorization of *n*. Can it be easier?
- Do not use the same keypair for encrypting and signing
- *n* should be >= 2048 bits

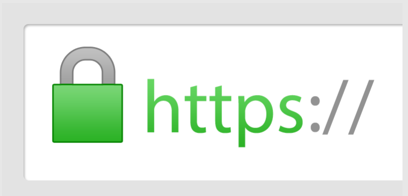
Date	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash	
2017 - 2022	128	2000	250	2000	250	SHA-256 SHA-512/256 SHA-384 SHA-512	SHA3-256 SHA3-384 SHA3-512
> 2022	128	3000	250	3000	250	SHA-256 SHA-512/256 SHA-384 SHA-512	SHA3-256 SHA3-384 SHA3-512

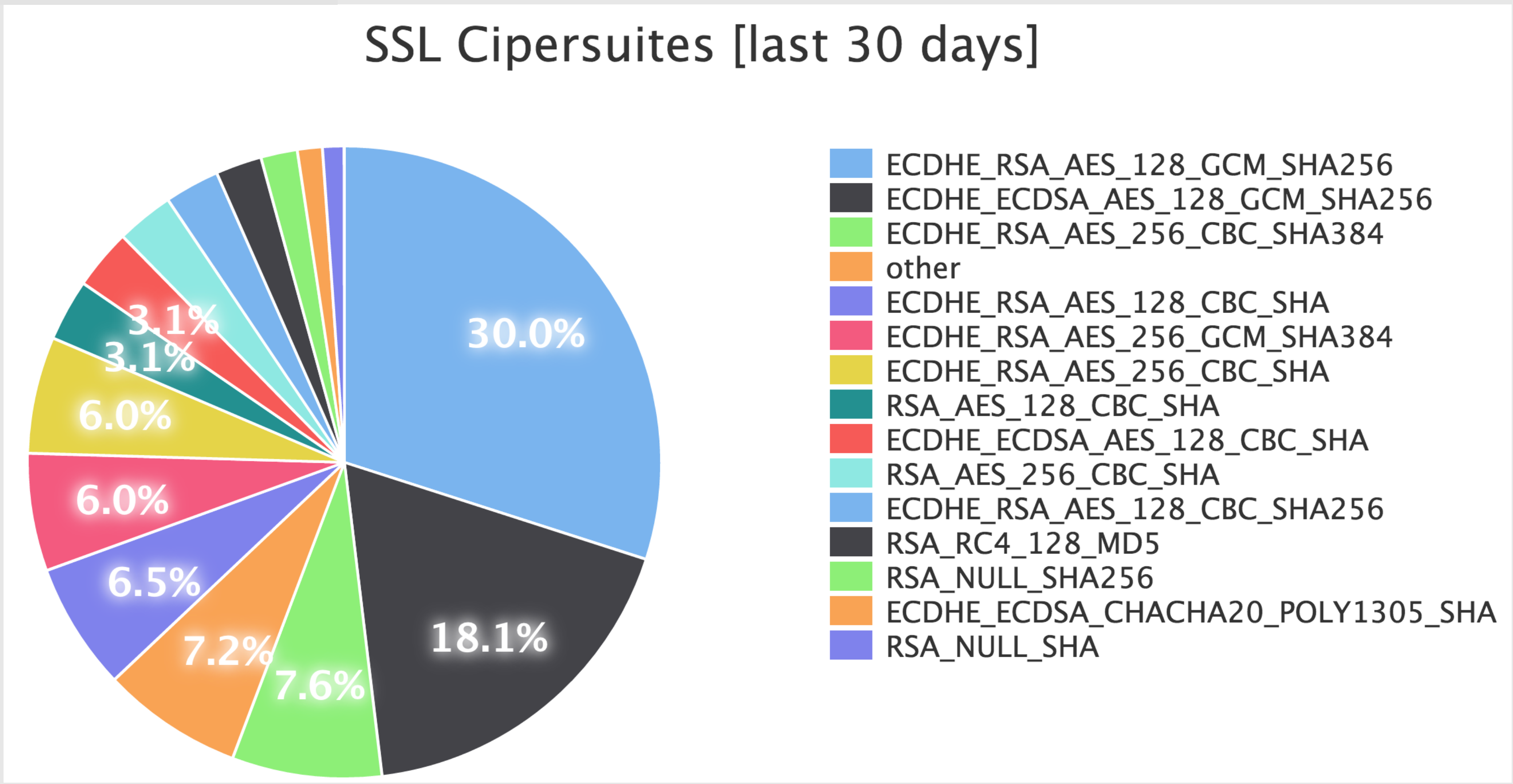
- *p* and *q* should be of equal size
- Timing attacks

Properties

- Encryption
  - *e* is usually small to speed up computations
    - Be careful with encrypting short messages
  - Adv. can try to precompute ciphertexts if message space is small
  - Can also distinguish encryptions
    - If two messages are the same, the ciphertexts will be the same
  - Optimal Asymmetric Encryption Padding

Why it is Important?





Discussion&Classwork