



University of
Pittsburgh

Applied Cryptography and Network Security

CS 1653



Summer 2023
Sherif Khattab
ksm73@pitt.edu

(Slides are adapted from Prof. Adam Lee's CS1653 slides.)

Announcements

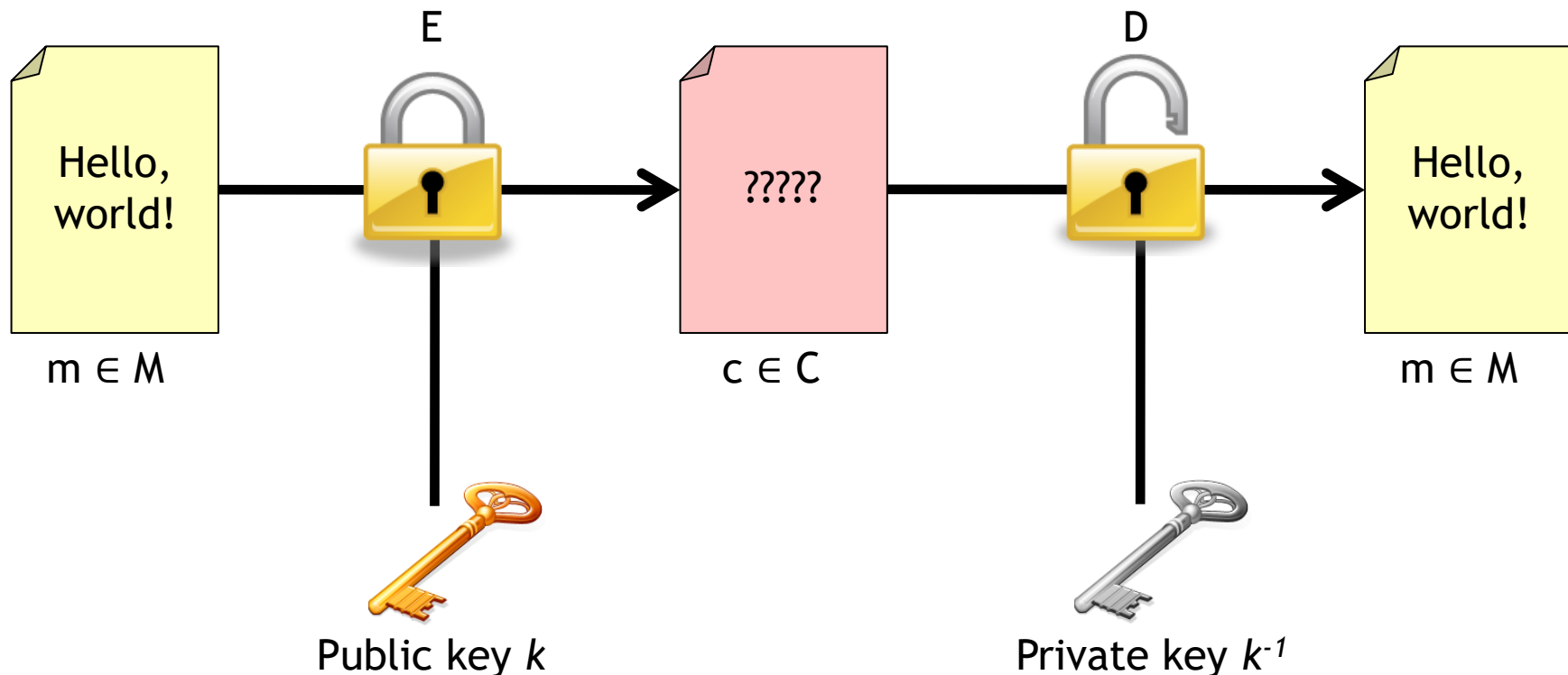
- Homework 3 due this Friday @ 11:59 pm
- Phase 2 of Project
 - due on Friday 6/30 @ 11:59 pm
- Makeup lecture on Friday 6/16 @ 11:00 am

Public key cryptosystems

Formally, a cryptosystem can be represented as the 5-tuple (E, D, M, C, K)

- M is a message space
- K is a key space
- $E : M \times K \rightarrow C$ is an encryption function
- C is a ciphertext space
- $D : C \times K \rightarrow M$ is a decryption function

Note: Each “key” in K is actually a pair of keys, (k, k^{-1})



What can we do with public key cryptography?

First, we need some way of finding a user's public key



Print it in
the newspaper



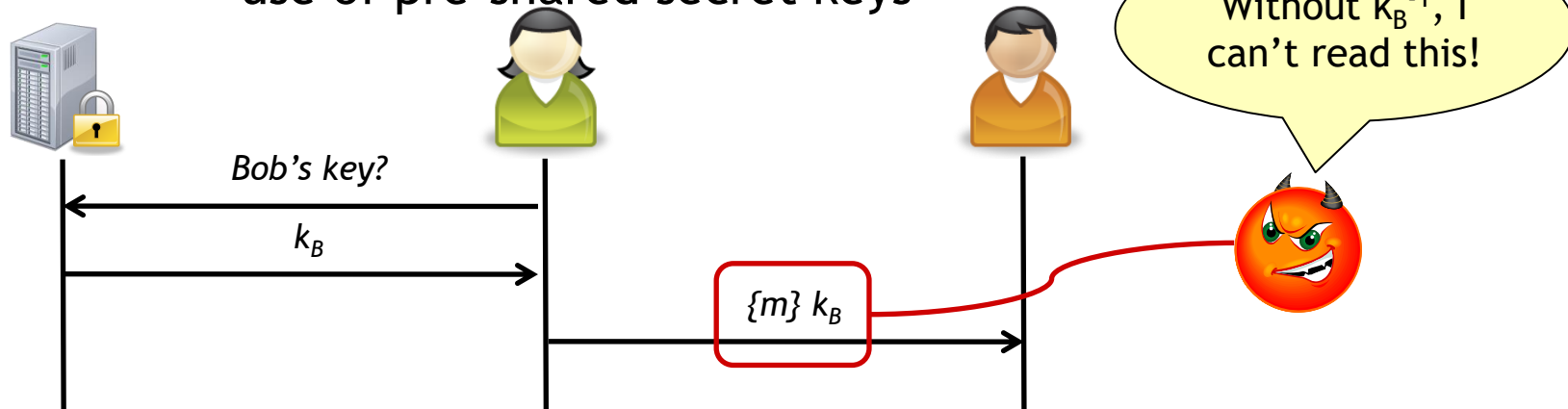
Post it on your
webpage



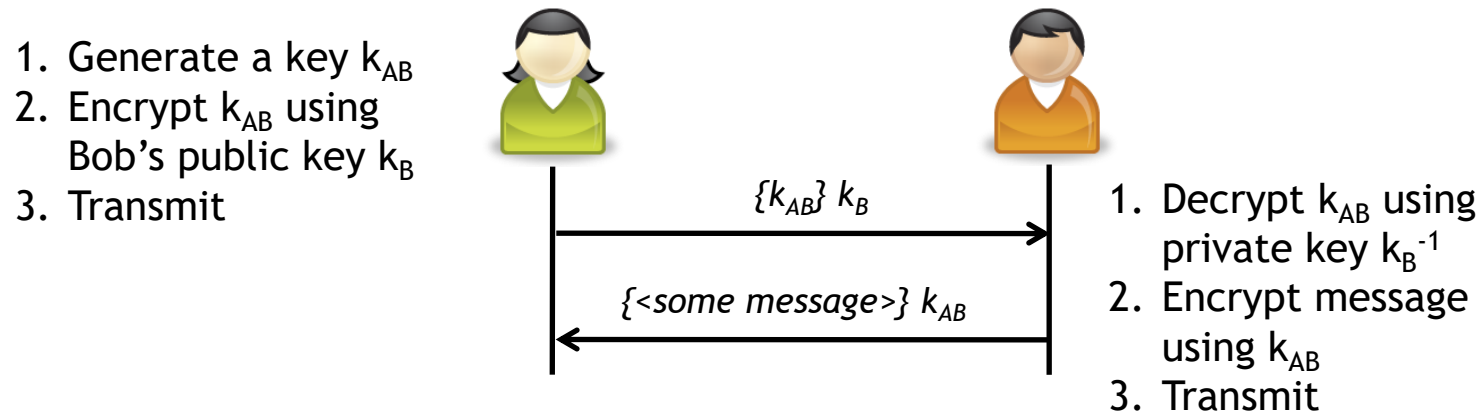
A trusted
keyserver (PKI)

Important: It is critical to verify the authenticity of any public key!
(How?)

Public key cryptography allows us to send private messages **without** the use of pre-shared secret keys



Public key cryptography help exchange symmetric keys



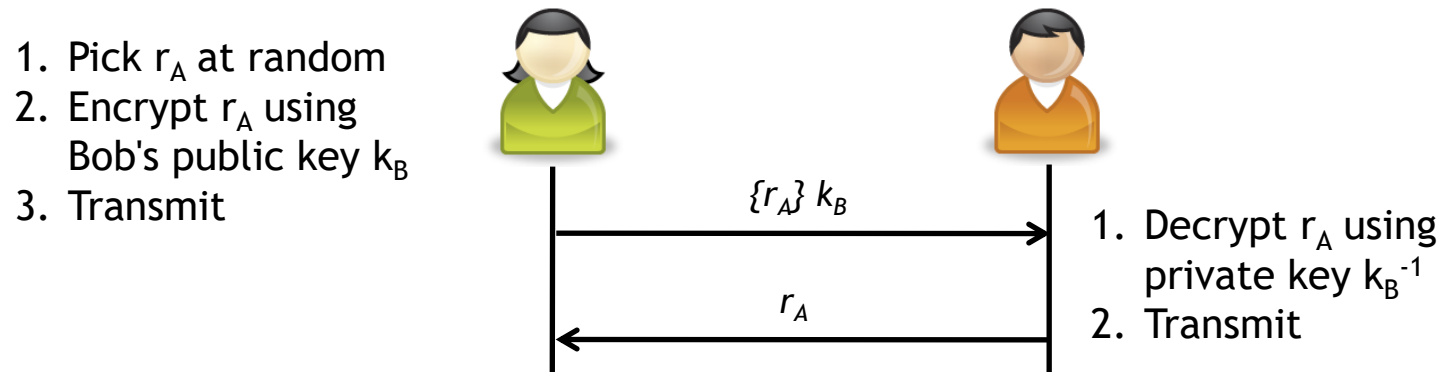
Note: Only Bob can decode k_{AB} , since only he knows k_B^{-1}

- Unfortunately, Bob doesn't know who this key is from
- Key exchange is not quite this easy in practice, but it isn't *much* harder

Question: Why do we want to exchange symmetric keys?!

- Public key cryptography is usually pretty slow...
 - Based on “fancy” math, not bit shifting
 - Symmetric key algorithms are orders of magnitude faster
- It's always a good idea to change keys periodically

Public key cryptography to authenticate users



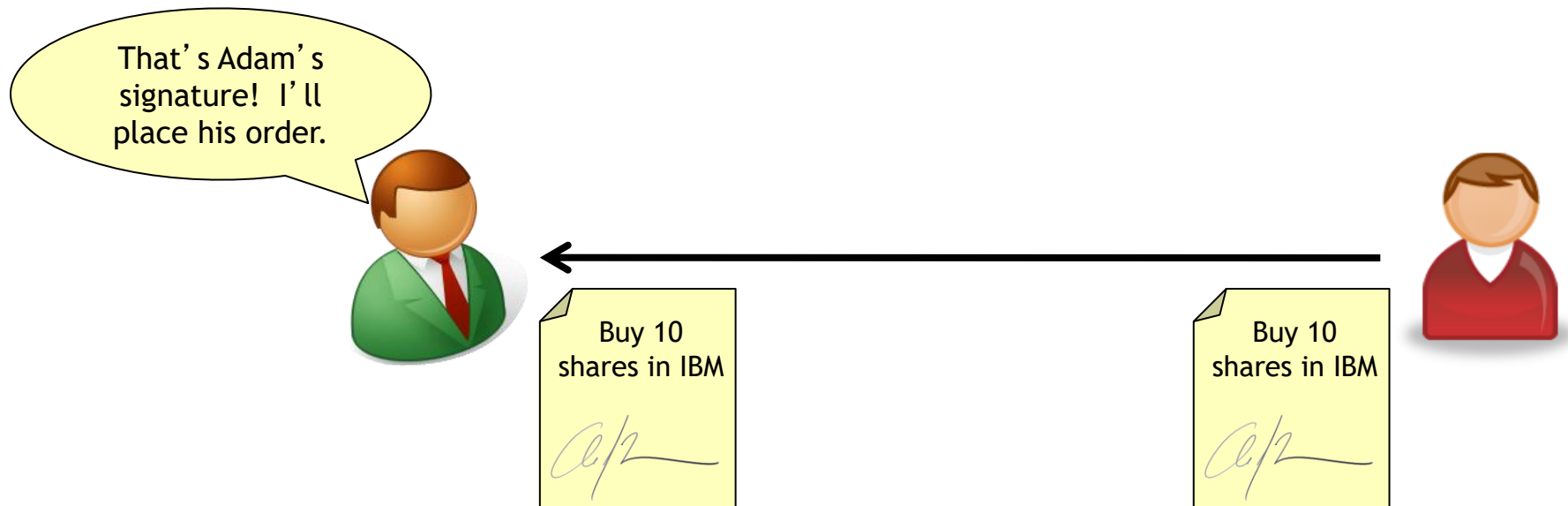
Note: As in the previous key exchange, only Bob can decrypt r_A , since only Bob knows k_B^{-1} .

It is of absolute importance that the random numbers used during this type of protocol are **not predictable** and are **never reused** (Why?)

- **Unpredictable:**
 - The security of this protocol is a proof of possession of k_B^{-1}
 - If predictable, an adversary can guess the “challenge” without decrypting!
 - (This is bad news)
- **Reusing** challenges may* lead to replay attacks (When?)

Public key systems also let us create digital signatures

Goal: If Bob is given a message m and a signature $S(m)$ supposedly computed by Adam, he can determine whether Adam wrote m



For this to occur, we require that

- The signature $S(m)$ must be **unforgeable**
- The signature $S(m)$ must be **verifiable**

Question: *How can we do this?*

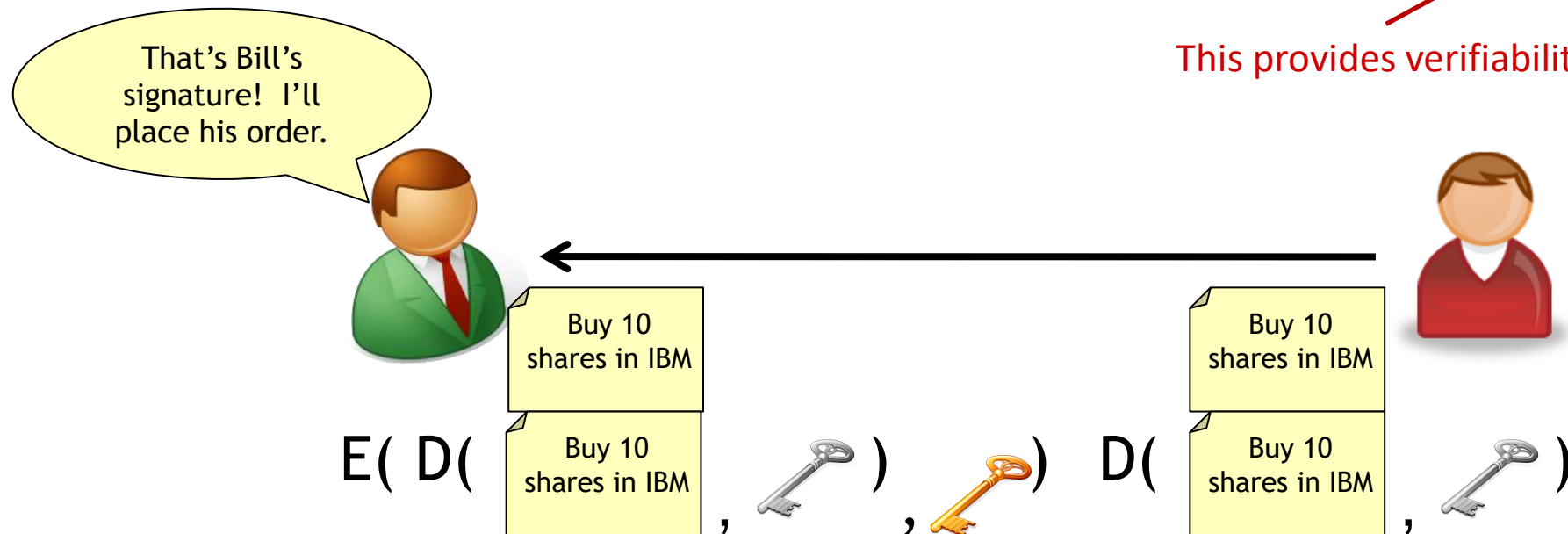
In many public key cryptosystems, encryption and decryption are **commutative**

That is, $D(E(m, k), k^{-1}) = E(D(m, k^{-1}), k) = m$

In such a system, we can use digital signatures as follows: *This is unforgeable*

- To sign a message, compute $D(m, k^{-1})$
- Transmit m and $D(m, k^{-1})$ to the recipient
- The recipient uses the sender's public key to verify that $E(D(m, k^{-1}), k) = m$

This provides verifiability



Question: Does encryption with a shared key have the same properties?

Features and Requirements

These features all require that for a given key pair (k, k^{-1}) , k can be made public and k^{-1} must remain secret

So, in a public key cryptosystem it must be

1. Computationally **easy** to encipher or decipher a message
2. Computationally **infeasible** to derive the private key from the public key
3. Computationally **infeasible** to determine the private key using a **chosen plaintext attack**

Informally, **easy** means “polynomial complexity”, while **infeasible** means “no easier than a brute force search”

How do public key cryptosystems work?

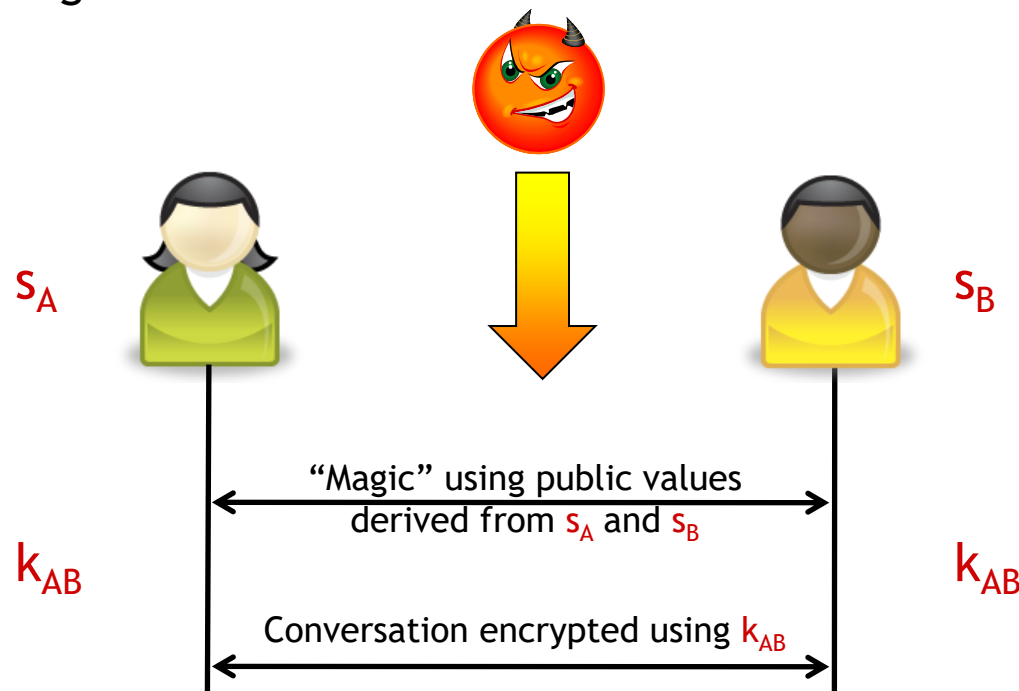
Diffie and Hellman proposed* the notion of public key cryptography

Diffie and Hellman **did not** succeed in developing a full-fledged public key cryptosystem

- i.e., their system cannot be used to encrypt/decrypt documents directly
- Rather, it allows two parties to agree on a shared secret using an entirely public channel

Question: Why is this an interesting problem to solve?

- Key exchange!



Diffie and Hellman proposed their system in 1976

Seminal paper: Whitfield Diffie and Martin E. Hellman, “New Directions in Cryptography,” IEEE Transactions on Information Theory (22)6 : 644 - 654, Nov. 1976

Problem: The widening use of telecommunications coupled with the key distribution problems inherent with secret key cryptography point to the fact that current solutions are **not** scalable!

This paper accomplishes many things:

- Clearly articulates why the key distribution problem must be solved
- Motivates the need for digital signatures
- Presents the first public key cryptographic algorithm
- Opened the “challenge” of designing a general-purpose public key cryptosystem

Variants of the Diffie-Hellman key exchange algorithm are widely used today!



How does the Diffie-Hellman protocol work?

Step 0: Alice and Bob agree on a finite cyclic group G of (large) prime order q , and a generator g for this group. This information is all **public**.

a is Alice's private key

$g^a \pmod{q}$ is Alice's public key

Step 1:

- Randomly choose $a \in \{1, 2, \dots, q-1\}$
- Compute $g^a \pmod{q}$
- Send $g^a \pmod{q}$



$g^a \pmod{q}$

$g^b \pmod{q}$

Step 2:

- Randomly choose $b \in \{1, 2, \dots, q-1\}$
- Compute $g^b \pmod{q}$
- Send $g^b \pmod{q}$

Step 3:

- Compute $(g^b \pmod{q})^a \pmod{q} = g^{ba} \pmod{q} = K_{ab}$

Step 3':

- Compute $(g^a \pmod{q})^b \pmod{q} = g^{ab} \pmod{q} = K_{ab}$

Why is the Diffie-Hellman key exchange protocol safe?

Recall: We need to show that it is hard for a “bad guy” to learn any of the secret information generated by this protocol, assuming that they know all public information

Public information: $G, g, q, g^a \pmod{q}, g^b \pmod{q}$

Private information: $a, b, K_{ab} = g^{ab} \pmod{q}$

Tactic 1: Can we get $g^{ab} \pmod{q}$ from $g^a \pmod{q}$ and $g^b \pmod{q}$?

- We can get $g^{am+bn} \pmod{q}$ for arbitrary m and n , but this is no help...

Tactic 2: Can we get a from $g^a \pmod{q}$?

- This called taking the discrete logarithm of $g^a \pmod{q}$
- The discrete logarithm problem is **widely believed** to be very hard to solve in **certain types** of cyclic groups

Conclusion: If solving the discrete logarithm problem is hard, then the Diffie-Hellman key exchange is secure!

Hm, interesting...

Recall from previous lectures that:

- Block ciphers secure data through confusion and diffusion
 - Designing block cipher mechanisms is equal parts art and science
 - The security of a block cipher is typically accepted over time (**Assurance!**)
 - Recall the initial skepticism over DES
 - The NIST competitions promote this as well
-

In public key cryptography, the relationship between k and k^{-1} is intrinsically mathematical

Result: The security of these systems is also rooted in mathematical relationships, and proofs of security involve reductions to mathematically “hard” problems

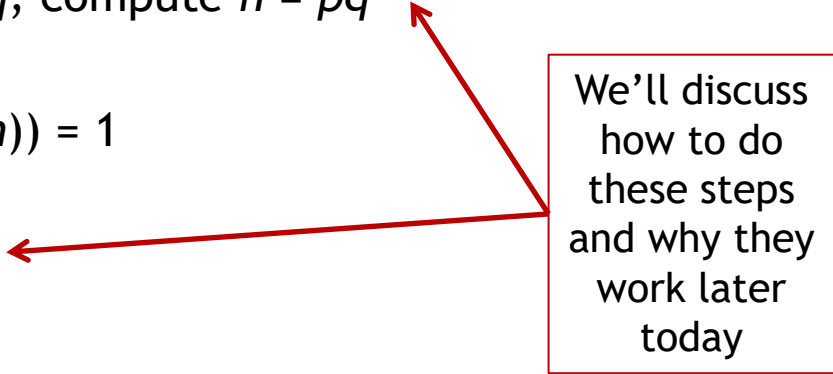
- e.g., Diffie-Hellman safe if the discrete logarithm is hard

The RSA cryptosystem picks up where Diffie and Hellman left off

RSA was proposed* by Ron Rivest, Adi Shamir, and Leonard Adelman in 1978. It can be used to encrypt/decrypt and digitally sign arbitrary data!

Key generation:

- Choose two large prime numbers p and q , compute $n = pq$
- Compute $\phi(n) = (p-1)(q-1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- **Public key:** n, e
- **Private key:** p, q, d

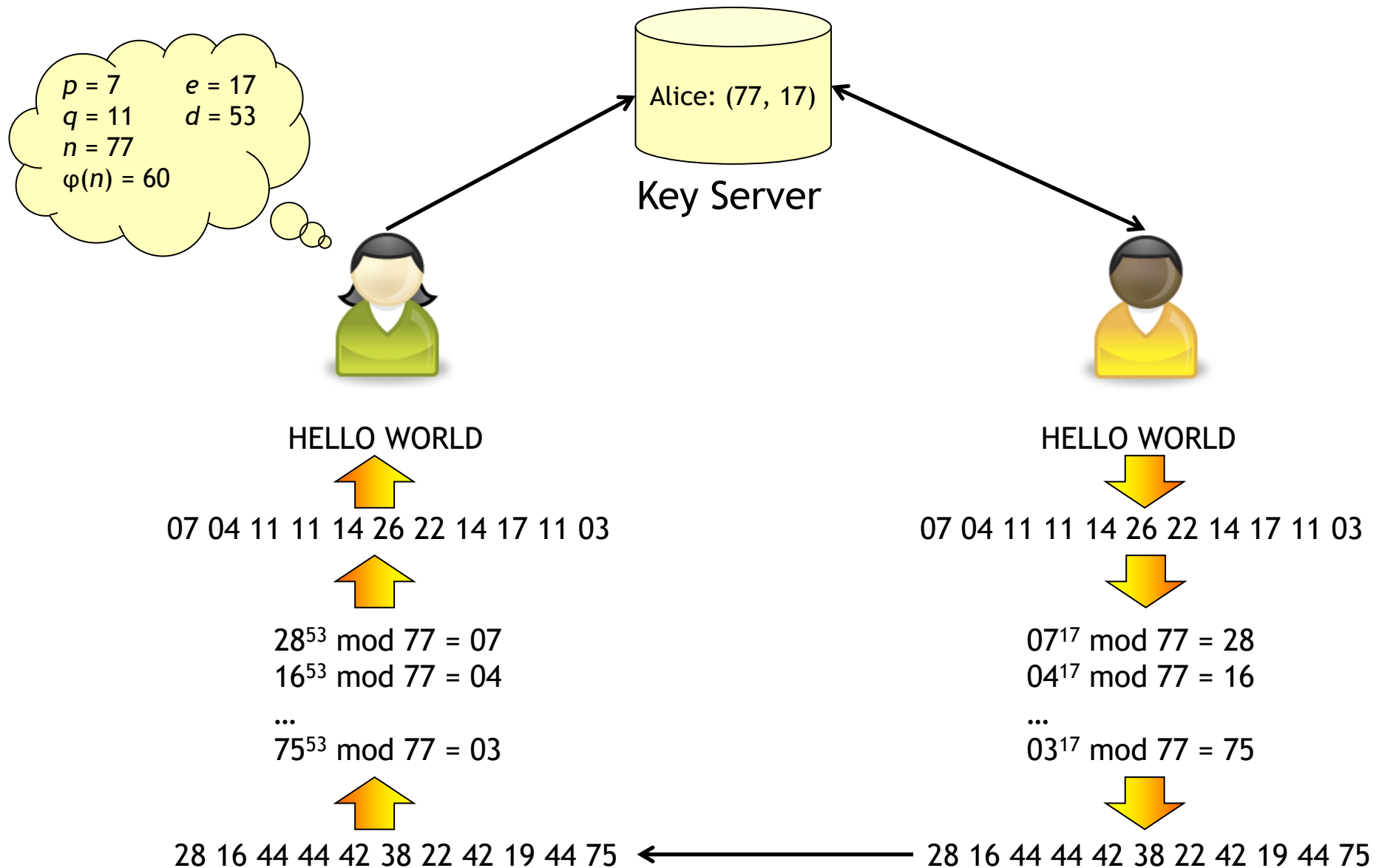


We'll discuss how to do these steps and why they work later today

Usage:

- Encryption: $M^e \pmod n$
- Decryption: $C^d \pmod n = M^{ed} \pmod n = M^{k\phi(n) + 1} \pmod n = M^1 \pmod n = M$

An RSA Example



What is involved in breaking RSA?

To break RSA, an attacker would need to derive the decryption exponent d from the public key (n, e)

Mathematicians think that this is a hard problem

This is **conjectured** to be as hard as factoring n into p and q . Why?

- Given p , q , and e , we can compute $\phi(n)$
- This allows us to compute d easily!

But what if there is some entirely unrelated way to derive d from the public key (n, e) ?

Question: Should this make you uneasy? Why or why not?

My Answer: Probably not, since this bizarre new attack would also have applications to factoring large numbers.

As always, nothing is really that easy...

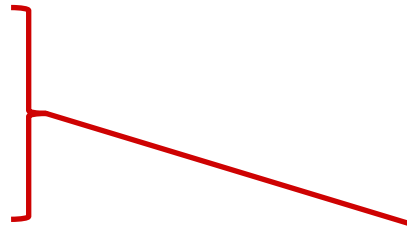
The bad news: Naive implementations of RSA are vulnerable to chosen ciphertext attacks

The good news: These attacks can be prevented by using a padding scheme like OAEP prior to encryption

Don't implement cryptography yourself! Use a standardized implementation, and verify that it is standards compliant.

Lastly, don't forget that implementations can be subjected to attacks

- Timing attacks
- Power consumption attacks
- Etc...

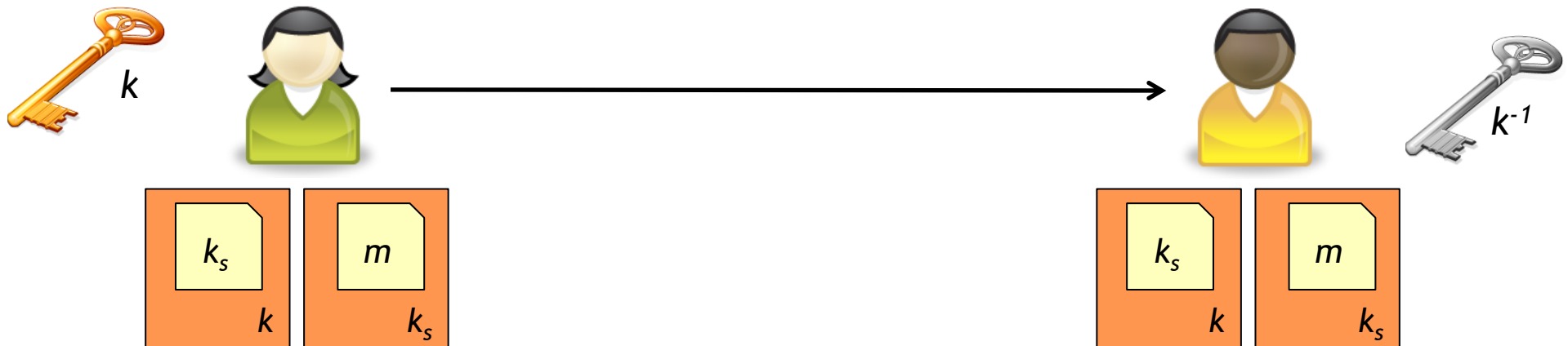


More on this later

Unfortunately, RSA is slow when compared to symmetric key algorithms like AES or HMAC-X

Using RSA as part of a **hybrid cryptosystem** can speed up **encryption**

- Generate a symmetric key k_s
- Encrypt m with k_s
- Use RSA to encrypt k_s using public key k
- Transmit $E_{k_s}(m)$, $E_k(k_s)$




Using hash functions can help speed up **signing** operations

- **Intuition:** $H(m) \ll m$, so signing $H(m)$ takes far less time than signing m
- Why is this safe? H 's **preimage resistance** property!

Some public-key systems have an interesting property known as malleability

Informally, a **malleable** cryptosystem allows meaningful modifications to be made to ciphertexts without revealing the underlying plaintext

- E.g., $E(x) \otimes E(y) = E(x + y)$



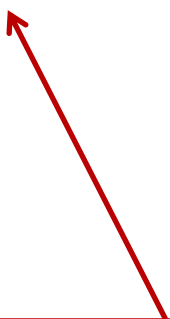
See: Pascal Paillier, Public-Key Cryptosystems Based on Composite Degree Residuosity Classes, EUROCRYPT 1999, pages 223-238.

MacKenzie et al. define a **tag-based** cryptosystem

- Messages encrypted relative to a key and a tag
- Only messages with the same tag can be combined

For example:

- $E(m, t) \otimes E(m', t) = E(mm', t)$
- $E(m, t) \otimes E(m', t') = \text{<garbage>}$



See: Philip MacKenzie, Michael K. Reiter, and Ke Yang, “Alternatives to Non-malleability: Definitions, Constructions, and Applications (Extended Abstract)”, TCC 2004, pages 171-190.

Discussion

Question 1: Why might malleability be an **interesting** property for a cryptosystem to have?

- Tallying electronic votes
- Aggregating private values
- A primitive for privacy-preserving computation
- ...

Question 2: Why might this be **bad**?

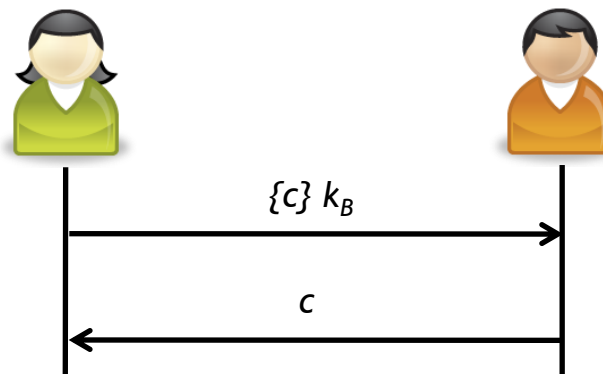
- Modifications by an active attacker!
- **Example:** Modifying an encrypted payment

In short, these types of cryptosystems have interesting properties, but require care to use properly.

Note that public key cryptography allows us to prove knowledge of a secret **without** revealing that secret

Example: Decrypting a challenge

1. Pick challenge c at random
2. Encrypt c using Bob's public key k_B
3. Transmit



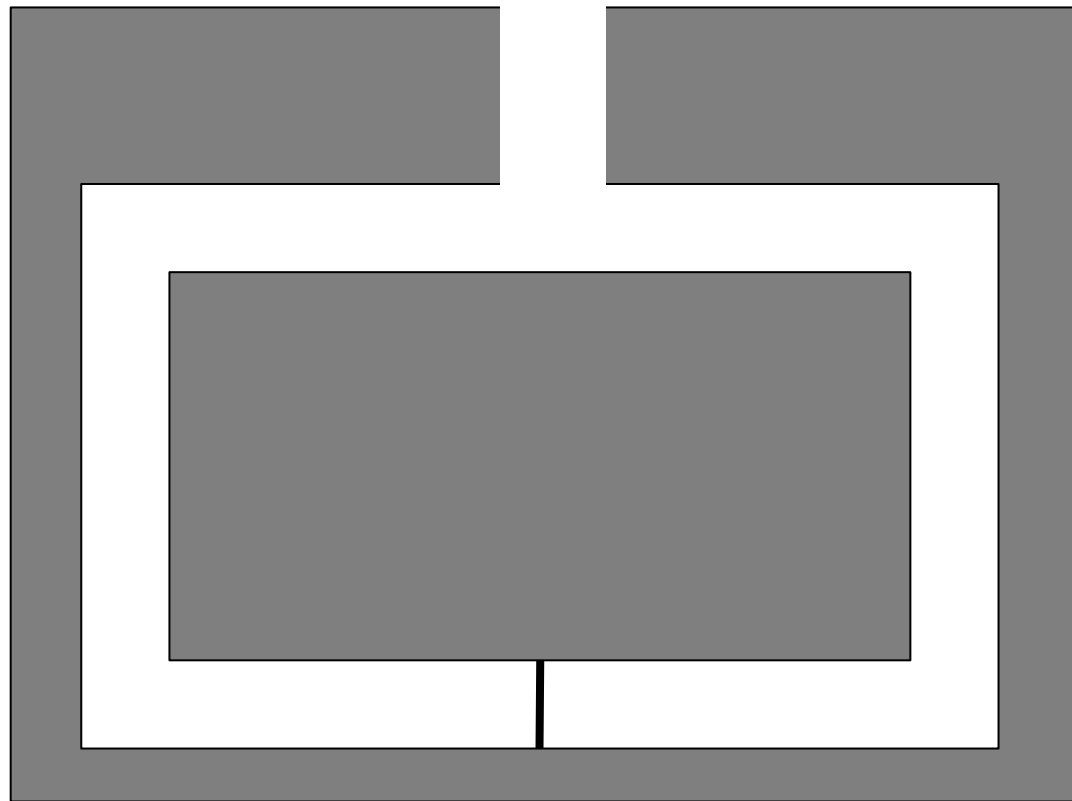
1. Decrypt c using private key k_B^{-1}
2. Transmit

Note: Revealing the challenge, c , does not leak information about the private key k_B^{-1} , yet Alice is (correctly) convinced that Bob knows k_B^{-1}

This type of protocol is called a **zero knowledge** protocol

Zero-knowledge proofs are easy to understand: A children's puzzle

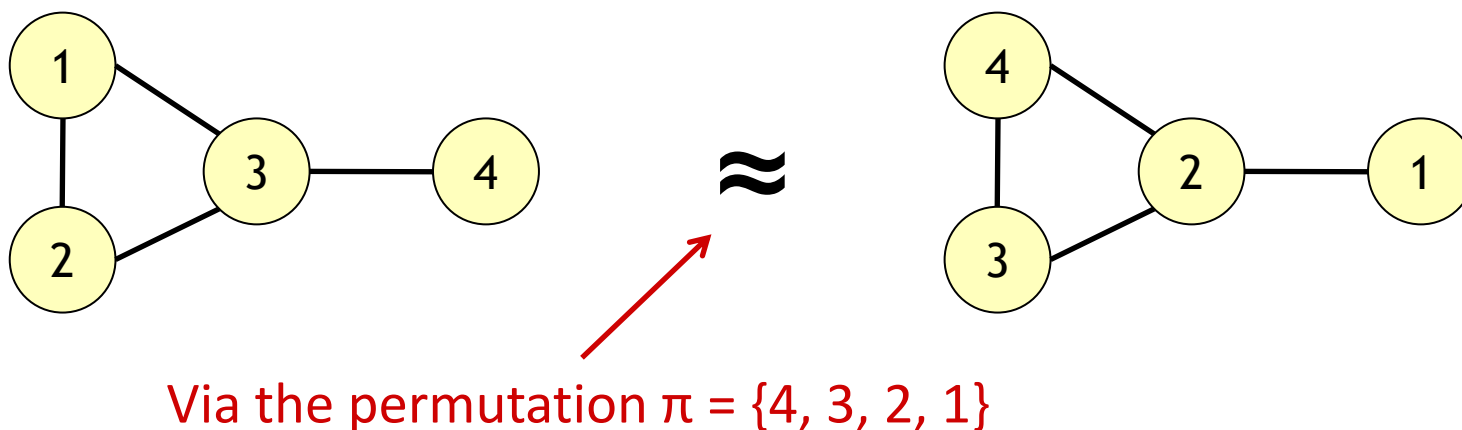
Example: The secret cave



Note: To ensure correctness, this “protocol” needs to be run multiple times (**Why?**)

We can construct a more realistic ZK system based on the (hard) problem of determining graph isomorphism

Informally, two graphs are **isomorphic** if the only difference between them is the names of their nodes



Determining whether two graphs are isomorphic is in NP, but the best known algorithm is $2^{O(\sqrt{n \log n})}$. This means that if the graphs are large, solving this problem will take a long time, but checking a solution is very easy.

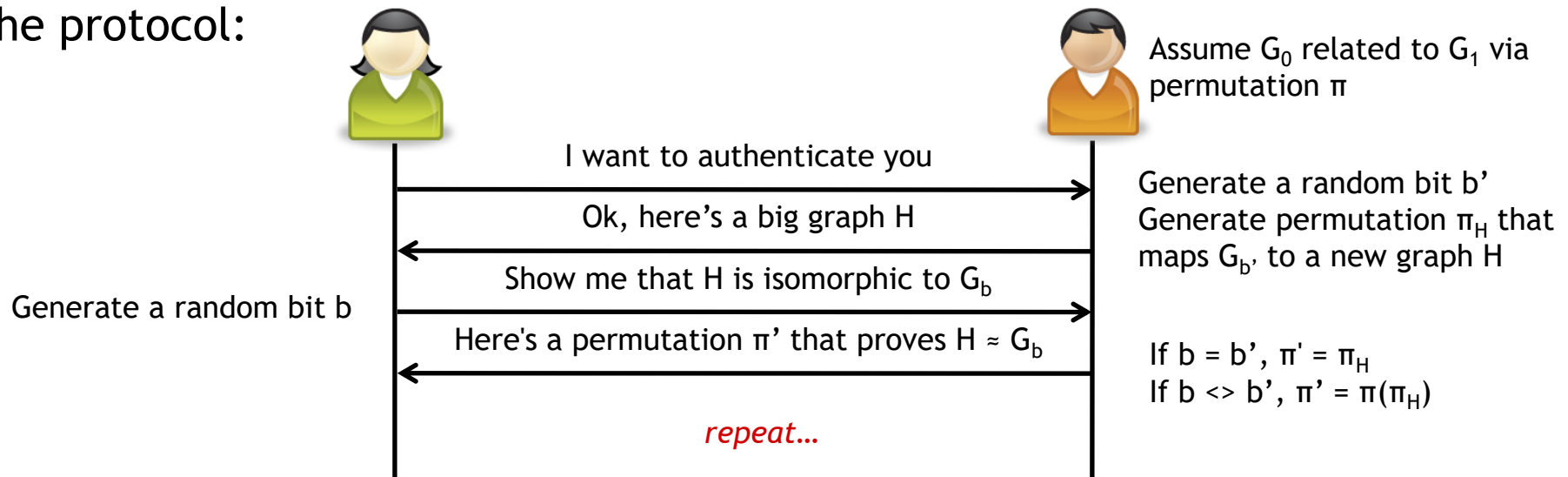
Authenticating via Graph Isomorphism

Our protocol has fairly simple parameters

- **Public key:** Two (big) isomorphic graphs G_0 and G_1
- **Private key:** The permutation mapping $G_0 \rightarrow G_1$

How do we find these efficiently?

The protocol:



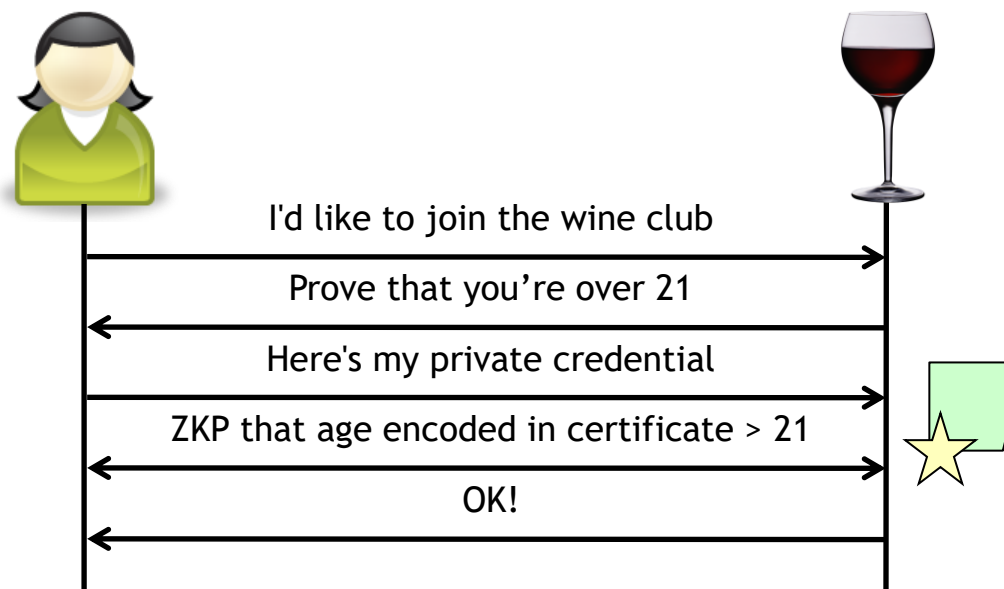
Why does this work?

- Answering this once means that Bob knows (at least) the permutation mapping from G_b to H .
- Doing this m times means that Bob knows the mapping between G_0 and G_1 with probability $1 - 0.5^m$
- Note that this leaks no information regarding the permutation π (Why?)

Zero knowledge proofs of knowledge can be used to solve a variety of interesting **authorization** problems

Private/Anonymous credential systems allow users to prove that they have certain attributes without actually revealing these attributes

Example: Purchasing wine over the Internet



The private credential scheme proposed by Stefan Brands enables many types of attribute properties to be checked in a zero-knowledge fashion

Summary so far ...

Secret key cryptography has a key distribution problem

Public key cryptography overcomes this problem!

- Public encryption key
- Private decryption key

Digital signatures provide both **integrity** protection and **non-repudiation**

Malleable cryptosystems are useful, but their usage entails certain risks

Zero knowledge proof systems have many interesting applications

Next: *Really* understanding RSA

Didn't we learn about RSA?

We saw **what** RSA does and learned a little bit about how we can use those features

Our goal will be to explore

- Why RSA actually works
- Why RSA is efficient* to use
- Why it is reasonably safe to use RSA

In short, it's time for more details ...

Note: Efficiency is a relative term 😊

RSA Overview / Roadmap

How do we choose large, pseudo-random primes?!

Key generation:

- Choose two large prime numbers p and q , compute $n = pq$
- Compute $\phi(n) = (p - 1)(q - 1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- **Public key:** n, e
- **Private key:** p, q, d

Why is $\phi(n) = (p - 1)(q - 1)$

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

Before we can do anything, we need a few large, pseudo-random primes

If our numbers are small, primality testing is pretty easy

- Try to divide n by all numbers less than \sqrt{n}
- The Sieve of Eratosthenes is a general extension of this principle

RSA requires **big** primes, so brute force testing is not an option (**Why?**)

To choose the types of numbers that RSA needs, we instead use a **probabilistic primality testing** method test : $\mathbb{Z} \times \mathbb{Z} \rightarrow \{T, F\}$

- test(n, a) = F means that n is composite based on the witness a
- test(n, a) = T means that n is **probably** prime based on the witness a

To test a number n for primality:

1. Randomly choose a witness a
2. if test(n, a) = F, n is composite
3. if test(n, a) = T, loop until we're reasonably certain that n is prime

Often with probability $\approx 1/2$



k repetitions means $P[n \text{ composite}] = 2^{-k}$



Fermat's little theorem can help us!

Fermat's little theorem: Given a prime number p and a natural number a such that $1 \leq a < p$, then $a^{p-1} \equiv 1 \pmod{p}$

How does this help with primality testing?

- If $a^{p-1} \not\equiv 1 \pmod{p}$, then p is **definitely** composite
- If $a^{p-1} \equiv 1 \pmod{p}$, then p is **probably** prime

Note: Some composite numbers will always pass this test (Yikes!)

- These are called Carmichael numbers
- Carmichael numbers are rare, but may still be found
- Other primality tests (e.g., Miller-Rabin) avoid detecting these numbers

This helps us test whether some number is prime. But how exactly does this help us generate RSA keys?

Putting it all together...

```
foundPrime = false
while (!foundPrime)
  let r = some large, odd, random number
  foundPrime = true
  for (iters = 0; iters < k; iters++)
    let a = random number less than r
    if ( $a^{r-1} \neq 1 \bmod r$ )
      foundPrime = false
      break
return r
```

The **prime number theorem** tells us that, on average, the number of primes less than n is approximately $n/\ln(n)$

- That is, $P[n \text{ prime}] \approx 1/\ln(n)$
- Searching for a prime is hard, but not ridiculously so

RSA Overview / Roadmap

How do we choose large, pseudo-random primes?!

Key generation:

- ✓ Choose two large prime numbers p and q , compute $n = pq$
- Compute $\phi(n) = (p - 1)(q - 1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- **Public key:** n, e
- **Private key:** p, q, d

Why is $\phi(n) = (p - 1)(q - 1)$

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

$\varphi(n)$ is called Euler's totient function

Definition: The totient function, $\varphi(n)$, counts the number of elements less than n that are **relatively prime** to n

For an RSA modulus $n = pq$, calculating $\varphi(n)$ is actually pretty simple

Consider each of the pq numbers $\leq n$

- All multiples of p share a common factor with n
 - There are q such numbers $\{p, 2p, 3p, \dots, qp\}$
- Similarly, all multiples of q share a common factor with n
 - There are p such numbers $\{q, 2q, 3q, \dots, pq\}$
- So, we have that $\varphi(n) = pq - p - q + 1$
- As a result, $\varphi(n) = pq - p - q + 1 = (p - 1)(q - 1)$

The +1 controls for subtracting pq twice

Note: Calculating $\varphi(n)$ is easy because we know how to factor n !

RSA Overview / Roadmap

Key generation:

- ✓ • Choose two large prime numbers p and q , compute $n = pq$
- ✓ • Compute $\phi(n) = (p - 1)(q - 1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- **Public key:** n, e
- **Private key:** p, q, d

Why is $\phi(n) = (p - 1)(q - 1)$

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

Review of greatest common divisors

Definition: Let a and b be integers, not both zero. The largest integer d such that $d \mid a$ and $d \mid b$ is called the **greatest common divisor** of a and b , and is denoted by $\gcd(a, b)$

Note: We can (naively) find GCDs by comparing the common divisors of two numbers.

Example: What is the GCD of 24 and 36?

- Factors of 24: 1, 2, 3, 4, 6, 12
- Factors of 36: 1, 2, 3, 4, 6, 9, 12, 18
- $\therefore \gcd(24, 36) = 12$

Wait. Aren't we dealing with numbers that are hard to factor?

Luckily, computing GCDs is not all that hard...

Intuition: Rather than computing the GCD of two big numbers, we can instead compute the GCD of smaller numbers that have the same GCD!

Interesting observation: $\gcd(a, b)$ is the same as $\gcd(a - b, b)$

Wait, what?

First, we must show that $d|a \wedge d|b \rightarrow d|(a - b)$



- If $d|a$ and $d|b$, then $a = kd$ and $b = jd$
- Then $a - b = kd - jd = (k - j)d$
- So, $d|a \wedge d|b \rightarrow d|(a - b)$

Ok, so d is a divisor of $(a - b)$, but is it the **greatest** divisor?

- The **relevant** divisors of $(a - b)$ are a subset of the divisors of a and the divisors of b
- Since $d = \gcd(a, b)$, it is the greatest of the remaining divisors

Euclid's algorithm optimizes this process!

Euclid's algorithm finds $\gcd(a, b)$ as follows:

- Set $r_{-1} = a$, $r_{-2} = b$, $n = 0$
- While $r_{n-1} \neq 0$
 - divide r_{n-2} by r_{n-1} to find q_n and r_n such that $r_{n-2} = q_n r_{n-1} + r_n$
 - $n = n + 1$
- $\gcd(a, b) = r_{n-2}$

Example: Computing $\gcd(414, 662)$

n	q_n	r_n
-2	-	662
-1	-	414
0	1	248
1	1	166
2	1	82
3	2	2
4	41	0



That's all fine and good, but how does this help us compute $e \equiv 1 \pmod{\phi(n)}$?

Method 1: Use Euclid's algorithm

- Choose a random e
- Use Euclid's algorithm to determine whether $\gcd(e, \phi(n)) = 1$
- Repeat as needed

Method 2: We can just choose a large prime number, $r > \max(p, q)$

Why does method 2 work?

- r is a prime number, so it has no divisors other than itself and 1
- r is larger than p and q , so $r \neq p$ and $r \neq q$

RSA Overview / Roadmap

Key generation:

- ✓ • Choose two large prime numbers p and q , compute $n = pq$
- ✓ • Compute $\phi(n) = (p - 1)(q - 1)$
- ✓ • Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- **Public key:** n, e
- **Private key:** p, q, d

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

It turns out that Euclid's algorithm can help us compute $d \equiv e^{-1}$, too

If we maintain a little extra state, we can figure out numbers u_n and v_n such that $r_n = u_n a + v_n b$ when calculating $\gcd(a, b)$

If a and b are relatively prime, this will allow us to calculate a^{-1}

- $1 = u_n a + v_n b$ // If a and b are relatively prime, $r_n = 1$
- $u_n a = 1 - v_n b$ // Subtract $v_n b$ from both sides
- $u_n a \equiv 1 \pmod{b}$ // Definition of congruence
- So $u_n = a^{-1}$ // Definition of inverse

This makes $r_n = u_n a + v_n b$
for $n = -1$ and $n = -2$

The extended Euclid's algorithm works as follows:

- Set $r_{-1} = b$, $r_{-2} = a$, $n = 0$, $u_{-2} = 1$, $v_{-2} = 0$, $u_{-1} = 0$, $v_{-1} = 1$
- While $r_{n-1} \neq 0$
 - divide r_{n-2} by r_{n-1} to find q_n and r_n such that $r_{n-2} = q_n r_{n-1} + r_n$
 - $u_n = u_{n-2} - q_n u_{n-1}$
 - $v_n = v_{n-2} - q_n v_{n-1}$
 - $n = n + 1$
- $\gcd(a, b) = r_{n-2} = u_{n-2} a + v_{n-2} b$

How about an example?

Example: Find the inverse of 797 mod 1047

n	q_n	r_n	u_n	v_n
-2		797	1	0
-1		1047	0	1
0	0	797	1	0
1	1	250	-1	1
2	3	47	4	-3
3	5	15	-21	16
4	3	2	67	-51
5	7	1	-490	373

So, $1 = -490 \cdot 797 + 373 \cdot 1047$

- $-490 \cdot 797 = 1 + (-373)1047$
- $-490 \cdot 797 \equiv 1 \pmod{1047}$
- In other words, -490 is the inverse of 797 mod 1047

RSA Overview / Roadmap

Key generation:

- ✓ • Choose two large prime numbers p and q , compute $n = pq$
- ✓ • Compute $\phi(n) = (p - 1)(q - 1)$
- ✓ • Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- ✓ • Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- **Public key:** n, e
- **Private key:** p, q, d

This seems tricky, too

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Isn't this expensive?

Why does this work?

Isn't exponentiation really expensive?

Exponentiation can be sped up using a trick called **successive squaring**

```
int pow(int m, int e)
    if (e is even)
        return pow(m*m, e/2)
    else
        return m * pow(m, e - 1)
```

For example, consider computing 2^{15}

- Naive method: $2 * 2 * 2 * \dots * 2 = 32,768$

- Fast method: $2^{15} = 2 * 4^7$

$$= 2 * 4 * 4^6$$

$$= 2 * 4 * 16^3$$

$$= 2 * 4 * 16 * 16^2$$

$$= 2 * 4 * 16 * 256$$

$$= 32,768$$

$O(e)$ multiplications



$O(\log(e))$ multiplications



This only gets us partway there. Various other algorithmic tricks enable modulo exponentiation to be efficient!

RSA Overview / Roadmap

Key generation:

- ✓ • Choose two large prime numbers p and q , compute $n = pq$
- ✓ • Compute $\phi(n) = (p - 1)(q - 1)$
- ✓ • Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- ✓ • Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
 - **Public key:** n, e
 - **Private key:** p, q, d

Usage:

- ✓ • Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Isn't this expensive?



Why does this work?



Why *does* decryption work?

Note: Decryption will work if and only if $C^d \bmod n = M$

$$\begin{aligned} C^d \bmod n &= M^{ed} \bmod n \quad // \quad C = M^e \bmod n \\ &= M^{k\phi(n)+1} \bmod n \quad // \quad ed \equiv 1 \bmod \phi(n), \text{ so } ed = k\phi(n) + 1 \\ &= M^1 \bmod n \quad // \quad \text{?!?} \\ &= M \bmod n \quad // \quad M^1 = M \end{aligned}$$

The only hitch in showing the correctness of the decryption process is proving that $M^{k\phi(n)+1} \bmod n = M^1 \bmod n$

Fortunately, two smart guys can help us out with this...



Pierre de Fermat
160? - 1665



Leonhard Euler
1707 - 1783

First, we need to learn about the set Z_n^*

Definition: Z_n^* is the set of all integers relatively prime to n

Example: Z_{10}^*

\times	1	3	7	9
1	1	3	7	9
3	3	9	1	7
7	7	1	9	3
9	9	7	3	1

Interesting note: $\forall a, b \in Z_n^*: ab \in Z_n^*$

- a relatively prime to n means that $\exists u_1, v_1: u_1 a + v_1 n = 1$
- b relatively prime to n means that $\exists u_2, v_2: u_2 b + v_2 n = 1$
- Multiplying gives us $(u_1 u_2) ab + (u_1 v_2 a + v_1 u_2 b + v_1 v_2 n) n = 1$

The above states that Z_n^* is **closed under multiplication**

This leads us to something called Euler's theorem

Theorem: $\forall a \in Z_n^*: a^{\phi(n)} \equiv 1 \pmod n$

Proof:

- Multiply all $\phi(n)$ elements of Z_n^* together, calling the product x
- Note that $x \in Z_n^*$, and has an inverse x^{-1}
- Now, multiply each element of Z_n^* by a and multiply each of the resulting elements together. This will give us $a^{\phi(n)}x$
- Multiplying each element of Z_n^* actually just rearranges these elements.
- As a result, we have that $a^{\phi(n)}x = x$
- If we divide both sides of the equation by x , we get that $a^{\phi(n)} = 1$ \square

Ok, so what does Euler's theorem have to do with RSA?

We can restate Euler's theorem so that it more clearly connects to RSA math

Theorem: $\forall a \in Z_n^*, k \in Z^+ : a^{k\phi(n)+1} \equiv a \pmod n$

Proof: $a^{k\phi(n)+1} = a^{k\phi(n)}a = a^{\phi(n)k}a = 1^ka = a \quad \square$

From Euler's theorem!



Now, in RSA

- All of our math is done mod n
- Our message space is chosen from elements of Z_n^*
- $ed \equiv 1 \pmod{\phi(n)}$, so $ed = k\phi(n) + 1$ for some k
- $\therefore M^{ed} \pmod n = M^{k\phi(n)+1} \pmod n = M^1 \pmod n = M$

Decryption works!

