CMSC414 Computer and Network Security

Public Key Cryptography

Yizheng Chen | University of Maryland
surrealyz.github.io

Apr 4, 2024

Credits: original slides from Dave Levin

Agenda

- Diffie Hellman Key Exchange
- Public Key Cryptography
- Certificates
- Passwords

DIFFIE HELLMAN KEY EXCHANGE

HIGH-LEVEL REVIEW OF MODULAR ARITHMETIC

$x \bmod N$

g is a generator of mod N if
$\{1, 2, \ldots, N\text{-}1\} = \{g^0 \bmod N, g^1 \bmod N, \ldots, g^{N-2} \bmod N\}$

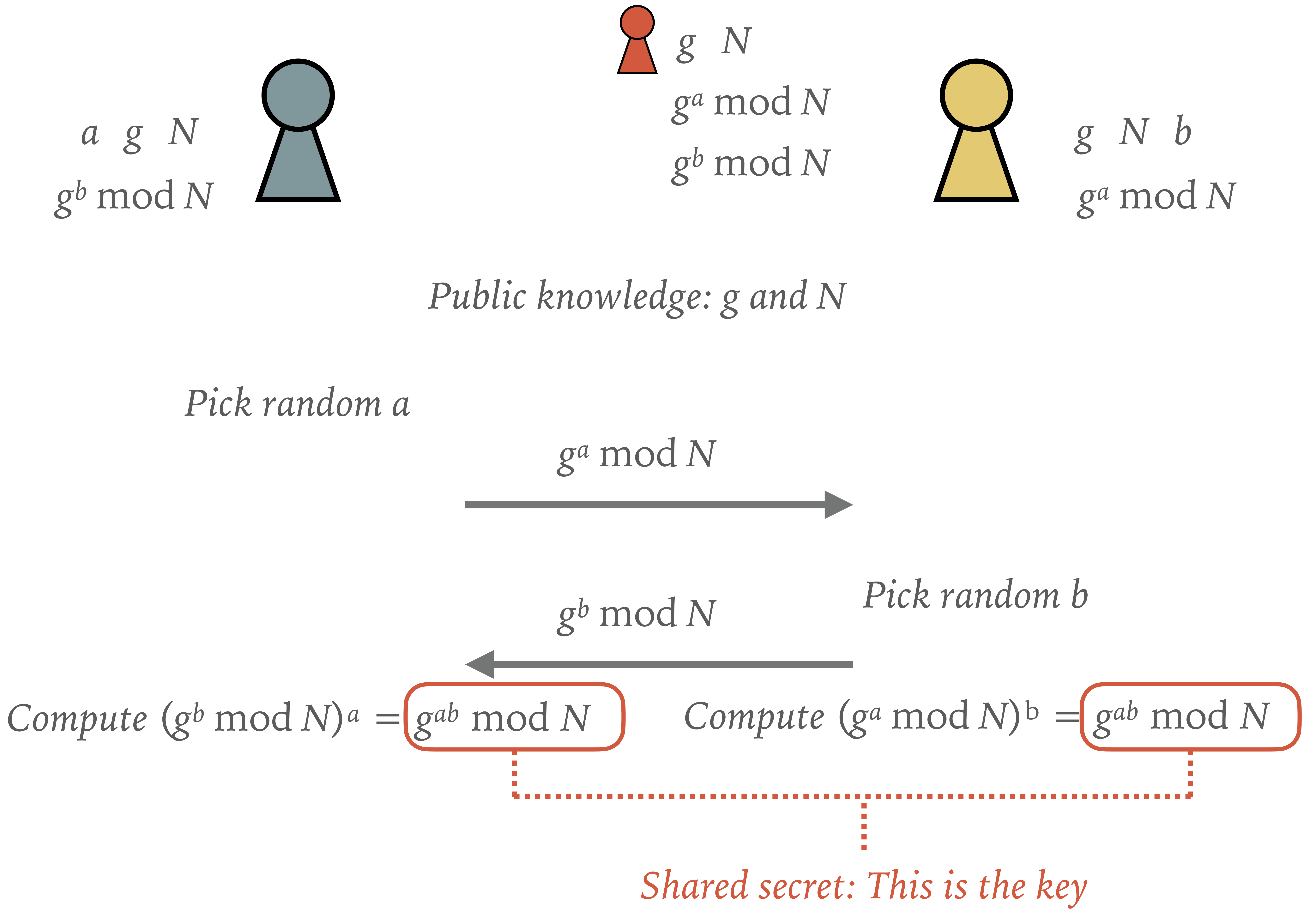
$N=5, g=3$
$3^0 \bmod 5 = 1$ $3^1 \bmod 5 = 3$ $3^2 \bmod 5 = 4$ $3^3 \bmod 5 = 2$

Given x and g, it is efficient to compute
$g^x \bmod N$

Given g and g^x, it is efficient to compute x
(simply take $\log_g g^x$)

Given g and g^x mod N it is *infeasible* to compute x
Discrete log problem

DIFFIE-HELLMAN KEY EXCHANGE



DIFFIE-HELLMAN KEY EXCHANGE

g N

$g^a \bmod N$

$g^b \bmod N$

$g^{ab} \bmod N$

Given g and $g^x \bmod N$ it is *infeasible* to compute x

Discrete log problem

Note that just multiplying g^a and g^b won't suffice:

$$g^a \bmod N \ * \ g^b \bmod N \quad = g^{a+b} \bmod N$$

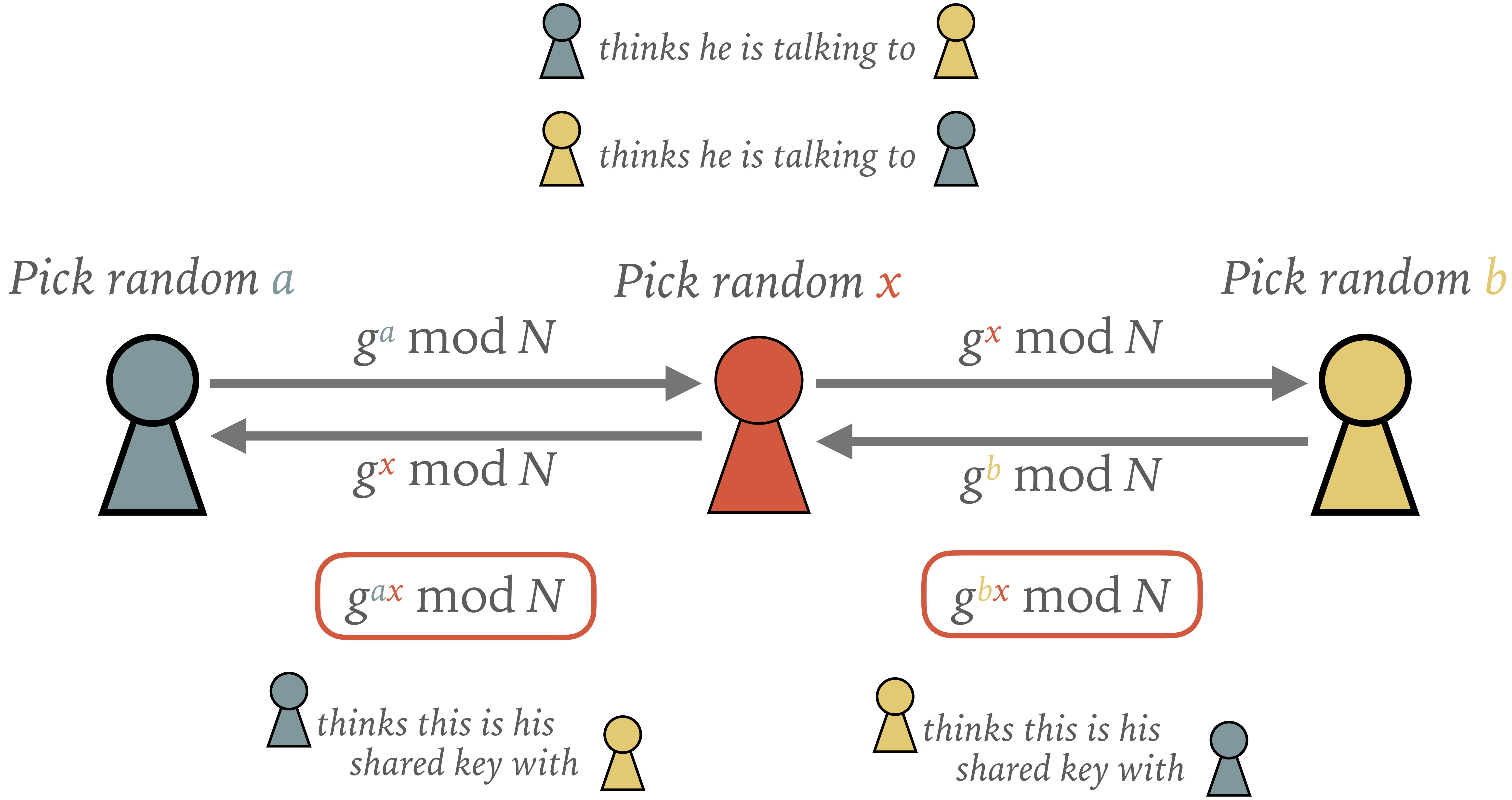
Key property:

An *eavesdropper* cannot infer the shared secret (g^{ab}).

But what about *active intermediaries*?

MAN-IN-THE-MIDDLE (MITM) ATTACKS

The attacker can interpose between the two communicating parties and insert, delete, and modify messages.



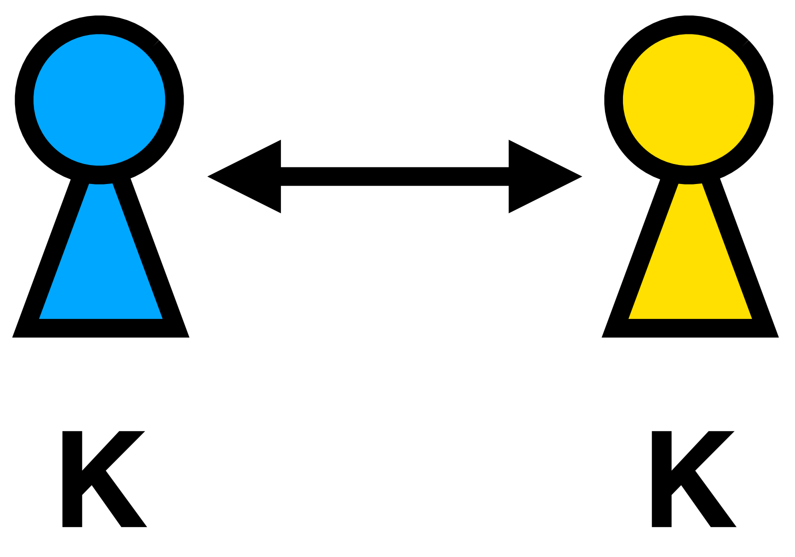
The attacker can now eavesdrop on the conversation.

Key property: Diffie-Hellman is *not* resilient to a MITM attack

TO FIX THIS PROBLEM WE NEED…

PUBLIC KEY CRYPTOGRAPHY

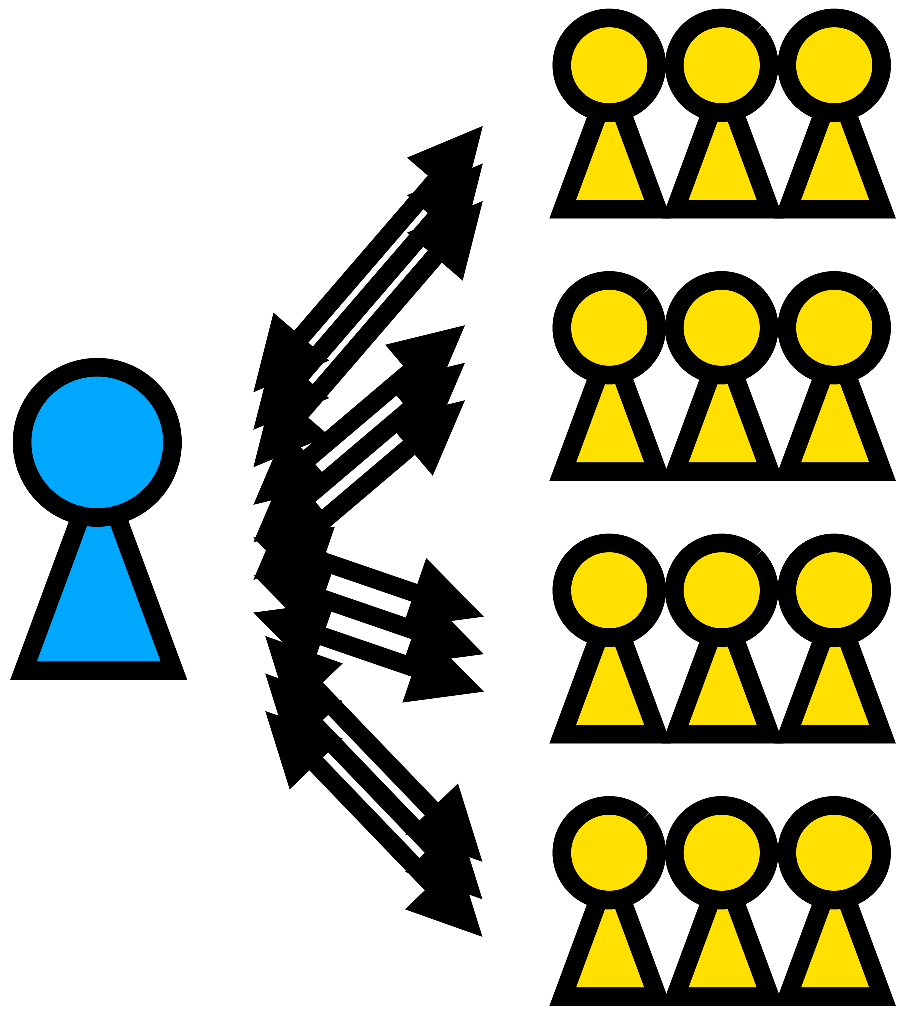
Shortcomings of symmetric key

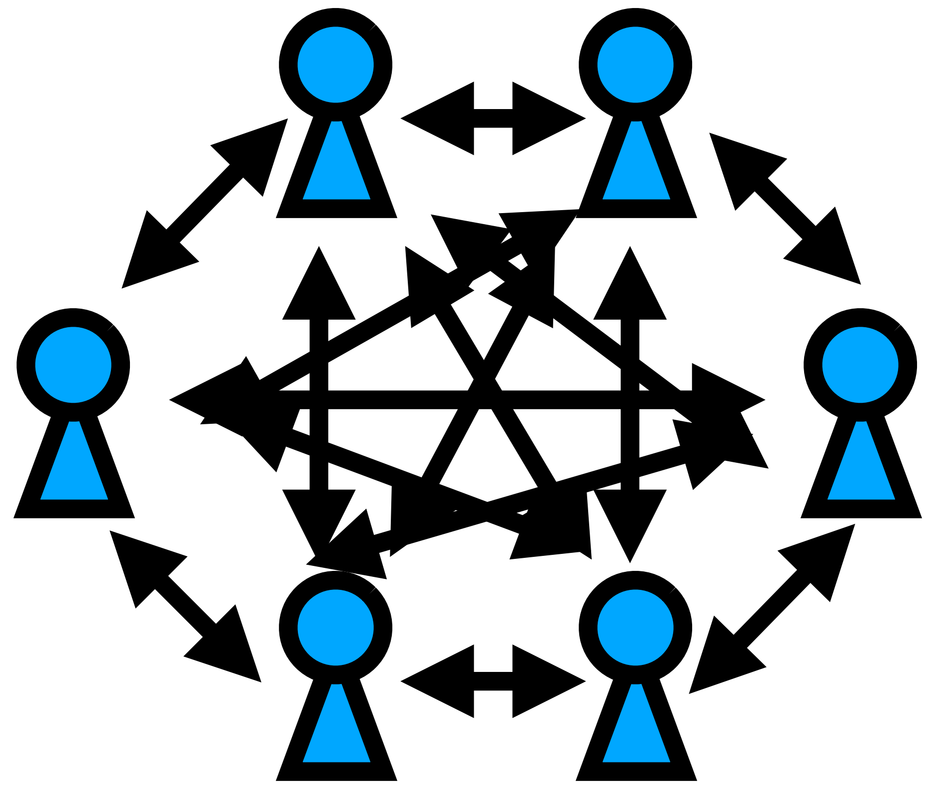


Establishing a pairwise key requires a **key exchange**, which requires both parties to be ***online***

Issue #1: Requires *pairwise* key exchanges

File downloads



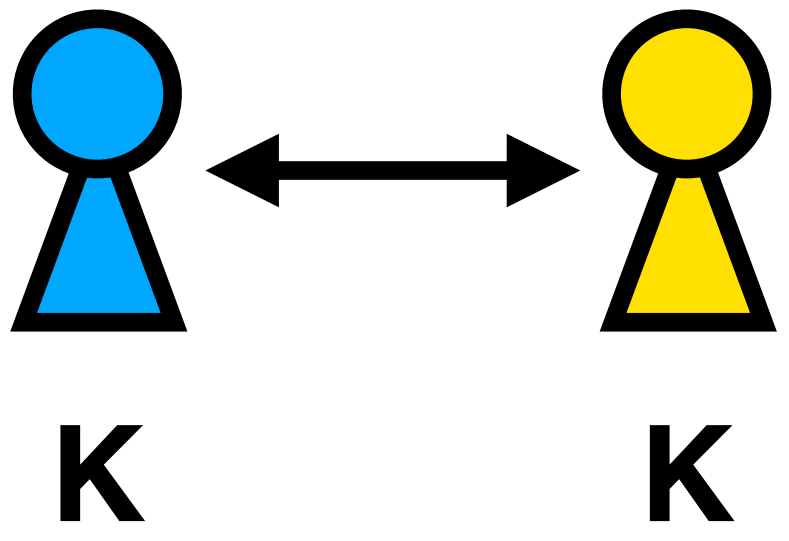


Email / chat

One-to-many: O(N) key exchanges

All-to-all: O(N^2) key exchanges

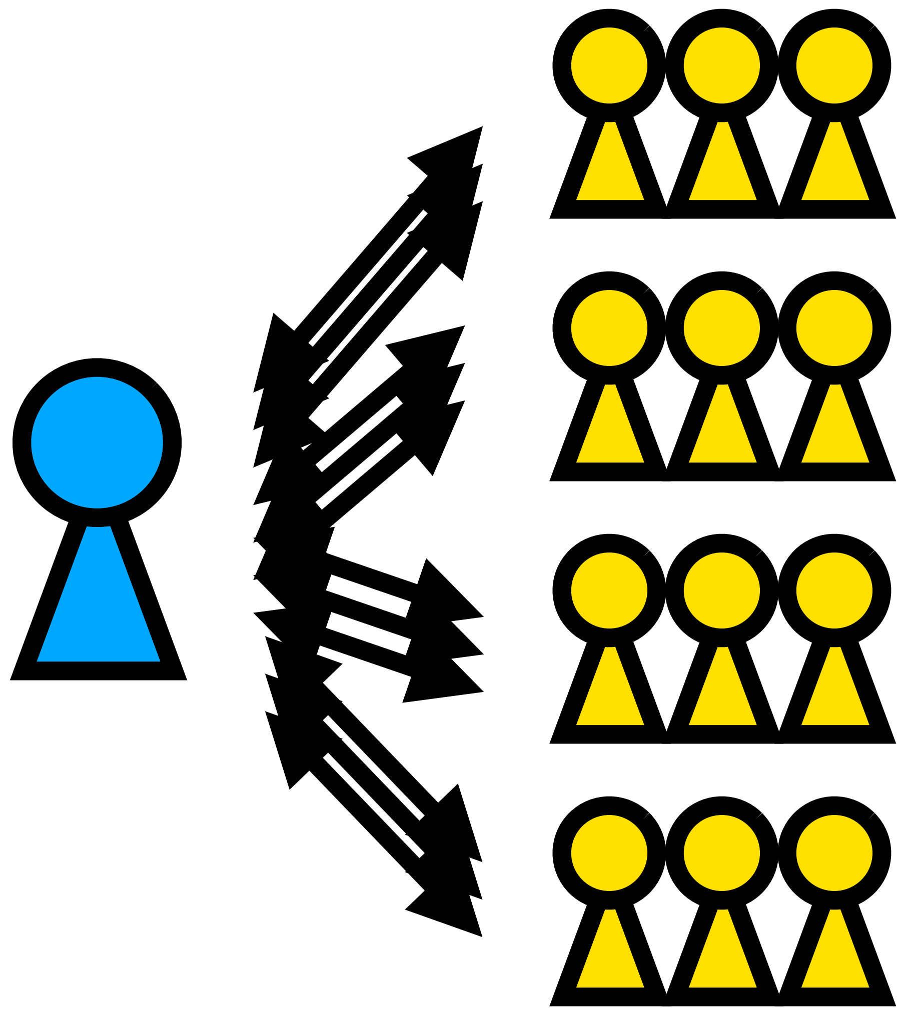
Shortcomings of symmetric key



Establishing a pairwise key requires a **key exchange**, which requires both parties to be ***online***

Issue #2: Parties must be online

File downloads

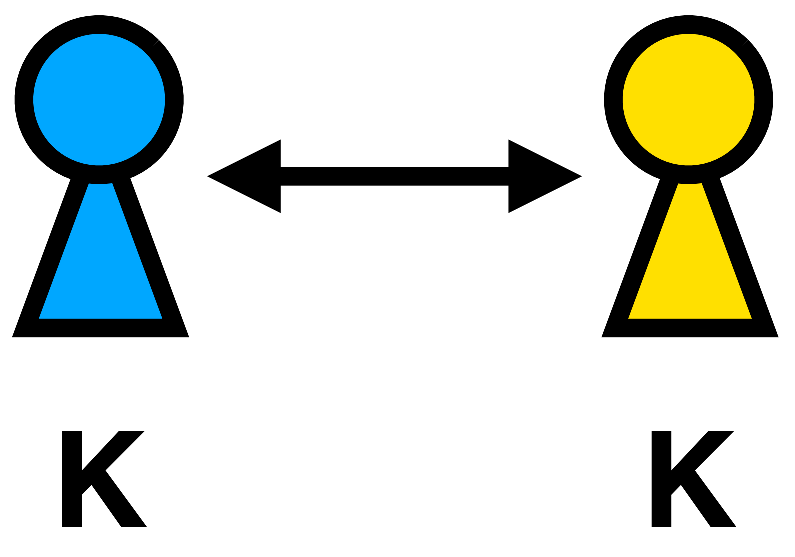


One-to-many:
O(N) key
exchanges

Blue user uploads a document, then goes offline (e.g., forever)

Later, a yellow user wants to get a copy; how can it know the copy is really from the blue user?

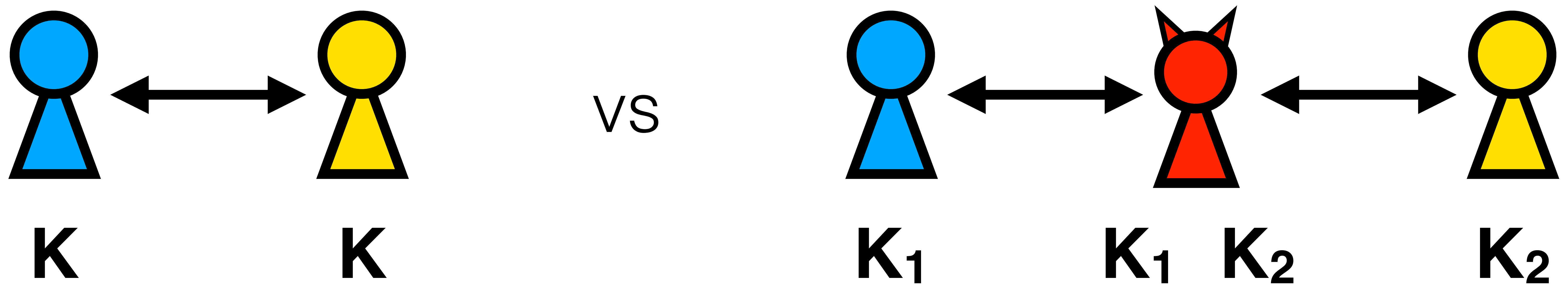
Shortcomings of symmetric key



Establishing a pairwise key requires a **key exchange**, which requires both parties to be ***online***

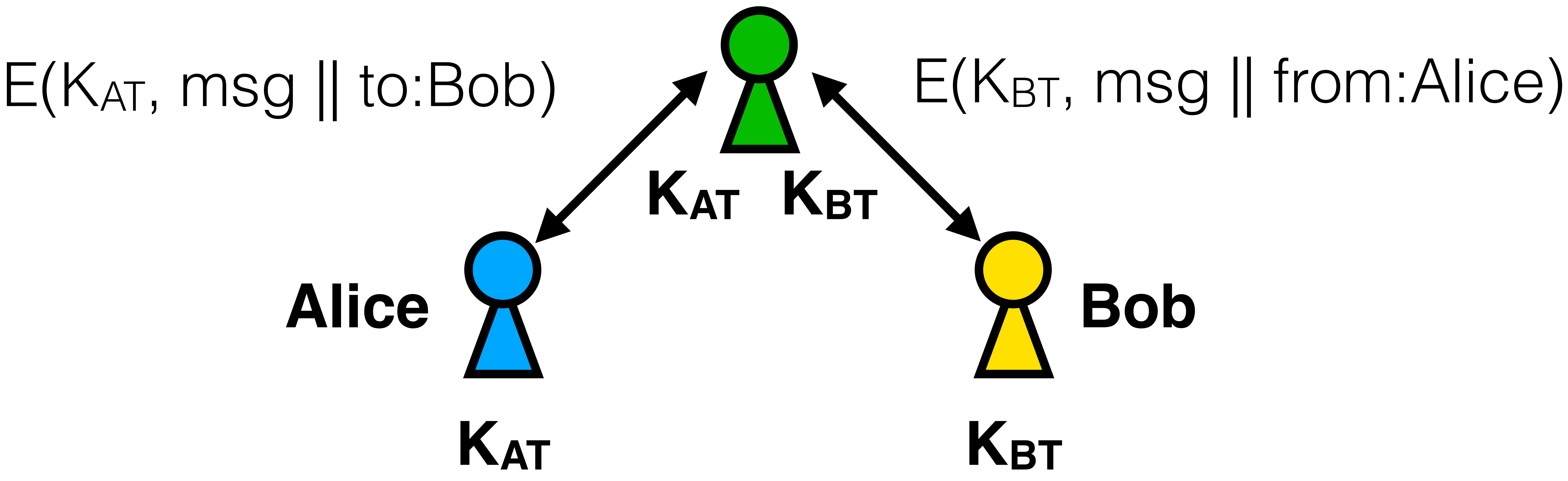
Issue #3: How do you know to whom you're talking?

Diffie-Hellman is resilient to *eavesdropping*,
but ***not tampering***



A protocol that solves this with ***trust***

Trent: A *trusted* third party



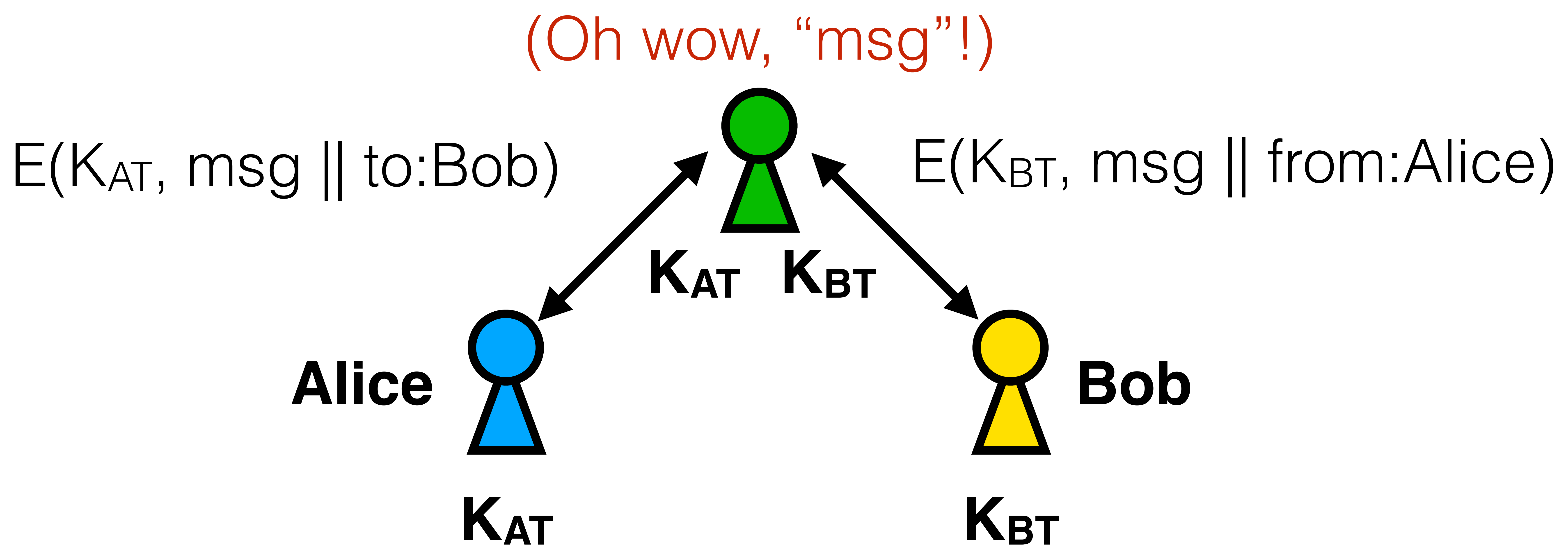
1. Everybody establishes a pairwise key with Trent
Good: *O(N) key exchanges*

2. Trent validates each user's identity; includes in message
Good: *Authenticated communication*

Bad: All messages get sent through Trent

What are we trusting Trent not to do?

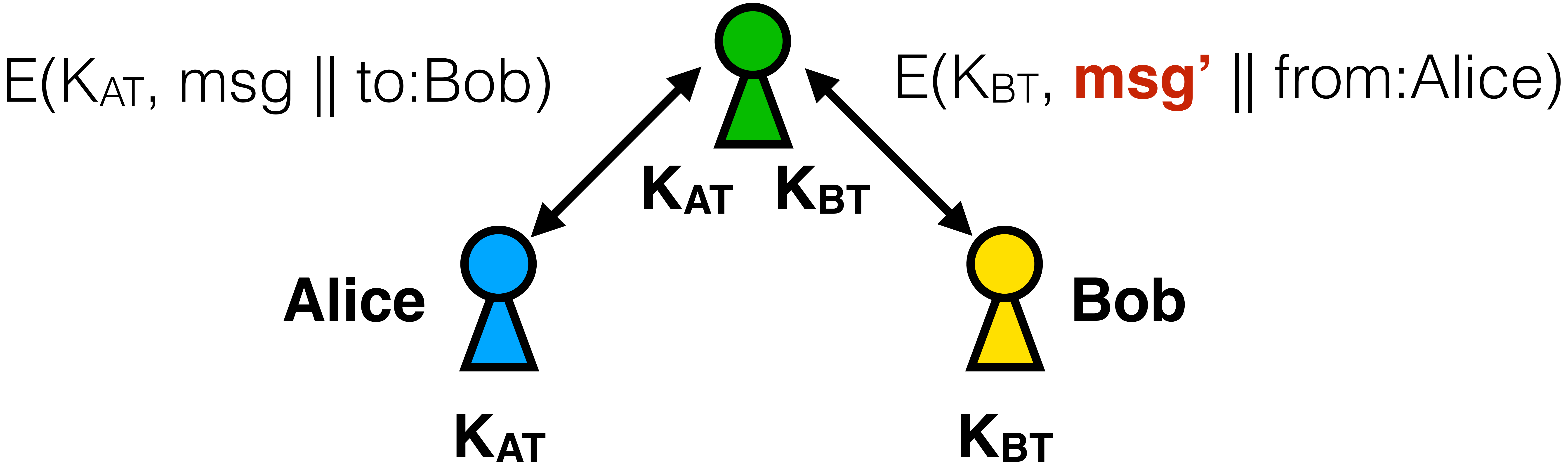
Just as "secure" meant nothing without an attack model, "trusted" means nothing without a **trust model**



1. Do not *read* messages

What are we trusting Trent not to do?

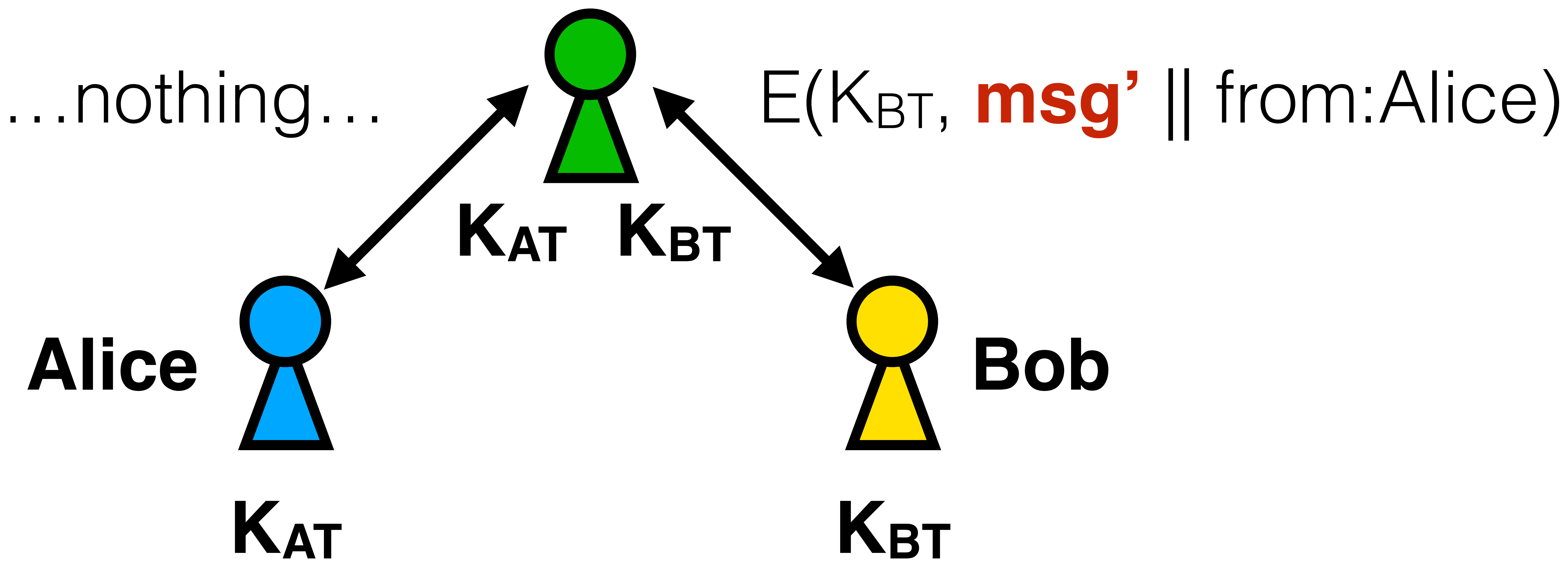
Just as “secure” meant nothing without an attack model, “trusted” means nothing without a **trust model**



1. Do not *read* messages
2. Do not *alter* messages

What are we trusting Trent not to do?

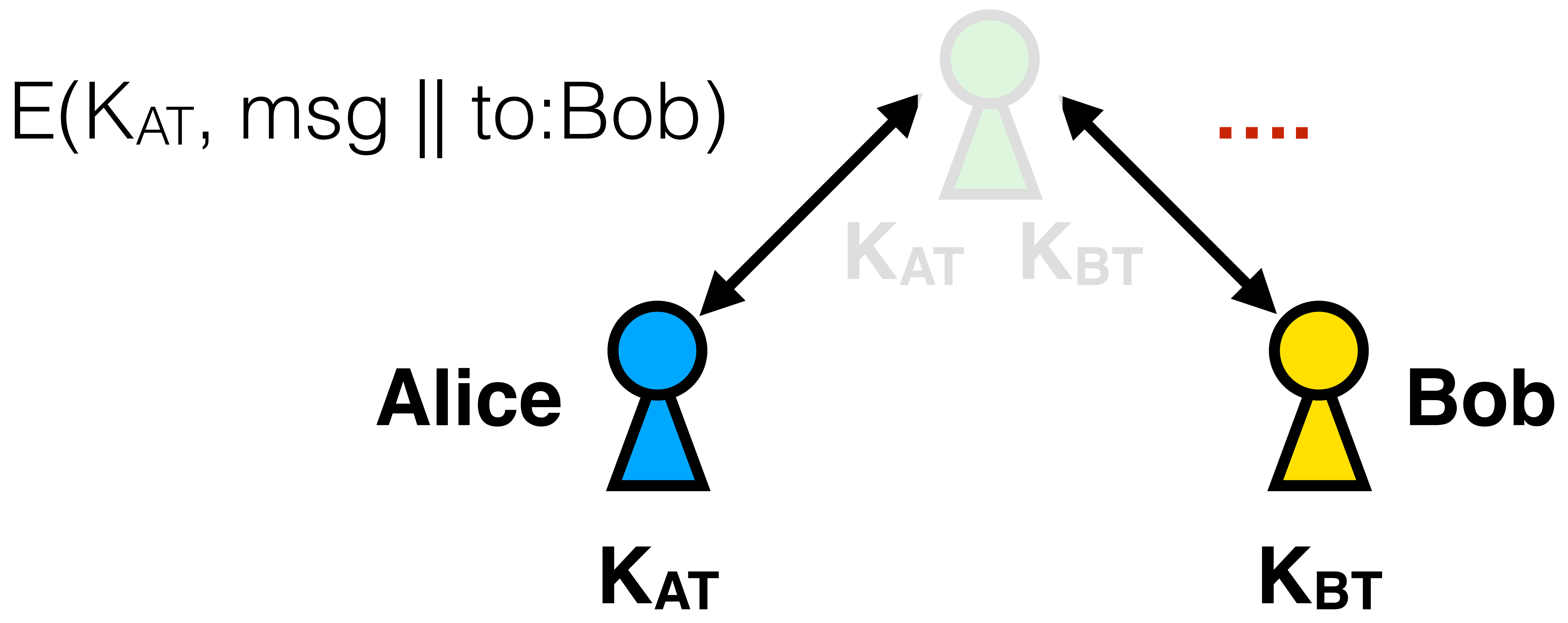
Just as "secure" meant nothing without an attack model, "trusted" means nothing without a **trust model**



1. **Do not *read* messages**
2. **Do not *alter* messages**
3. **Do not *forge* messages**

What are we trusting Trent not to do?

Just as "secure" meant nothing without an attack model,
"trusted" means nothing without a **trust model**



1. Do not *read* messages
2. Do not *alter* messages
3. Do not *forge* messages
4. Do not *go offline*

Public key encryption

A public key encryption scheme comprises three algorithms

Key generation **G**

- Inputs
  - Source of randomness
  - Maximum key length L
- Outputs: a *key pair*
  - *PK* = **public key**
  - *SK* = **secret key**

This is a *randomized* algorithm
(nondeterministic output)

Difficult to infer SK from PK
Only one person should know SK;
PK should be public to *all*

PK and SK are intrinsically bound together:
for a given PK, there is a single *corresponding* SK

Example: RSA's public keys are a pair: (exponent, modulus)

Public key encryption

A public key encryption scheme comprises three algorithms

Encryption **E(PK, msg)**

- Inputs
  - **Public** key PK
  - Message msg of *fixed size*
- Outputs: a cipher text *c* *same size as msg*

This is a *randomized* algorithm
(vanilla RSA is deterministic; in practice, RSA-PKCS is used instead, which adds a nonce to the message)

PK a.k.a. "Encryption key"

Anyone who knows Alice's PK can encrypt a message to her…

Public key encryption

A public key encryption scheme comprises three algorithms

[Decryption **D(SK, c)**]
- Inputs
  - **Secret** key SK
  - Cipher text c
- Outputs: original msg

This is a *deterministic* algorithm

Should always return the original message

…but only Alice can decrypt that message

Public key encryption

A public key encryption scheme comprises three algorithms

Key generation G
→ *PK* = **public key**
→ *SK* = **secret key**

Encryption E(PK, m)
→ cipher text *c*

Decryption D(SK, c)
→ original msg

Correctness

D(SK, E(PK, m)) = m

Security

E(PK, m) should appear random (small change to (PK,m) leads to large changes to c)

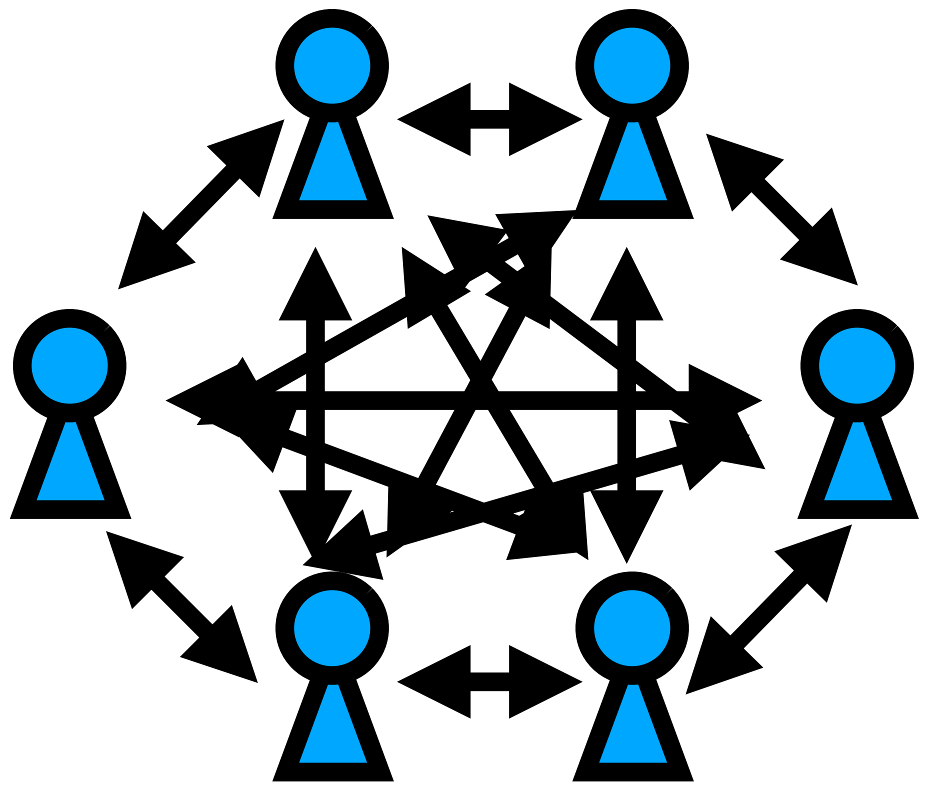
E() should approximate a one-way trapdoor function: cannot invert without access to SK

Protocols with public key encryption

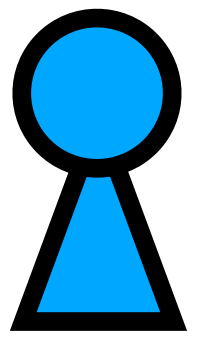
Goal: deliver a confidential message

Symmetric key

Email / chat

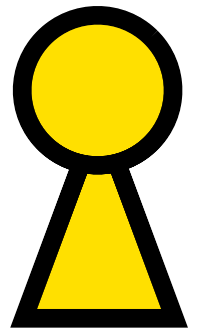


All-to-all:
$O(N^2)$ key
exchanges



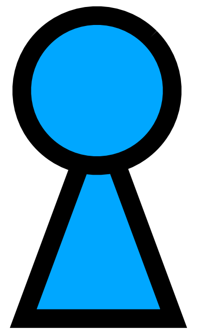
Generate public/private
key pair (PK,SK)

Annouce PK publicly
(on website, in newspaper, …)



Obtain PK

Send c = E(PK, msg)



Decrypt D(SK, c) = msg

O(N) keys in total

Overcoming fixed message sizes

Encryption E(PK, msg)
- Inputs
  - **Public** key PK
  - Message msg of *fixed size*
- Outputs: a cipher text *c* *same size as msg*

Like block ciphers, but there are not “modes” of public key encryption

Public key operations are *slooooow!*

Symmetric key operations are fast

Hybrid encryption

Generate public/private key pair (PK,SK); publicize PK

Obtain PK

Generate *symmetric* key K

Symm key Compute c_{msg} = e(K, msg)

Public key Compute c_K = E(PK, K) ***Now throw away K***

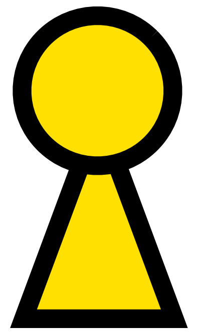
Send c_K || c_{msg}

Decrypt D(SK, c_K) = K *Public key*

Decrypt d(K, c_{msg}) = msg *Symm key*

Hybrid encryption

Obtain PK



Generate *symmetric* key K

Compute c_{msg} = e(K, msg)

Compute c_K = E(PK, K)

Send c_K || c_{msg}

The easy key distribution of public key

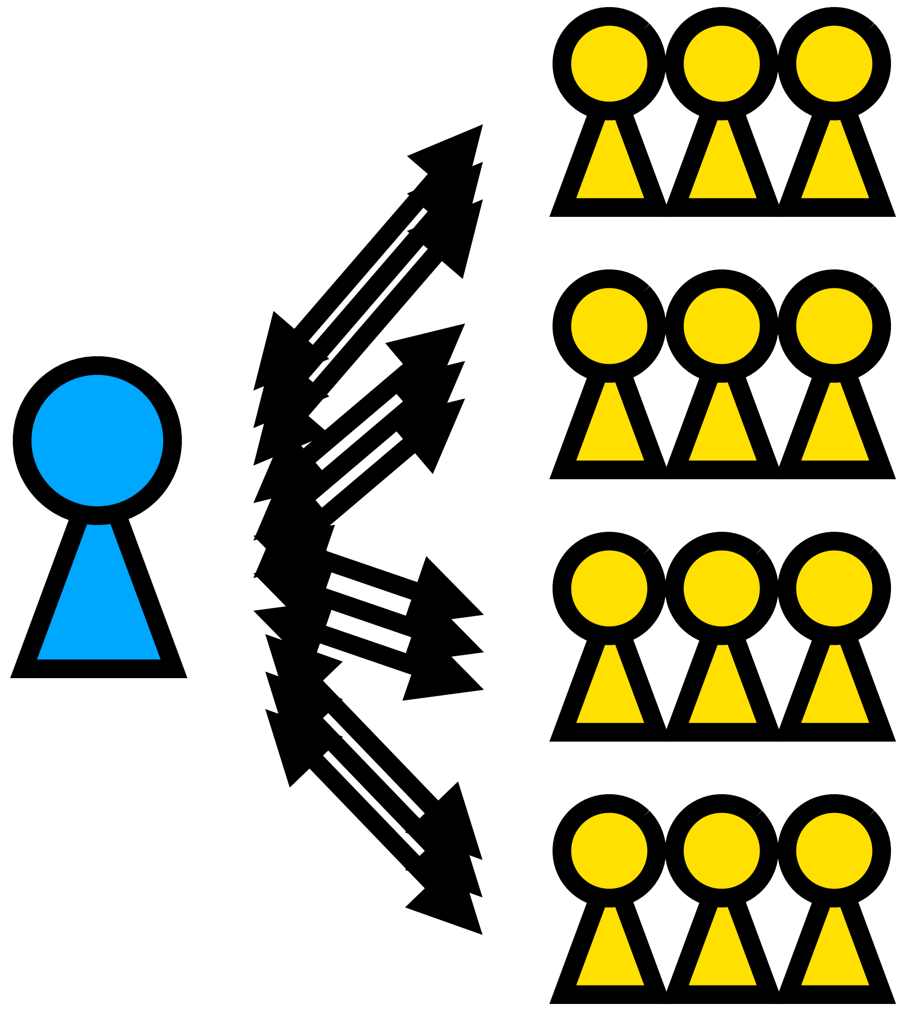
The speed and arbitrary message length of symmetric key

Protocols with public key cryptography

Goal: determine from whom a message came

Symmetric key

File downloads



One-to-many:
O(N) key
exchanges

Ideally, a user (blue) could post a message (e.g., sensitive documents or a kernel update), and then go offline

And downloaders (yellow) could subsequently infer the message's authenticity without having to have already established a pairwise key with the publisher

Digital signatures

A digital signature scheme comprises two algorithms

Signing function Sgn(SK, m)
- Inputs
  - **Secret** key SK
  - Fixed-length message
- Outputs: a *signature s*

This is a *randomized* algorithm
(nondeterministic output)

SK a.k.a. "Signing key"

Only one person can sign with a given (PK,SK) pair

Verification function Vfy(PK, m, s)
- Inputs
  - **Public** key PK
  - Message and signature
- Outputs: Yes/No if valid (m,s)

Deterministic algorithm

Anyone with the PK can verify

Digital signatures

A digital signature scheme comprises two algorithms

Signing **Sgn(SK, m)**	Verification **Vfy(PK, m, s)**
→ a *signature s*	→ Yes/No if valid (m,s)

Correctness

Vfy(PK, m, Sgn(SK, m)) = Yes

Security

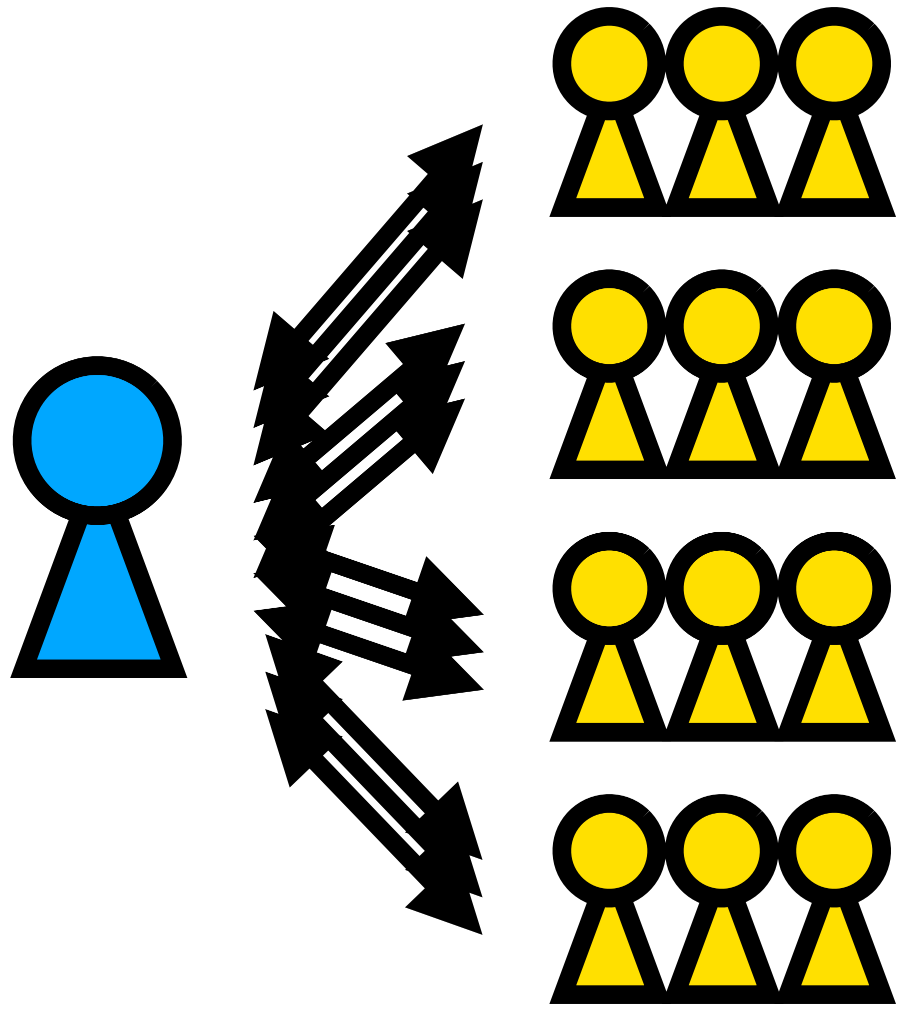
Same as with MACs: even after a chosen plaintext attack, the attacker cannot demonstrate an existential forgery

Protocols with digital signatures

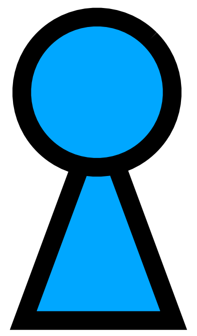
Goal: determine from whom a message came

Symmetric key

File downloads



One-to-many:
O(N) key
exchanges



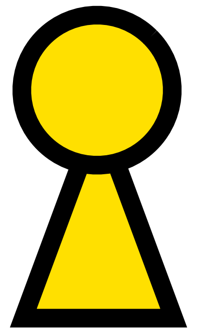
Generate public/private
key pair (PK,SK)

Annouce PK publicly
(on website, in newspaper, …)

Compute sig = Sgn(SK, msg)

Publish msg || sig

can now go offline!



Obtain PK, msg || sig

Vfy(PK, msg, sig)

Digital signature properties

Authenticity

Bob can prove that a message signed by Alice is truly from Alice (even without a *pairwise* key)

Integrity

Bob can prove that no one has tampered with a signed message

Non-repudiation

Once Alice signs a message, she cannot subsequently claim she did *not* sign that message

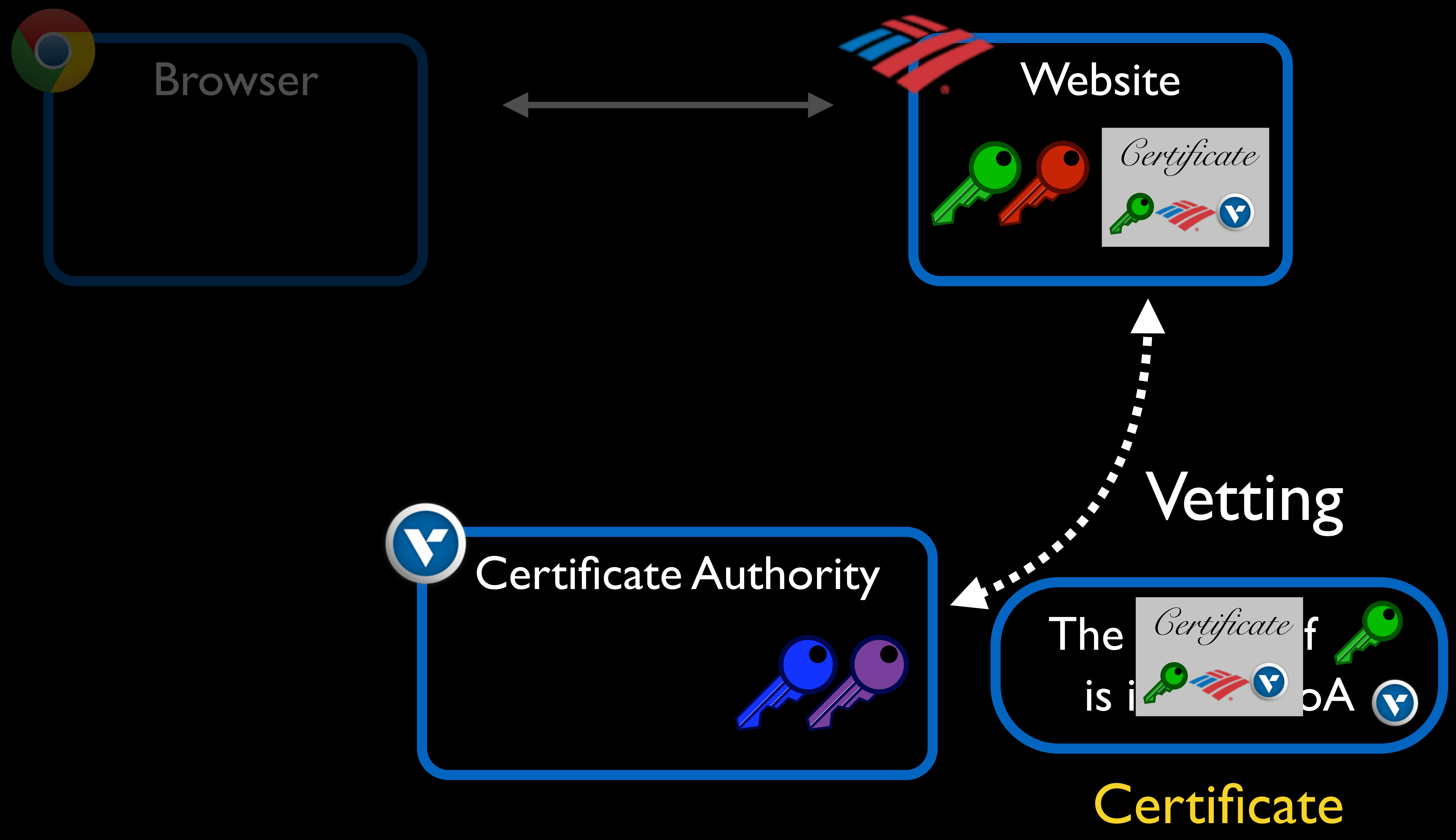
Do *handwritten* signatures at the end of a letter have these properties?

Authenticity	Bob can prove that a message signed by Alice is truly from Alice (even without a *pairwise* key) **Would require unforgeable handwritten signatures. This is the one property they sort of get**
Integrity	Bob can prove that no one has tampered with a signed message **Would require having a signature that depended on each part in the body of the letter**
Non-repudiation	Once Alice signs a message, she cannot subsequently claim she did not sign that message **Would require both of the above (unforgeable signature that depends on each part of letter)**

PUTTING IT ALL TOGETHER:
PUBLIC KEY INFRASTRUCTURE

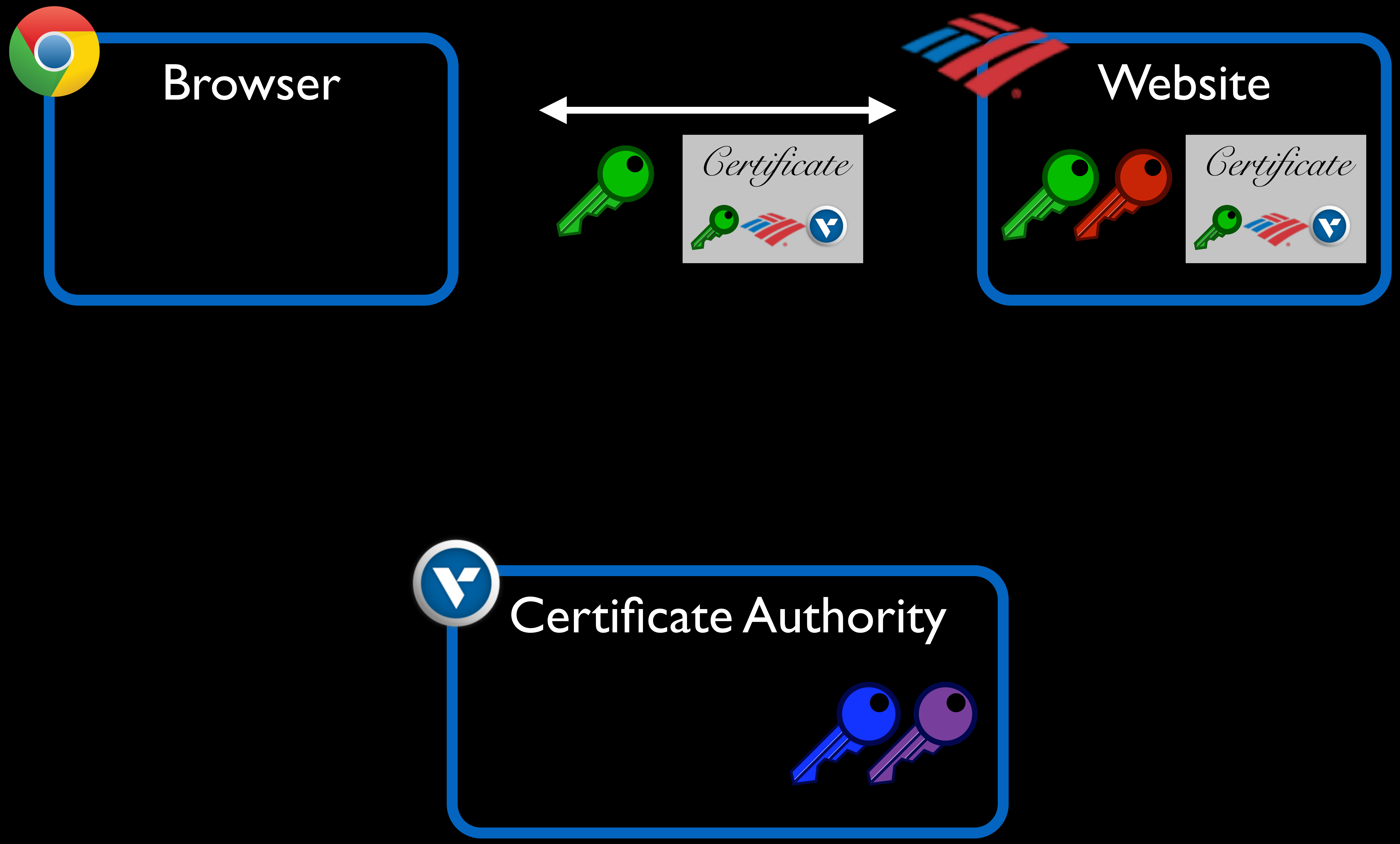
Public Key Infrastructures (PKIs)

How can users truly know with whom they are communicating?

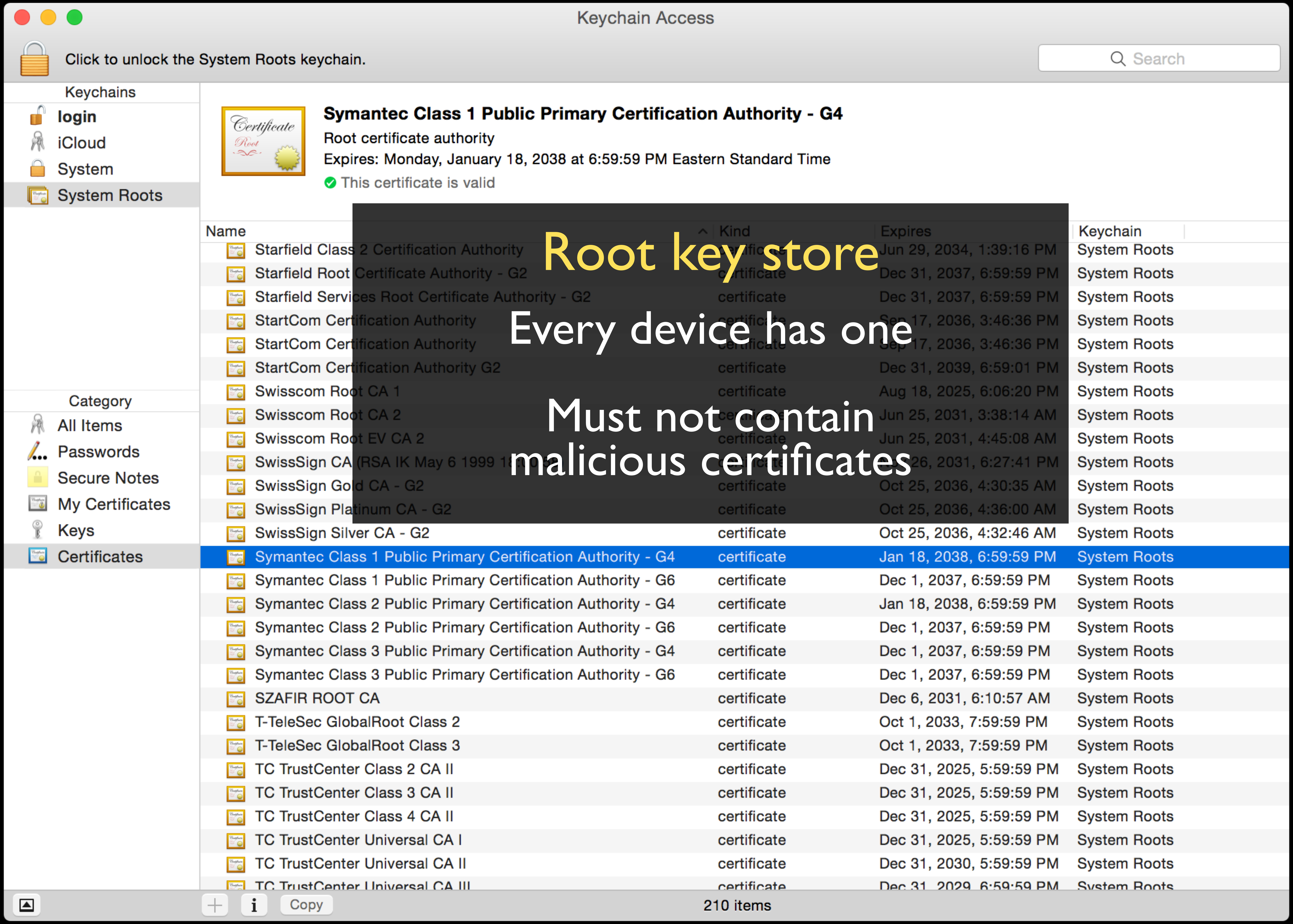


Public Key Infrastructures (PKIs)

How can users truly know with whom they are communicating?



Verifying certificates



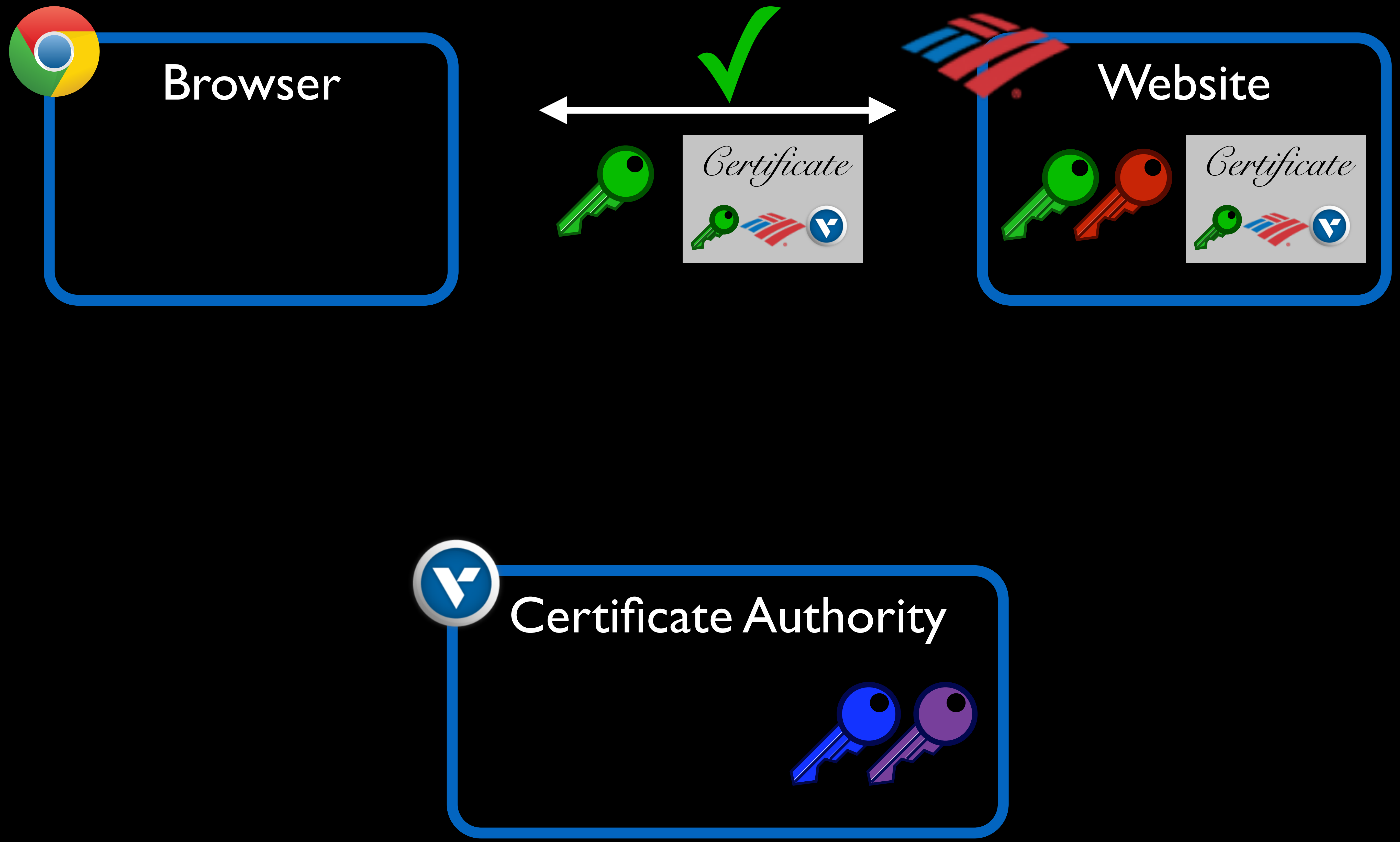
Root key store

Every device has one

Must not contain malicious certificates

Public Key Infrastructures (PKIs)

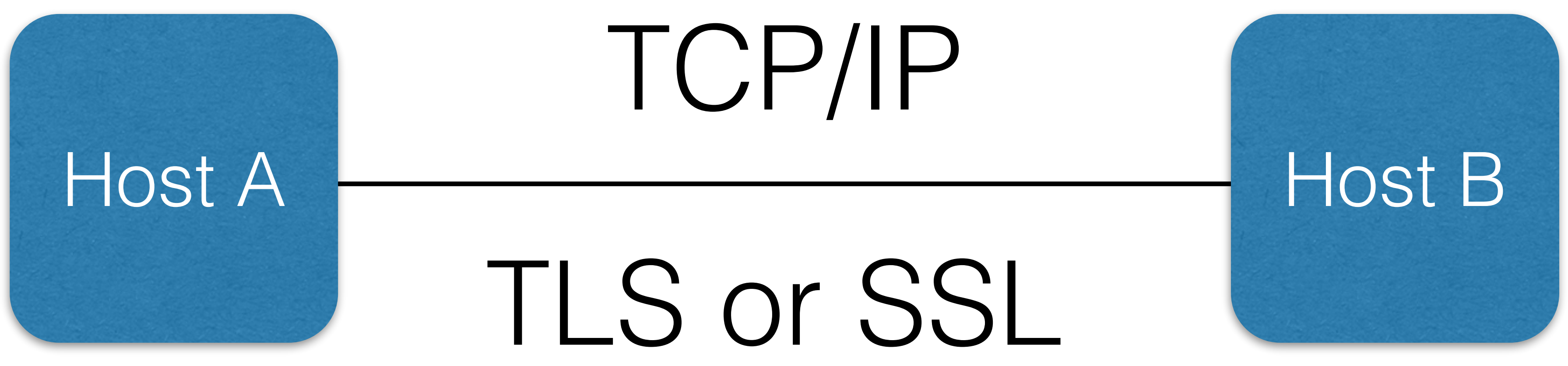
How can users truly know with whom they are communicating?



PUTTING IT ALL TOGETHER:
TLS

TLS/SSL

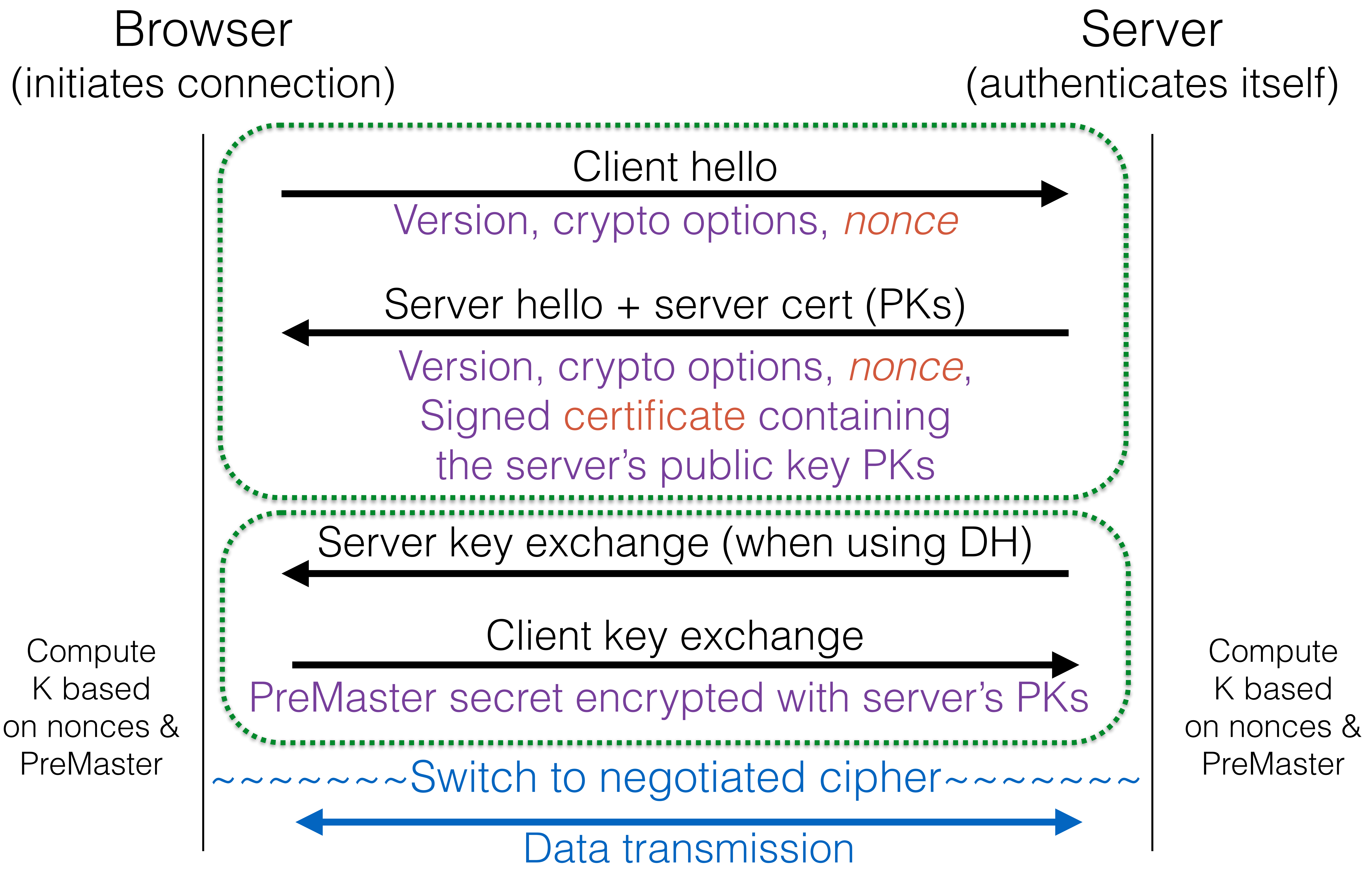
- TLS (Transport Layer Security)
  - A suite of protocols to provide secure communication
    - **Confidentiality** by applying block & stream ciphers
    - **Integrity** with MACs
    - **Authenticity** with certificates
  - Predecessor: SSL (secure sockets layer)
    - TLS was proposed as an upgrade
    - All versions of SSL are considered insecure (recently, the POODLE —padding oracle—attack)



TCP/IP: Host A and B can send packets to one another

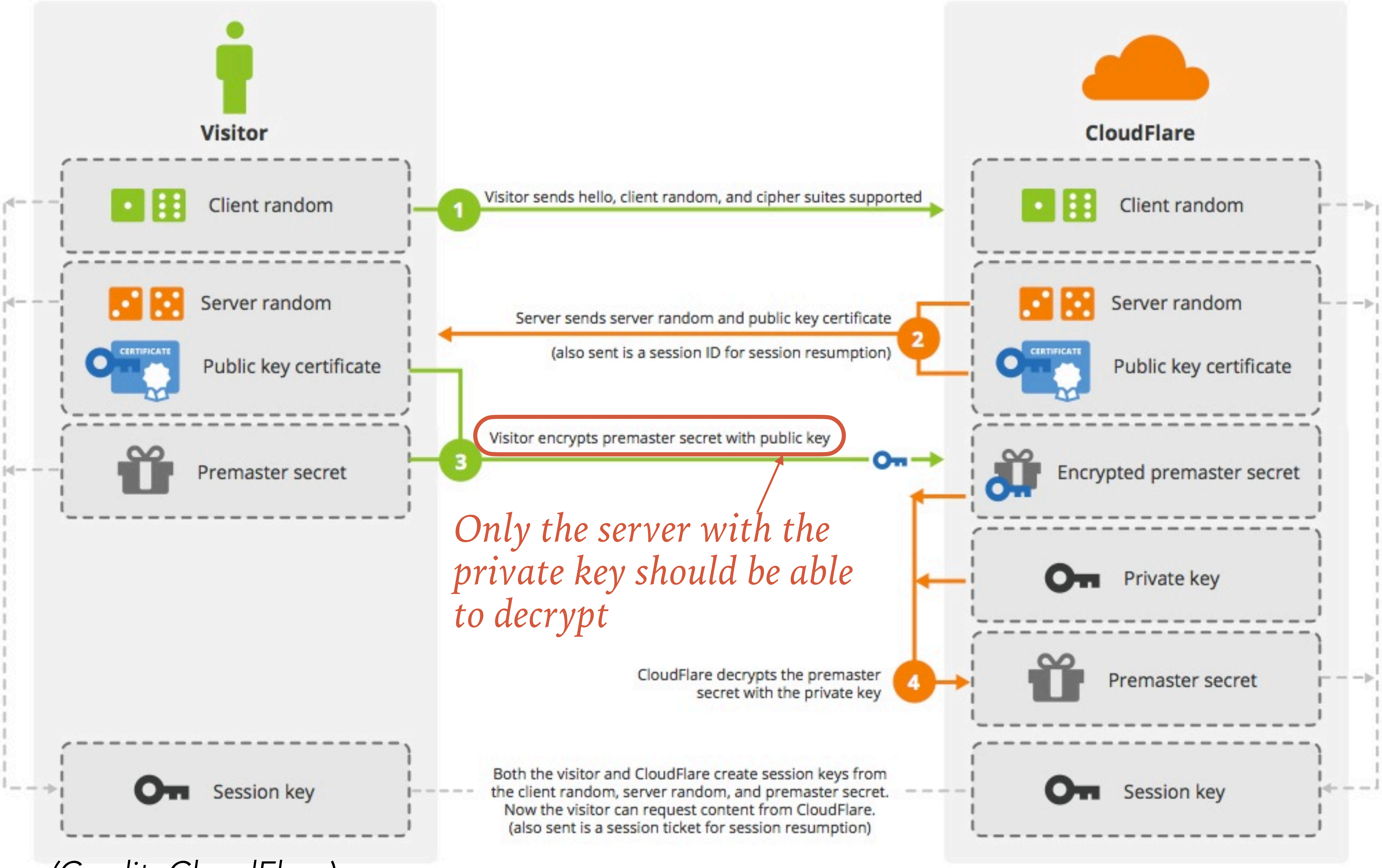
TLS/SSL: operate “over” TCP/IP to ensure security/authenticity

TLS/SSL protocol (high level)



SSL Handshake (RSA)

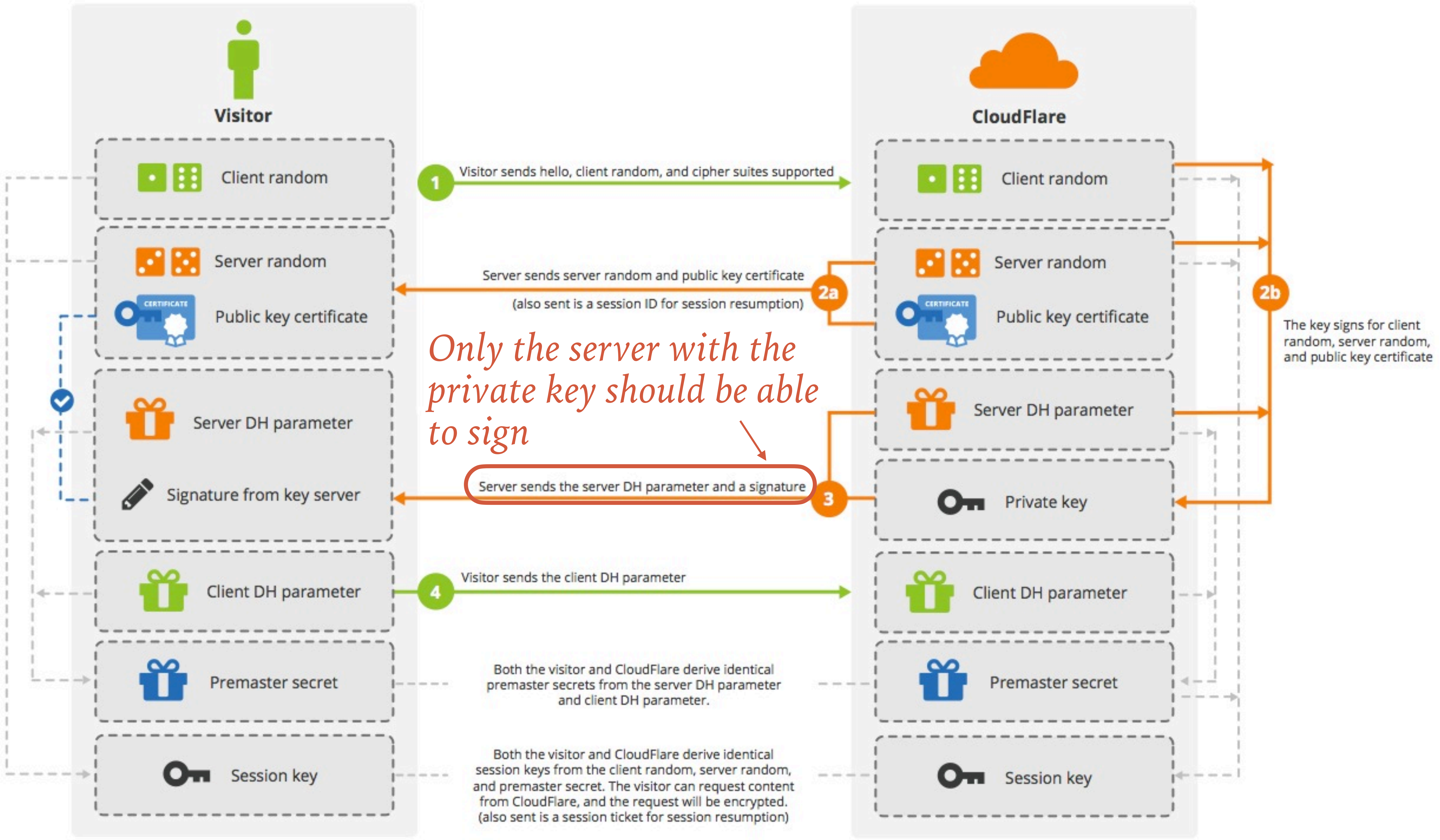
Handshake



(Credit: CloudFlare)

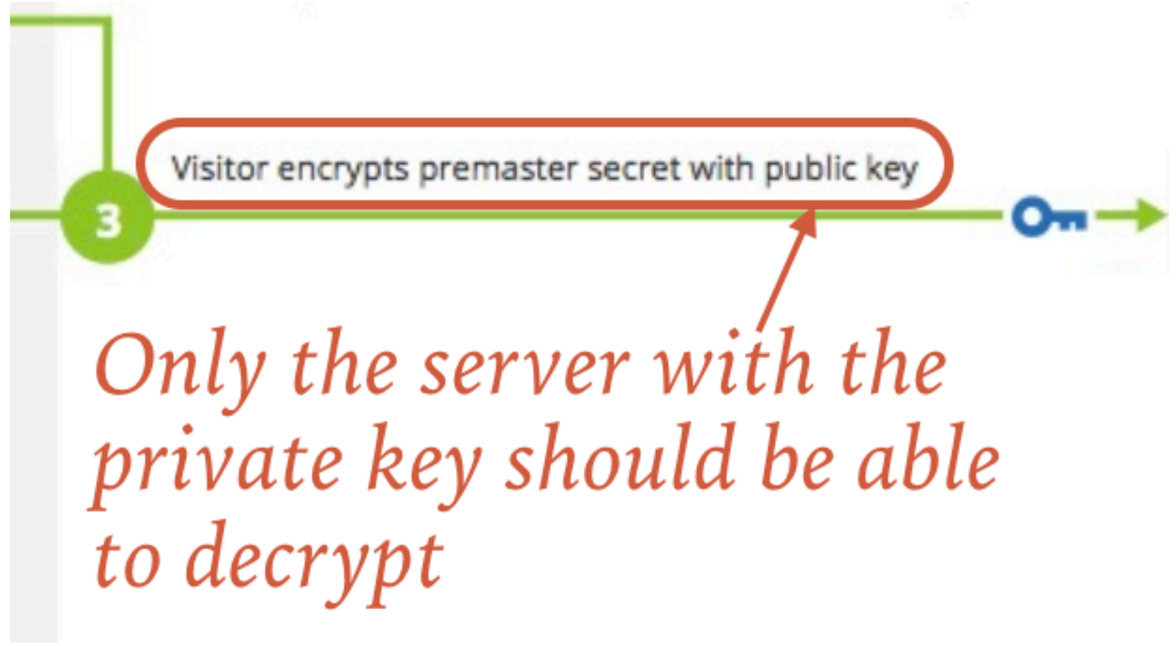
SSL Handshake (Diffie-Hellman)

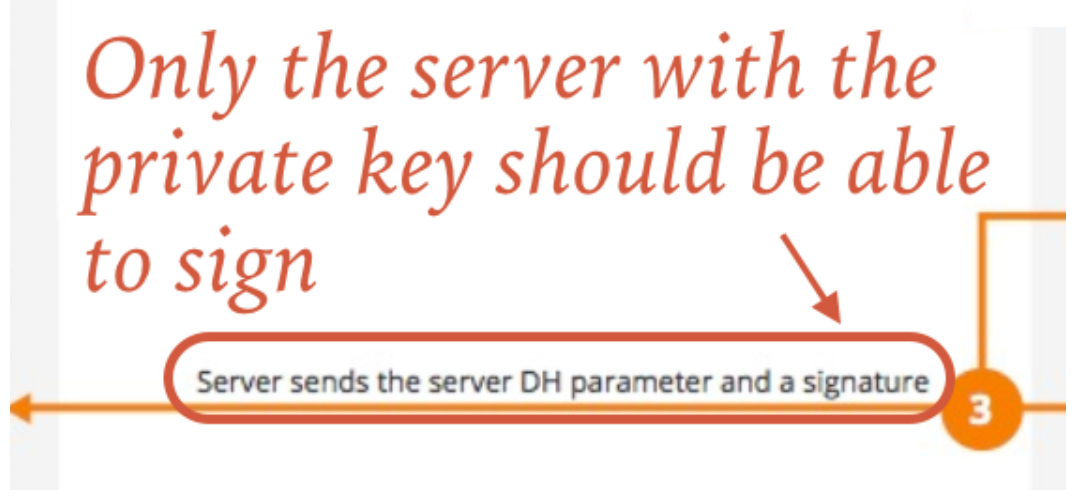
Handshake



(Credit: CloudFlare)

AUTHENTICATED DIFFIE-HELLMAN





Both of these serve as a "challenge/response" protocol:

The client is "challenging" the server to prove that it knows the secret key corresponding to the public key in the certificate

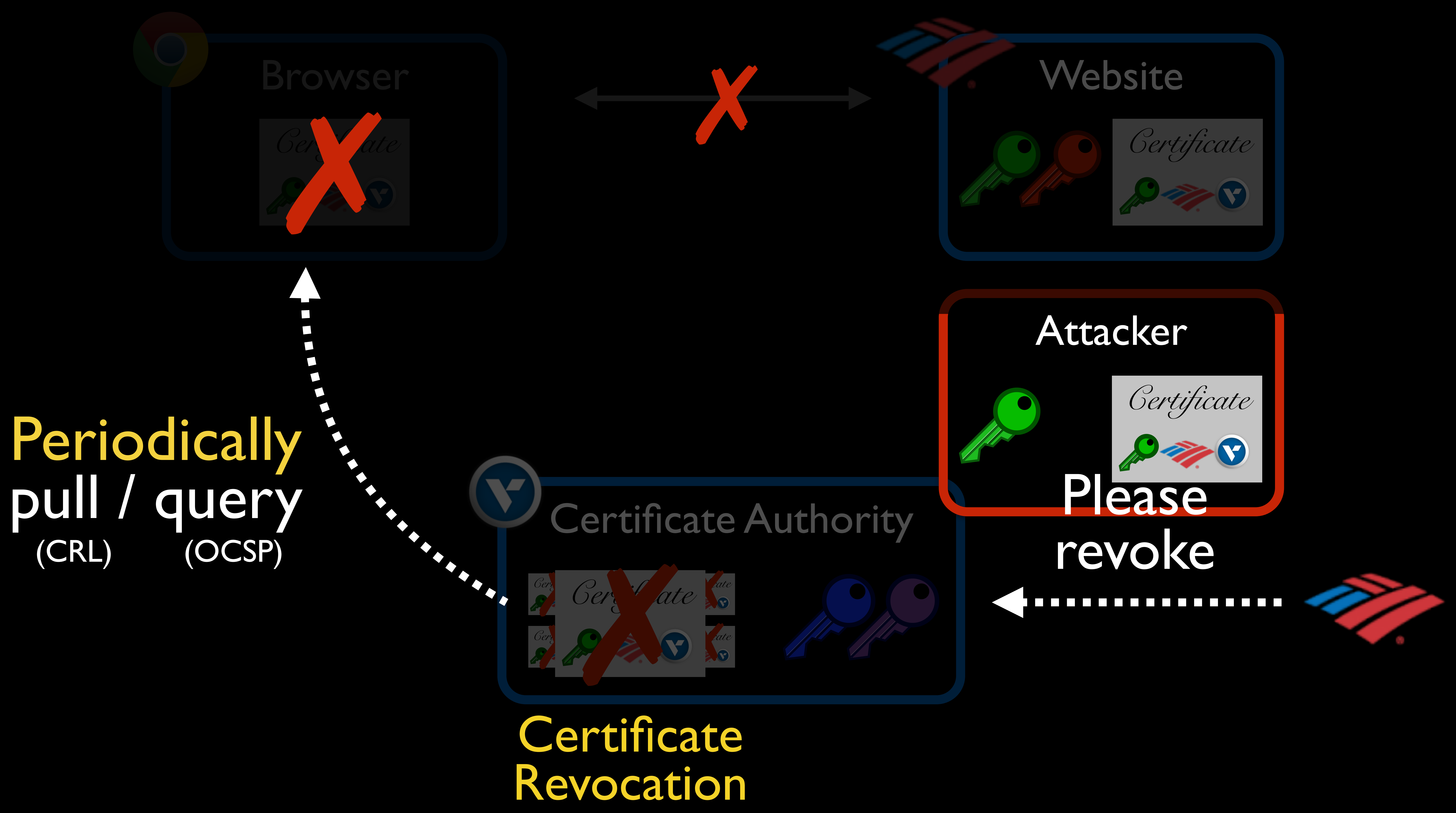
The server is providing a "zero-knowledge proof":

The server proves that it knows the secret key
without having to reveal the secret key itself

The key property that makes this work:
The only person who knows the secret key is the entity in the certificate

Certificate revocation

What happens when a certificate is no longer valid?



Certificate revocation

is a critical part of any PKI

Administrators must revoke and reissue
as quickly as possible

Browsers/OSes should obtain revocations
as quickly as possible

HASH FUNCTION APPLICATION
STORING PASSWORDS

THREAT MODEL

- Attacker can eventually gain access to the *hard drive* where (some version of) the passwords are stored long-term
- But attacker does *not* gain access to memory (where raw passwords might be stored while processing)
- Attacker gets as much prep time as they want, but not unlimited amounts of storage
- Goal of the attacker: recover passwords within some window of time

FAILED IDEA #1: STORE THE PASSWORDS

username : password

- Attacker can eventually gain access to the *hard drive* where (some version of) the passwords are stored long-term

The attacker trivially gains access to the passwords

FAILED IDEA #2: STORE ENCRYPTED PASSWORDS

$$username : E(K, IV, password), IV$$

- Attacker can eventually gain access to the *hard drive* where (some version of) the passwords are stored long-term

This can work if the key is not stored on the hard drive

But if the key is stored on the hard drive, then it is trivial for the attacker to recover

FAILED IDEA #3: STORE HASHED PASSWORDS

username : H(password)

- Attacker can eventually gain access to the *hard drive* where (some version of) the passwords are stored long-term

Problem 1: many users use the same password

Most common *H(password)* = most common password

1. 123456
2. password
3. 12345678
4. qwerty
5. 123456789
6. 12345
7. 1234
8. 111111
9. 1234567
10. dragon

Problem 2: attacker gets prep time

They can precompute hashes

(H(123456), H(password), ...)

More compact representation of this is a rainbow table

RAINBOW TABLES

username : H(password)

- Goal: compact storage of hashes of many passwords

$$\texttt{aaaaaa} \xrightarrow[H]{} \texttt{281DAF40} \xrightarrow[R]{} \texttt{sgfnyd} \xrightarrow[H]{} \texttt{920ECF10} \xrightarrow[R]{} \texttt{kiebgt}$$

Hash *"Reduction"*

$kiegbt = R(H(R(H(aaaaaa))))$

A "reduction" function is simply a function that takes a hash's output as its input and outputs a potential input (in this case, a 6-letter password)

Only store the beginning of this chain (aaaaaa) and the end (kiebgt)

RAINBOW TABLES

username : H(password)

$$\texttt{aaaaaa} \xrightarrow[H]{} \texttt{281DAF40} \xrightarrow[R]{} \texttt{sgfnyd} \xrightarrow[H]{} \texttt{920ECF10} \xrightarrow[R]{} \texttt{kiebgt}$$

Only store the beginning of this chain (aaaaaa) and the end (kiebgt)

Do this for many initial seed inputs (bbbbbb, password, 123456, etc.)

Given H(password)

$x = R(H(password))$

Is x one of the end values (e.g., kiebgt)?

If so, then the password must have been one of the passwords in the chain

If not, then $y = H(x); x = R(y)$ *and try again*

Give up after some maximum number of tries

FAILED IDEA #4: STORE SALTED HASHED PASSWORDS

$$username : H(salt \mid password), salt$$

- Remember: *small changes to the input leads to large, unpredictable changes in the output*

Good news: Rainbow tables don't work anymore

Bad news: Can still try a dictionary attack against a given user because **hash functions are *very efficient to compute***

*Ideally we would have a very **slow** hash function*

How can we create a slow hash function out of a fast hash function?

HOW PASSWORDS ARE STORED

username : H^k(salt | password), salt

- $H^k = H(H(H(\ldots H(x)\ldots)))$
- Compute the hash of the hash of the hash of the…

H is a fast hash function; H^k is a slow one!

This is how passwords are stored in Linux today

Recall: Given H(password), it is infeasible to recover password,

So what does it mean if a website can email you your password?