50.042 FCS Summer 2024
Lecture 13 – Digital Signatures

Felix LOH
Singapore University of Technology and Design

With selected materials adapted from: *Understanding Cryptography: A Textbook for Students and Practitioners, by C. Paar and J. Pelzl*

A quick recap on message integrity, authenticity and non-repudiation

- ***Integrity***: Eve/Oscar is unable to modify the message transmitted between Alice and Bob without detection

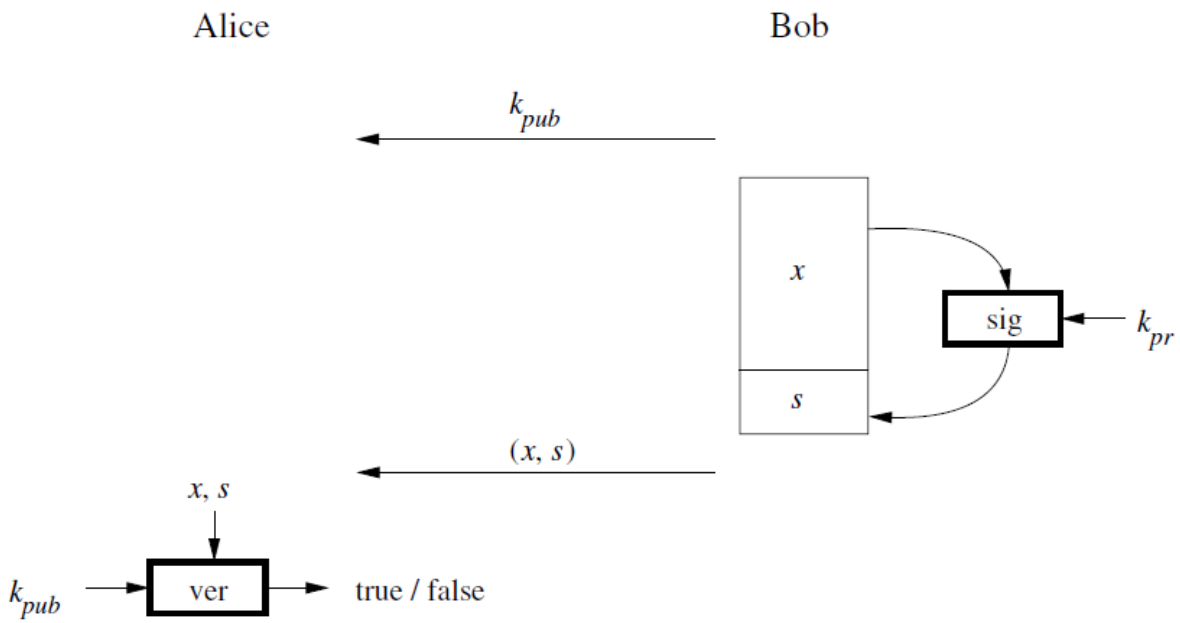
- ***Authenticity***: The property that the message originated from its purported (and legitimate) source, e.g. when Alice receives a message from Bob, if the message is *authentic*, she can be confident that the message was indeed sent by Bob, and not from Eve/Oscar

- ***Non-repudiation***: The assurance that the sender of the message, Bob, is provided with proof of delivery to Alice, and Alice is provided with proof of Bob's identity, so neither legitimate party can later deny having sent/received the message

Digital signatures: from the last lecture…

- MACs can provide *message integrity* and *authentication*, but **cannot** provide *non-repudiation*
- However, we can use an asymmetric cryptosystem, like RSA, to provide *non-repudiation* as a service: that's basically a digital signature
- The general idea:
  - Bob can use his private key to "decrypt" a plaintext message *x* – this is essentially a digital signature. He then sends *x* and the digital signature over to Alice
  - Alice can then verify that Bob was the sender of *x*, by "encrypting" the digital signature using Bob's public key
  - Since only Bob is in possession of his private key, he cannot later deny that he was the sender of the message *x*

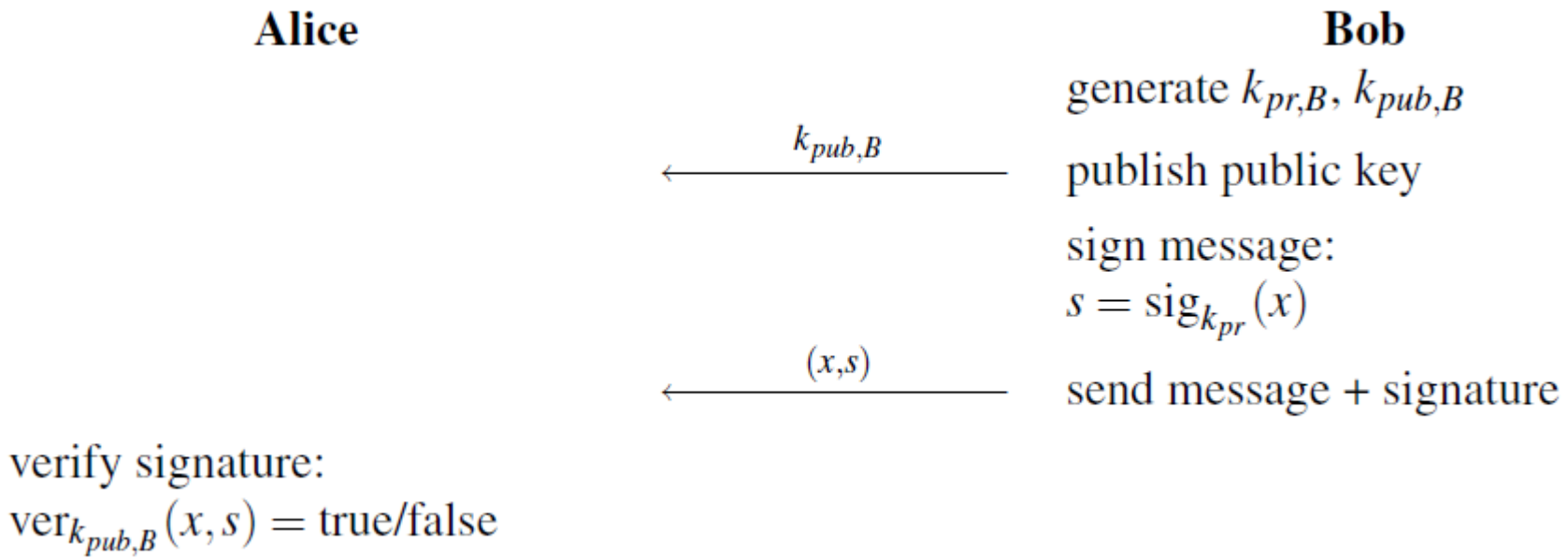
Digital signatures: basic concept



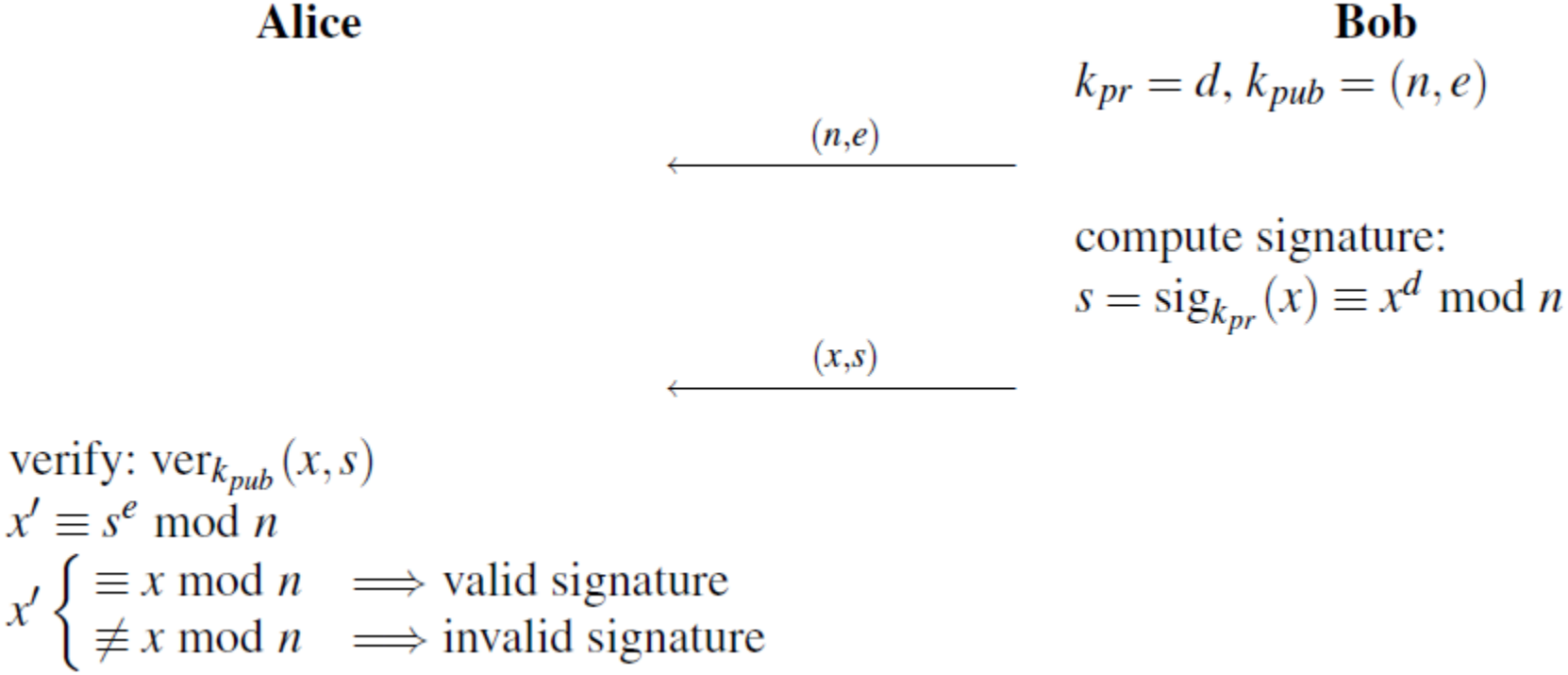
Digital signatures: basic concept

- Digital signatures can provide *integrity*, *authentication*, and *non-repudiation*
- Bob, the sender of a plaintext message *x*, creates a digital signature *s* of the plaintext message using his private key k_{pr} and sends both *x* and *s* over to Alice
  - Assuming that Bob has kept k_{pr} a secret known only to himself, only Bob can sign *x*
  - The signature generation function uses k_{pr} and *x* as inputs, and the output is *s*
- Alice, upon receiving *x* and *s*, uses a matching public key k_{pub} to verify Bob's signature *s*
  - The signature verification function uses k_{pub}, *x* and *s* as inputs, and the output is a Boolean value (True/False)
  - If the output is True, then Alice can be assured that the message *x* was indeed sent by Bob and that *x* wasn't modified in transit.
  - In addition, Bob cannot later deny that he sent the message *x*

Digital signature protocol: basic idea



The basic RSA digital signature protocol



- This basic digital signature protocol requires only minor modifications to the original RSA cryptosystem

The basic RSA digital signature protocol

- After the RSA parameters have been chosen, Bob creates his private key k_{pr} and public key k_{pub}
  - Note: these are the same keys that would be used for regular RSA encryption and decryption
- He then sends k_{pub} over to Alice

- Suppose Bob wants to send Alice a plaintext message x that will be signed by him. He uses his private key k_{pr} to sign the plaintext x, i.e. by **decrypting** x with k_{pr}, to obtain the signature $s \equiv x^d \bmod n$

The basic RSA digital signature protocol

- Bob now sends *x* and *s* over to Alice
- Upon receiving *x* and *s*, Alice uses Bob's public key k_{pub} to verify *x*. She does this by **encrypting** *s* with k_{pub} to obtain *x'*, such that $x' \equiv s^e \bmod n$

- Alice can finally check whether the signature *s* is valid, which indicates that the plaintext message *x* originated from Bob and has not been tampered with
  - To check the validity of the signature, Alice compares *x'* with *x*
  - If $x' \equiv x \bmod n$, the signature is valid
  - Otherwise, the signature is invalid – this means that either *x* or *s* (or both) were modified during transit

Basic RSA digital signature protocol: example

Alice

Bob

1. choose $p = 3$ and $q = 11$
2. $n = p \cdot q = 33$
3. $\Phi(n) = (3-1)(11-1) = 20$
4. choose $e = 3$
5. $d \equiv e^{-1} \equiv 7 \bmod 20$

$(n,e)=(33,3)$ ←

compute signature for message $x = 4$:
$s = x^d \equiv 4^7 \equiv 16 \bmod 33$

$(x,s)=(4,16)$ ←

verify:
$x' = s^e \equiv 16^3 \equiv 4 \bmod 33$
$x' \equiv x \bmod 33 \Longrightarrow$ valid signature

Proof of correctness of the basic RSA digital signature protocol

- If the signature *s* was not altered during transmission, we have:

  $x' \equiv s^e \equiv (x^d)^e \equiv x^{de} \equiv x \bmod n$, from our proof of correctness of the RSA encryption and decryption from Lecture 12

- Then, if the plaintext message *x* was not altered during transmission, we have:

  $x' \equiv x \bmod n = x$

- Thus, by verifying that *x'* is equivalent to *x*, one can check that the plaintext message *x* is authentic and that it has not been modified

The basic RSA digital signature protocol: comments

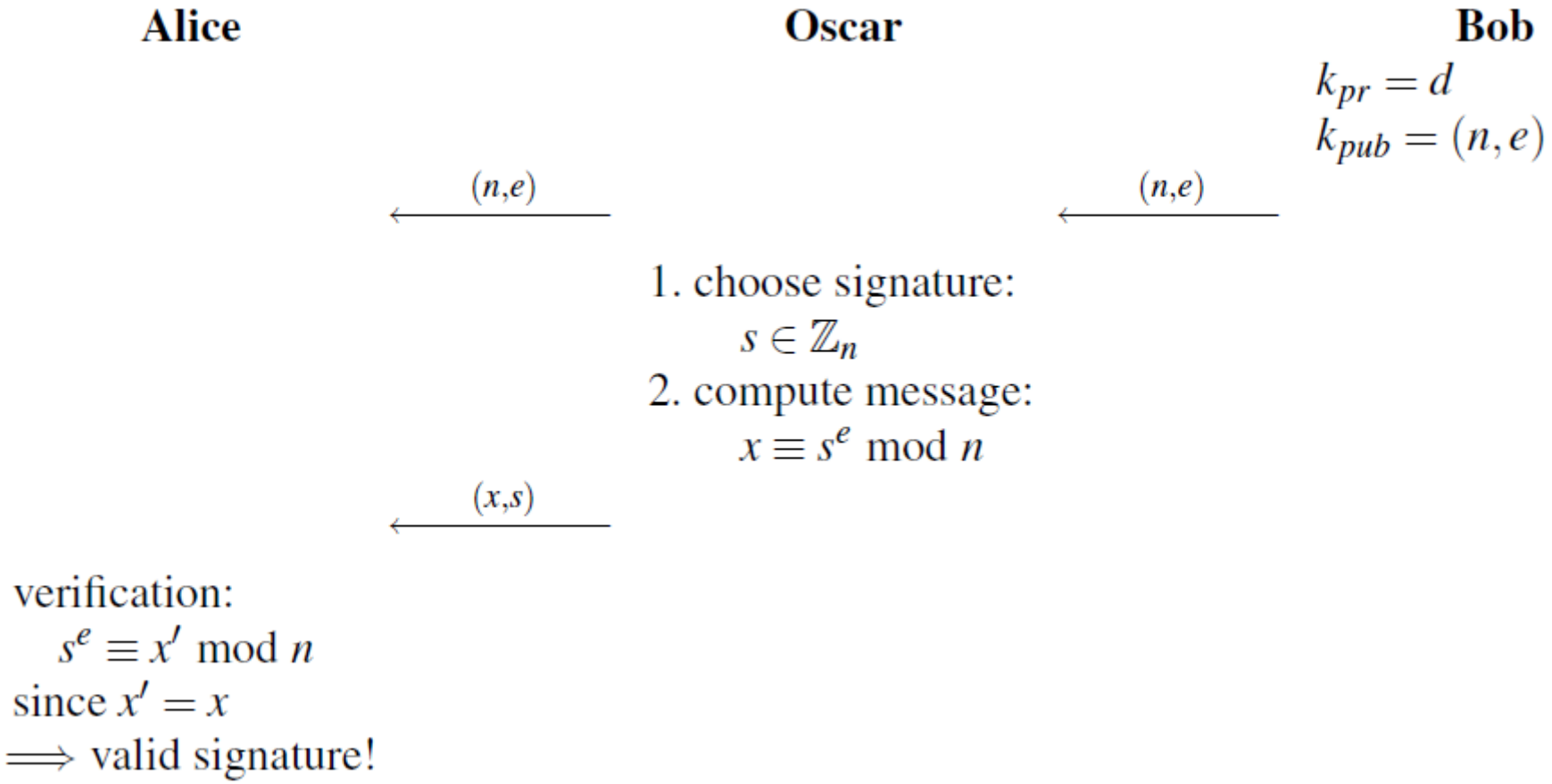
- The modulus n is typically in the range of 1024 to 3072 bits
- Like in RSA encryption/decryption, the square and multiply algorithm should be used to reduce the overhead of modular exponentiation during the computation and verification of the signature
- *Short public keys* (e.g. choosing $e = 2^{16} + 1$) can also help in reducing the overhead, as signature verification becomes a very fast operation
  - In most practical scenarios, the plaintext message is signed only once but verified many times
- This basic RSA signature protocol is vulnerable to certain kinds of attacks, like the existential forgery attack

Possible attacks against RSA signatures

- Mathematical attacks, such as factoring of the modulus *n*
  - This kind of attack can be defeated by making *n* large enough (1024 bits or more)

- Existential forgery attack
  - The attacker Oscar can generate a **valid** signature *s* for any **random** message *x*

Existential forgery attack



Existential forgery attack

- Bob attempts to send Alice his public key k_{pub}
- However, Oscar is listening in the channel. He takes note of the modulus n and parameter e contained in k_{pub}, but does not interfere with Bob's transmission
- Oscar first chooses some signature s, then he computes the corresponding message x, using the equation $x \equiv s^e \bmod n$
- Oscar then sends the bogus message x and its signature s over to Alice
- Alice upon receipt of the message and signature, will verify that the signature is valid, since the signature satisfies the mathematical requirements set by the basic RSA signature protocol

Existential forgery attack: comments

- Because Oscar has to first choose some signature *s* **before** computing the corresponding message *x*, Oscar **cannot** control the contents of the message *x*
  - The message *x* will most likely look random/gibberish to Alice
  - Still, this is an undesirable feature of the basic RSA signature protocol

- This attack can be prevented by allowing only certain valid message formats for *x*
  - This can be achieved though padding of *x* — e.g. require a simple formatting rule for *x* to have 64 trailing bits with the value ‘1’

Existential forgery attack: comments

- This attack can also be defeated by *hashing* the message *x* before computing its signature, i.e. sign a hash of the message instead of the message itself

- A widely-used padding scheme is the *Probabilistic Signature Standard*, RSA-PSS
  - It combines signature and verification with an encoding of the message
  - Generally speaking, the message is padded with a fixed pattern as well as a random salt, then a hash (i.e. message digest) of the resulting string is computed
  - After some additional computation (including masking) on the hash, the final result is then signed in the same manner as the basic RSA signature protocol

Other digital signature protocols

- There are also protocols that are based on the discrete logarithm problem in $\mathbb{Z}^*_p$
  - E.g. the Digital Signature Algorithm (DSA), which is a federal US government standard for digital signatures

- Protocols based on elliptic curves also exist
  - E.g. the Elliptic Curve Digital Signature Algorithm (ECDSA)

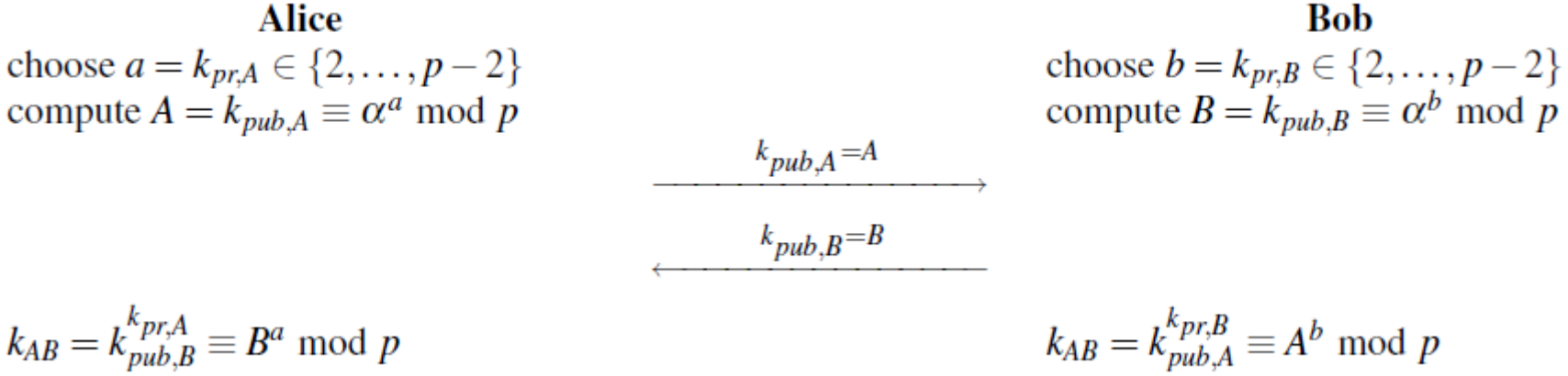
Certificates: motivation

- Back in Lecture 11, we discussed key distribution techniques using asymmetric (public key) algorithms
  - Particularly, the Diffie-Hellman key exchange (DHKE) protocol
  - Asymmetric cryptosystems allow us to safely transmit information *over an insecure channel* that would allow legitimate parties like Alice and Bob to securely derive a shared secret key
  - These provide good protection against eavesdropping (i.e. *passive* attacks)
- Unfortunately, key establishment techniques that use asymmetric cryptosystems, like DHKE, still have some shortcomings for key distribution
  - They are all vulnerable to a *man-in-the-middle* attack (i.e. an *active* attack)
  - As such, they require what we call an *authenticated channel* to distribute the public keys — this can be achieved through the use of *certificates*

DHKE (recap)

a. Diffie-Hellman setup phase (picking of domain parameters):
   1. Choose a large prime p
   2. Choose a generator $\alpha \in \{2, 3, \ldots, p\text{-}2\}$
   3. Publish the domain parameters p and α (for Alice and Bob to use in the next phase – key exchange)

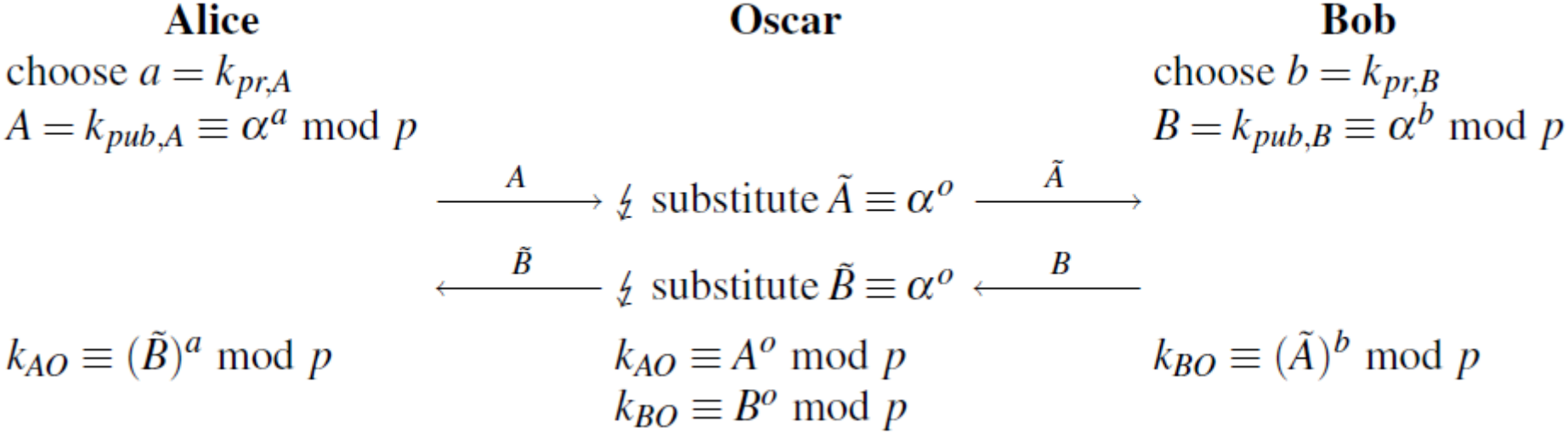
DHKE (recap)



b. Diffie-Hellman key exchange phase (generate a joint secret key k_{AB}):

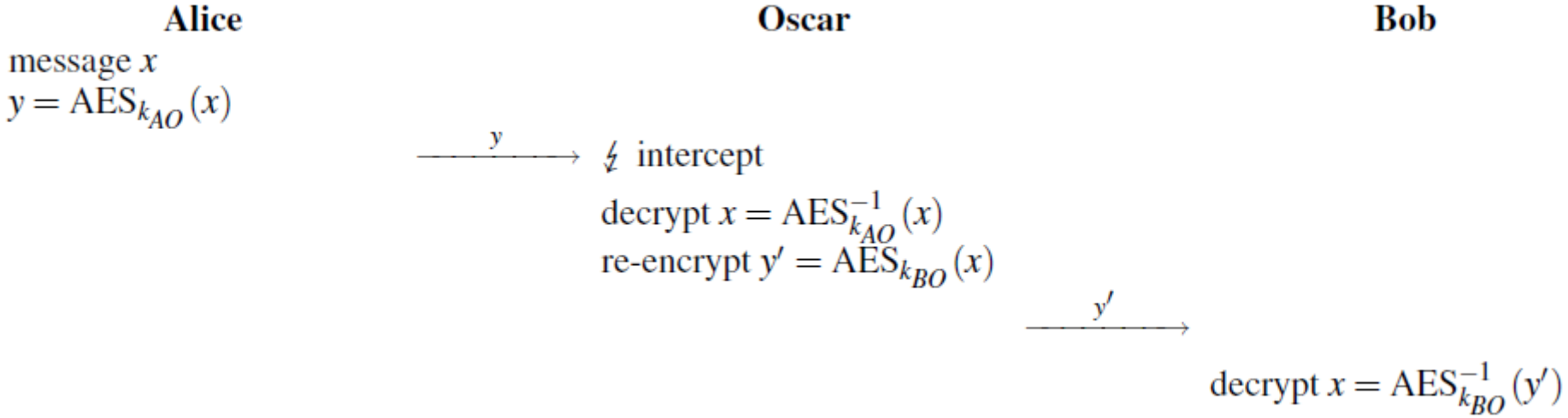
1. Alice picks a private key, $k_{pr,A} = a \in \{2, 3, \ldots, p\text{-}2\}$ and computes her public key $k_{pub,A} = A \equiv \alpha^a \bmod p$, then sends the public key over to Bob
2. Likewise, Bob picks a private key, $k_{pr,B} = b \in \{2, 3, \ldots, p\text{-}2\}$ and computes his public key $k_{pub,B} = B \equiv \alpha^b \bmod p$, then sends the public key over to Alice
3. Upon receiving each other's public key, Alice computes the joint secret key $k_{AB} = (k_{pub,B})^a \equiv \alpha^{ba} \bmod p$ and Bob computes $k_{AB} = (k_{pub,A})^b \equiv \alpha^{ab} \bmod p$

Man-in-the-middle attack against DHKE



- The underlying idea is that Oscar intercepts and replaces Alice's and Bob's respective public keys with his own public key
  - Same replacement public key for both Alice and Bob
- As a result, Alice computes a joint secret key k_{AO} with Oscar and Bob likewise computes a joint secret key k_{BO} with Oscar
  - Alice and Bob are unaware that they share a respective (and different) secret key with Oscar and not a joint secret key with each other!

Message manipulation after a successful man-in-the-middle attack



- Oscar can now act as a relay between Alice and Bob
- When Oscar receives a ciphertext *y* from Alice that is meant for Bob, Oscar can simply decrypt the ciphertext, then re-encrypt the plaintext *x* using his joint secret key k_{BO} with Bob, and relay the new ciphertext *y'* to Bob
- In addition to reading the plaintext *x*, Oscar can modify *x* before relaying it

The need for an authenticated channel and certificates

- The underlying issue with the man-in-the-middle attack is that the public keys are not authenticated
  - Alice and Bob are unable to verify the source of the public keys
  - In the scenarios we discussed, when Bob receives a public key that is purportedly Alice's, he has no way of telling that this is in fact the case
- In practice, a public key from, say, Alice would have the following format:

  $k_A = (k_{pub,\,A}, ID_A)$,

  where k_A is the "container" or message that is sent to Bob, $k_{pub,\,A}$ is the binary string representing the actual public key of Alice and ID_A is some identifying information, e.g. Alice's name and date of birth or her IP address, etc.

The need for an authenticated channel and certificates

- So, if Oscar is conducting a man-in-the-middle attack, he will change Alice's public key to:

  $k_A = (\boldsymbol{k_{pub,\,O}}, ID_A)$

- The only thing that is changed is the anonymous binary string representing Alice's actual public key, so Bob won't be able to detect that the changed binary string is in fact Oscar's

- This has a very significant consequence:

  Even though asymmetric cryptosystems do not require a secure channel, they require an authenticated channel for distribution of the public keys.

The need for an authenticated channel and certificates

- This is where *certificates* come in: Since the authenticity of the message $k_A = (k_{pub,\,A}, ID_A)$ is violated by an active attack like a man-in-the-middle attack, we apply a digital signature that provides authentication
- So a certificate in its basic form for Alice's public key would look like:

  $Cert_A = [(k_{pub,\,A}, ID_A), sig_{k_{pr}}(k_{pub,\,A}, ID_A)]$
- Note that here, we use "$sig_{k_{pr}}()$" and **not** "$sig_{k_{pr,\,A}}()$" for the signature
- That's because Alice will not be signing the certificate – that will be done by a mutually trusted third party (the certification authority)
- **Certificates tie the identity of a user to his/her public key**

Certificates and certification authorities (CAs)

- Certificates require that the receiving party has the correct public key to verify the digital signature in the certificate
- As hinted in the previous slide, Bob **cannot** be expected to use Alice's public key to verify the digital signature
  - We would be back to the same problem as before, because Bob needs to be in possession of Alice's public key **prior** to verifying the signature
  - Alice's public key would have to be sent over an authenticated channel in the first place
- Thus, Alice's private key cannot be used to sign the certificate; instead, the private key of a mutually trusted third party will be used
  - This third party is a certification authority (CA)
  - In practice, the CA not only signs the certificate, it generates the certificate

Certification authorities (CAs)

- The job of a CA is to generate and issue certificates for all users in a system
- Some of the most globally trusted CAs include:
  - IdenTrust
  - Sectigo
  - GlobalSign
  - Let’s Encrypt

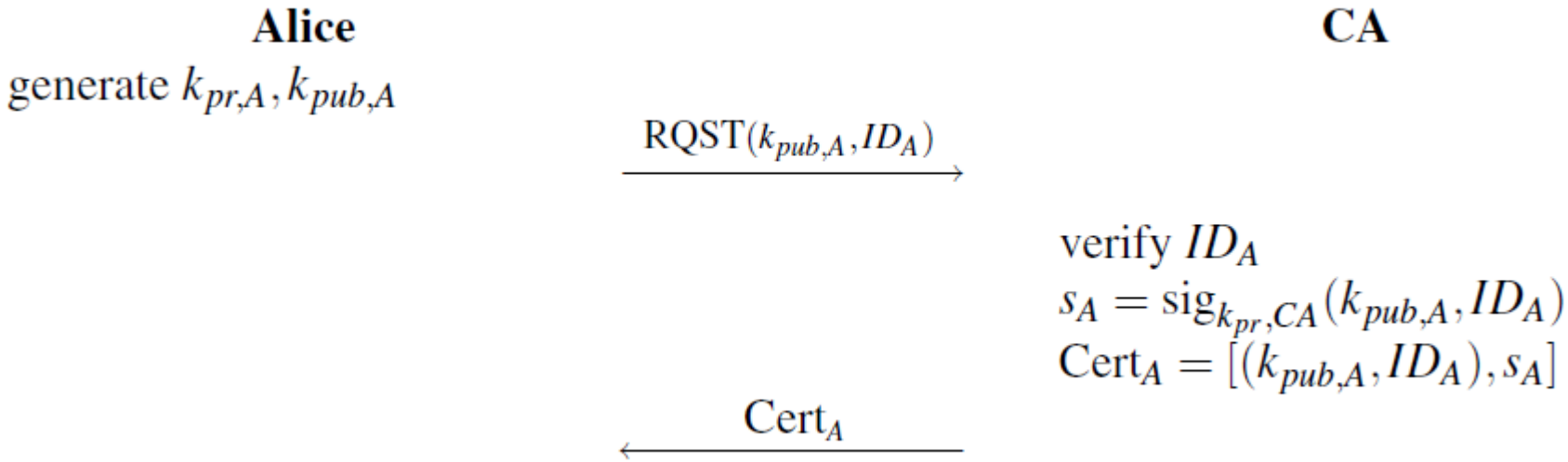
Certificate generation

- There are two main cases for certificate generation:

a) Certificate generation with user-provided keys

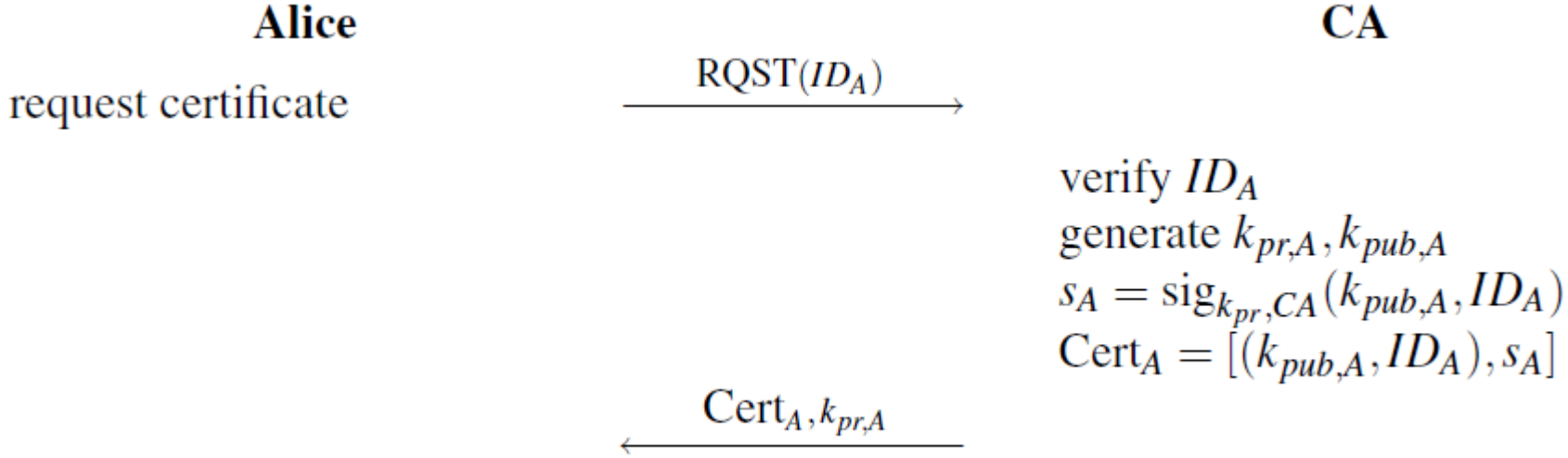
b) Certificate generation with CA-generated keys

Certificate generation with user-provided keys



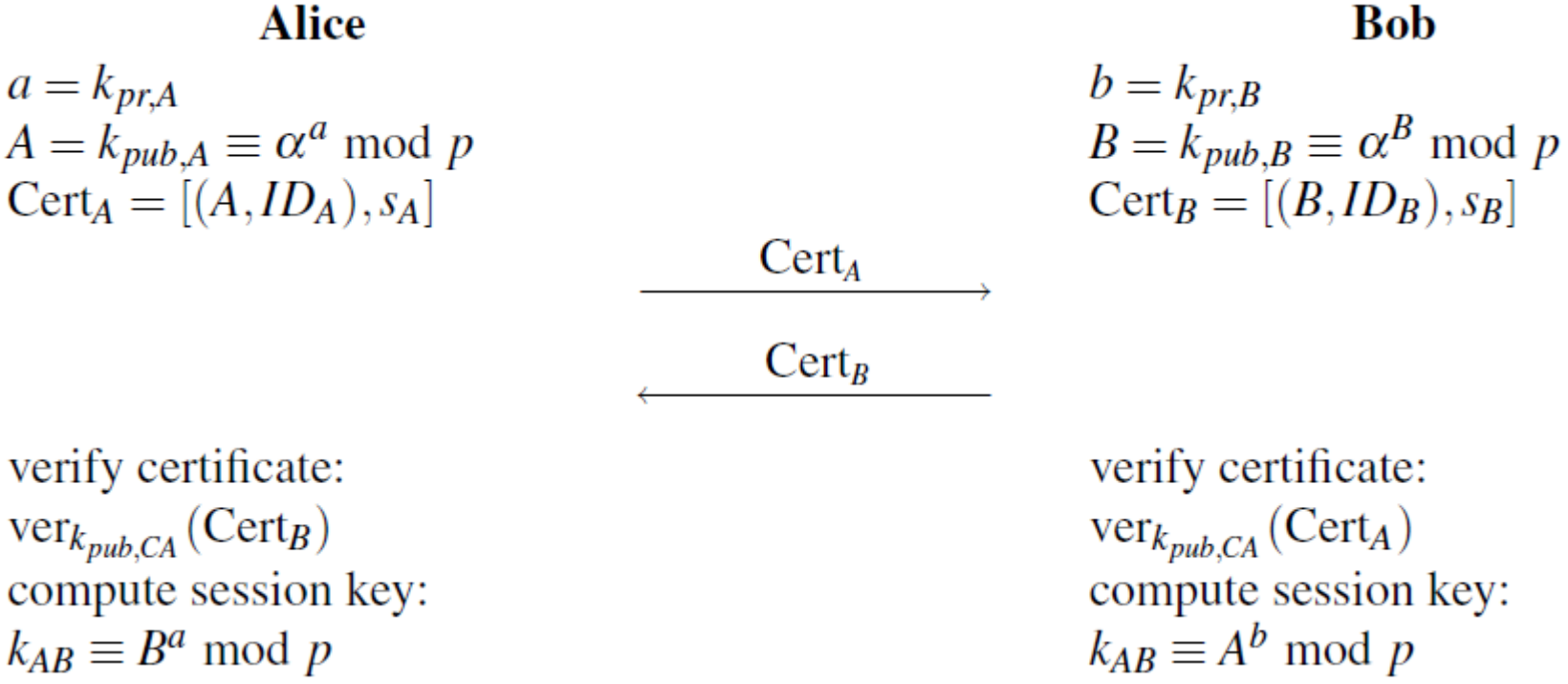
- In this method, Alice computes her own public-private key pair and simply requests the CA to sign her public key
  - Ideally, the first transmission (Alice's request to the CA) should be sent over an authenticated channel

Certificate generation with CA-generated keys



- In this method, Alice requests the CA to generate a public-private key pair on her behalf, and sign the public key
  - Ideally, the first transmission (Alice's request to the CA) should be sent over an authenticated channel
  - The second transmission (from the CA to Alice) **must** be sent over an authenticated **and** secure channel, since Alice's private key is being sent here

DHKE with certificates



- Now we can discuss DHKE that is augmented with certificates
  - If executed properly, this key exchange is protected against man-in-the-middle attacks

DHKE with certificates

1. After agreeing on the DHKE parameters α and *p*, Alice and Bob each generate their own respective public-private key pairs
2. Then Alice and Bob each request a CA to sign their respective public keys, and they obtain a corresponding certificate from the CA
   - Note: the certificates are typically signed using the RSA digital signature protocol
3. Now Alice and Bob send their respective certificates to each other (instead of their respective public keys)
4. Alice and Bob separately verify that the certificates are authentic using the **public key of the CA**, $k_{pub,\ CA}$
5. They then extract their respective public keys from the certificates and use those public keys to compute a joint secret session key

DHKE with certificates: comments

- In order to verify the certificates, Alice and Bob require the public key of the CA
  - This key needs to be transmitted over an authenticated channel
  - Otherwise, Oscar can carry out a man-in-the-middle attack again

- So it looks like we are back to square one…

DHKE with certificates: comments

- Fortunately, that's not the case
- Even though we require an authenticated channel for transmission of the public keys of the CA, this channel is only needed **once**, during set-up time
  - In practice, the public keys of major CAs are already pre-installed in many web browsers and operating systems
- What's happening here is a ***transfer of trust***
  - In the basic DHKE without certificates, Alice and Bob have to trust each other's public keys directly
  - With certificates, they only need to trust the public key of the CA, $k_{pub,\ CA}$
  - When the CA signs other users' public keys, Alice and Bob can trust those too
  - This is known as a *chain of trust*

Public-key infrastructure (PKI)

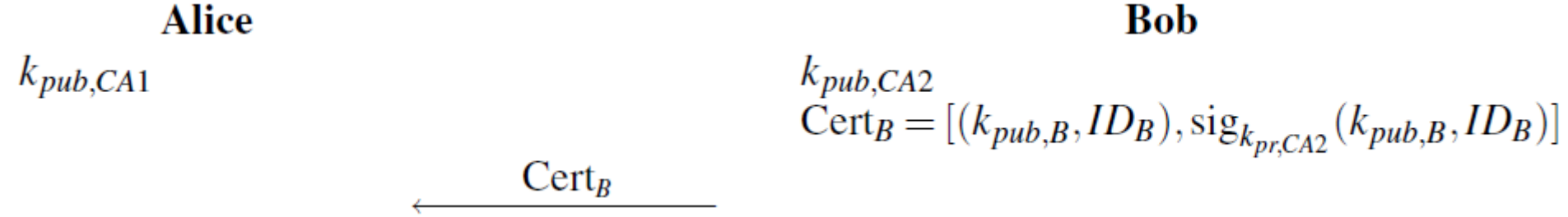
- Ideally, we would have a single CA that generates and issues certificates for all internet users in the world
- That's not the case in real life
  - There are many different entities that act as CAs
  - Many countries have their own official CA
  - Certificates for websites are issued by more than 50 commercial entities
- The entire system that is formed by the CAs, together with the necessary support mechanisms, is called a *public-key infrastructure*

- We'll just discuss some aspects of PKI and the use of certificates in practice

PKI and chain of CAs

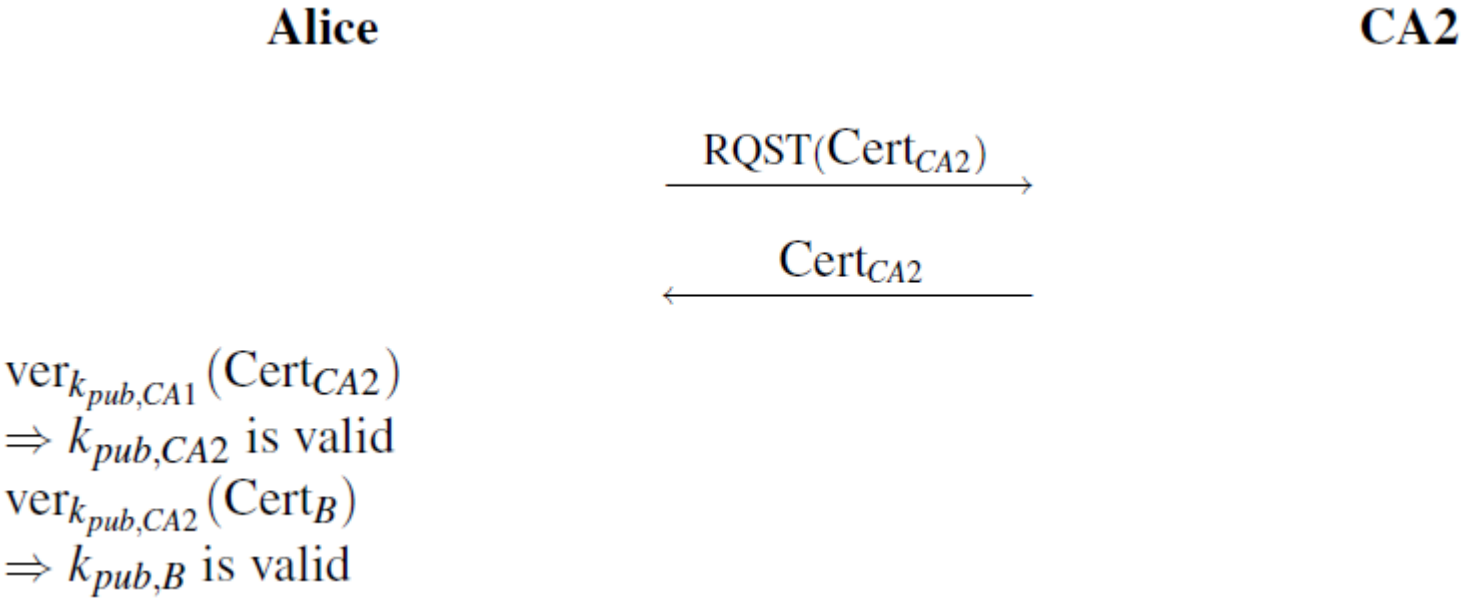
- Although the public keys of most major CAs (particularly *root* CAs) are pre-installed on web browsers and operating systems, we cannot expect the average user to possess the public keys of all CAs in the world
- There will be situations where Alice has the public keys and certificates of some CAs, while Bob has the public keys (and certificates) of other CAs that Alice does not possess
- So in a PKI, we have CAs certifying each other
  - This allows for Alice to request for a certificate of a CA that she does not have
  - That certificate will be signed/certified by a CA whose public key is already in Alice's possession

Chain of CAs: example



- Suppose only Alice possesses the public key of CA1, $k_{pub,\ CA1}$, while only Bob possesses the public key of CA2, $k_{pub,\ CA2}$
- Alice needs Bob's public key $k_{pub,\ B}$ (say, to do a DHKE)
- So Bob sends Alice his certificate containing his public key that is signed by CA2
- At this point, Alice cannot verify the authenticity of Bob's certificate because she does not have the public key of CA2

Chain of CAs: example



- So Alice requests CA2 to send her its public key, $k_{pub,\ CA2}$
- CA2 sends Alice its public key $k_{pub,\ CA2}$, in a certificate which is signed by CA1
  - The certificate would look like: $Cert_{CA2} = [(k_{pub,\ CA2}, ID_{CA2}), sig_{k_{pr,\ CA1}}(k_{pub,CA2}, ID_{CA2})]$
- Now Alice can verify the authenticity of CA2's certificate, then use its public key, $k_{pub,\ CA2}$, to verify Bob's certificate containing his public key

Chain of CAs: example

- In this example, a certificate chain is being established
- CA1 trusts CA2, so CA1 is able to sign the certificate (and public key $k_{pub,\ CA2}$) issued by CA2
- Alice can then trust Bob's public key $k_{pub,\ B}$ because it was contained in a certificate signed by CA2
- This is another case of a *chain of trust*, and trust is *delegated* in this situation

- In practice, CAs can be arranged in a hierarchical manner, where each CA only signs the public keys of the CAs one level below, with *root* CAs sitting at the top level

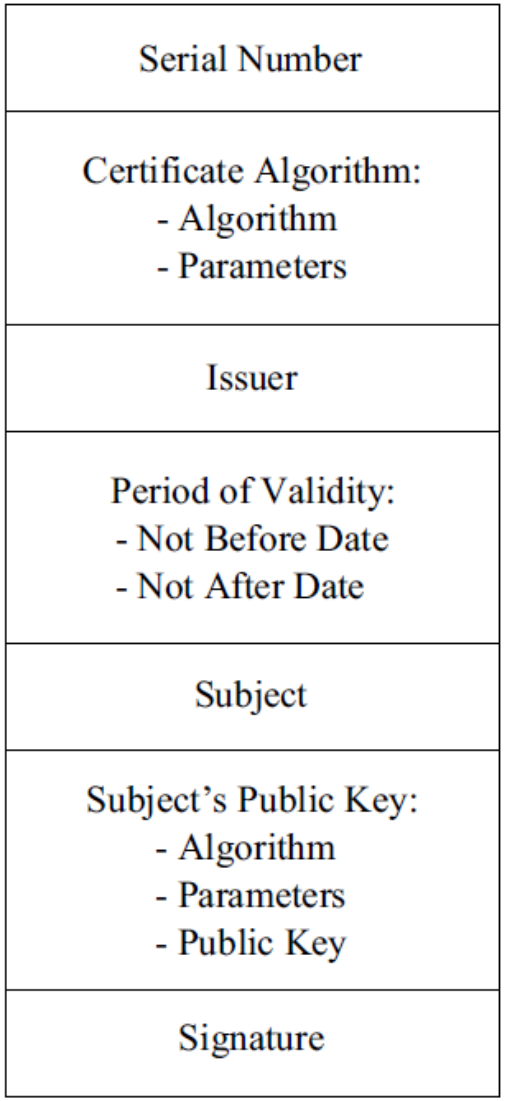
PKI and certificates

- In practice, certificates contain not only an ID, a public key and a signature; they also have many additional informative fields
  - E.g. the certificate algorithm used, the issuer, the period of validity, etc.

- X.509 is an important standard for network authentication services
  - Its certificates are widely used in internet communications, e.g. IPsec and SSL/TLS

X.509 certificates

- Let's take a look at some of the fields defined in a X.509 certificate
- **Certificate Algorithm**: the cryptoalgorithm used to sign the certificate, e.g. RSA digital signature protocol
- **Period of Validity**: in most cases, a public key is not certified indefinitely, but for one or two years; this limits the amount of damage if the corresponding private key is ever compromised
- **Subject's Public Key**: this is the public key that is to be protected by the certificate. Note that algorithm associated with the public key is **not** necessarily the same as that used by the certificate itself



PKI and certificate revocation lists

- We also need a way to revoke certificates
  - This allows for certificates on a lost device to be invalidated
  - Also allows for revocation of certificates belonging to an individual who is no longer affiliated with an organization

- This can be done with a *certificate revocation list* (CRL)
  - The serial numbers of the certificates can be used to identify invalid certificates
  - A CRL must be signed by a CA

PKI and certificate revocation lists

- There are a few problems with CRLs:
  - A CRL has to be updated frequently
  - A CRL needs to be checked every time a certificate is validated
  - This can lead to network performance issues

- Alternatively, the updates to the CRLs can be sent only periodically
  - This leads to a time lag between the invalidation of a certificate and the notification of the users

- Need to seek a balance between the competing goals of security and performance

Alternative to PKI: Web of Trust

- Another way to authenticate public keys is use a *Web of Trust* concept
  - Main idea: decentralized model, no CAs; certificates are signed by other users
  - If Alice trusts Bob, then it is assumed that she will also trust all other users whom Bob trusts
- Some real-life implementations include Pretty Good Privacy (PGP) and GNU Privacy Guard (GPG)
- Not very widely adopted