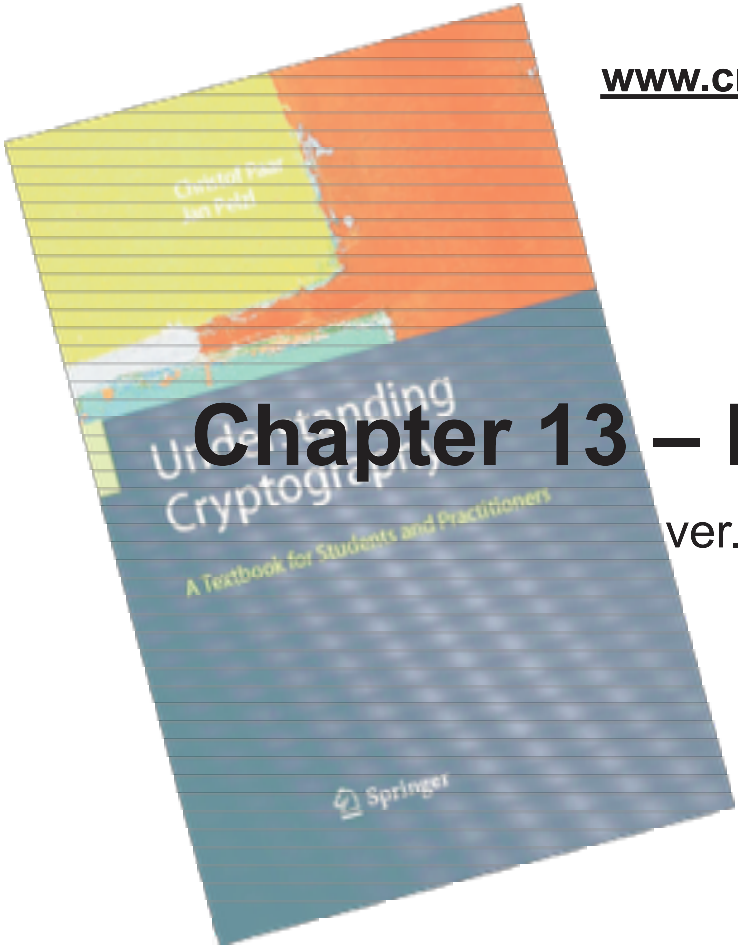
Understanding Cryptography

by Christof Paar and Jan Pelzl

www.crypto-textbook.com



Chapter 13 – Key Establishment

ver. Jan 7, 2010

These slides were prepared by Christof Paar and Jan Pelzl
and modified by Sam Bowne

Updated 12-4-17

Some legal stuff (sorry): Terms of Use

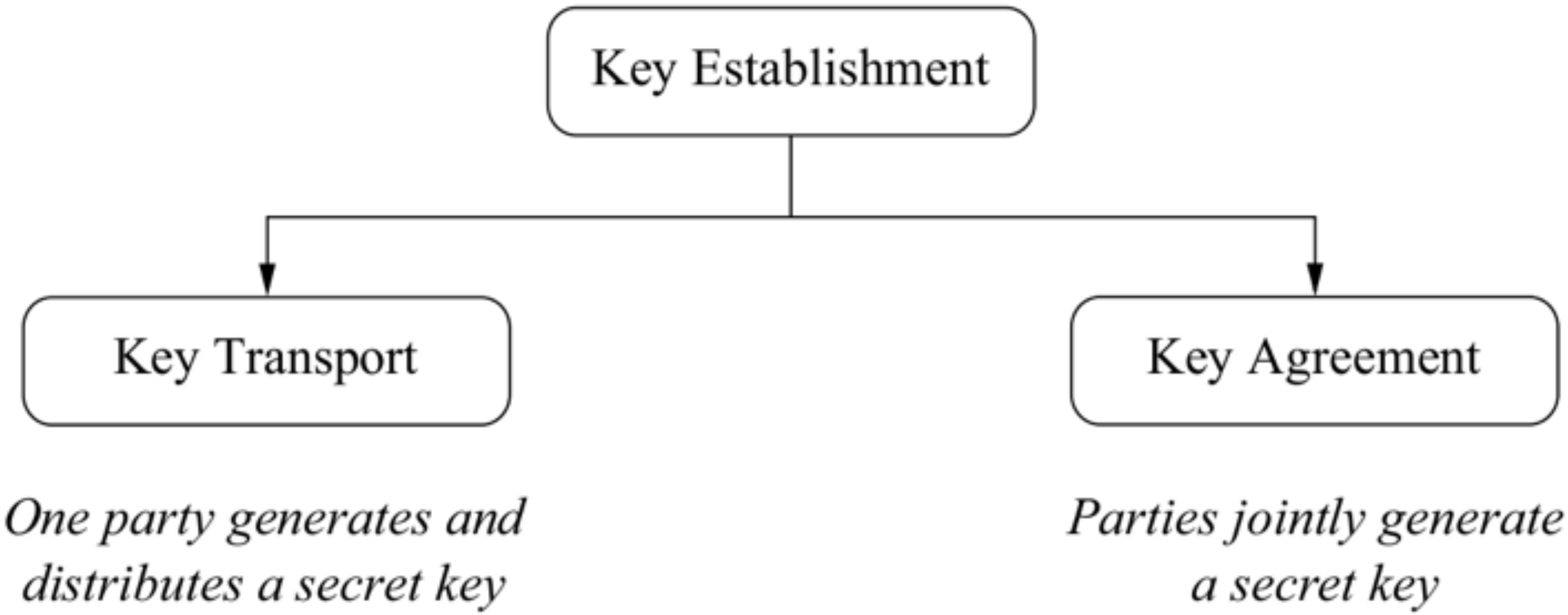
- The slides can used free of charge. All copyrights for the slides remain with Christof Paar and Jan Pelzl.
- The title of the accompanying book "Understanding Cryptography" by Springer and the author's names must remain on each slide.
- If the slides are modified, appropriate credits to the book authors and the book title must remain within the slides.
- It is not permitted to reproduce parts or all of the slides in printed form whatsoever without written consent by the authors.

Contents of this Chapter

13.1

Introduction

Classification of Key Establishment Methods



In an ideal key agreement protocol, no single party can control what the key value will be.

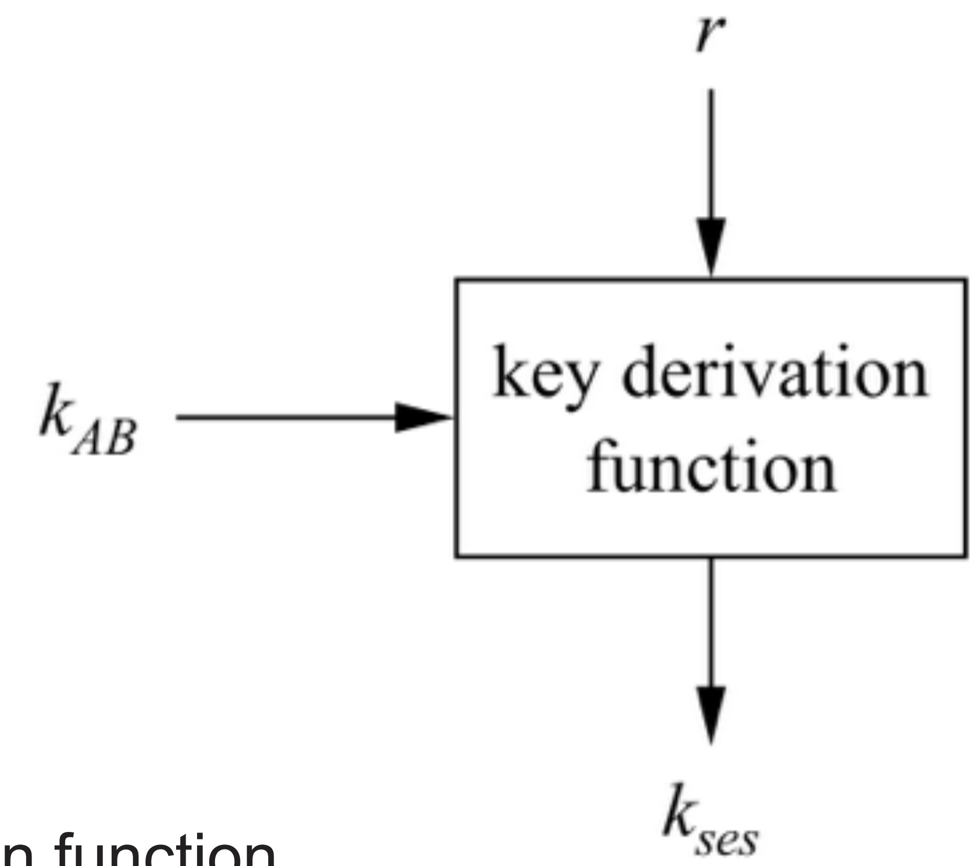
Key Freshness

It is often desirable to frequently change the key in a cryptographic system. Reasons for key freshness include:

- If a key is exposed (e.g., through hackers), there is limited damage if the key is changed often
- Some cryptographic attacks become more difficult if only a limited amount of ciphertext was generated under one key
- If an attacker wants to recover long pieces of ciphertext, he has to recover several keys which makes attacks harder

Key Derivation

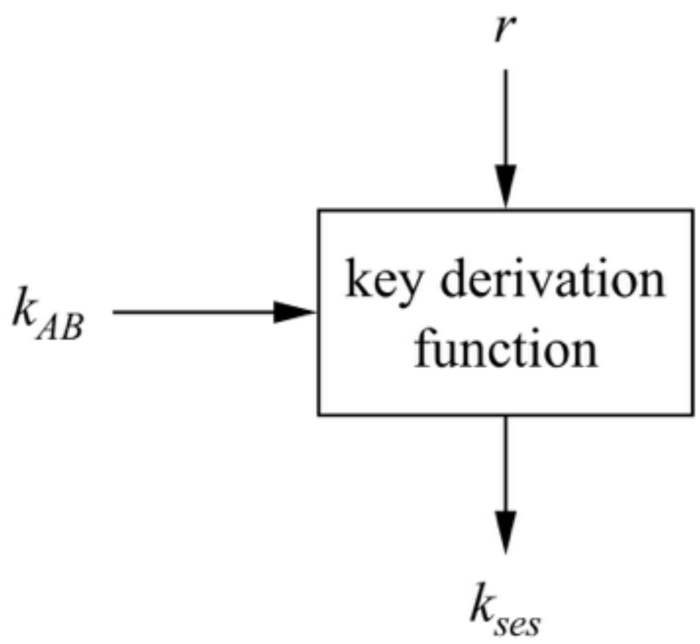
- In order to achieve key freshness, we need to generate new keys frequently.
- Rather than performing a full key establishment every time (which is costly in terms of computation and/ or communication), we can **derive multiple session keys k_{ses} from a given key k_{AB}.**



- The key k_{AB} is fed into a key derivation function together with a nonce r (number used only once).
- Every different value for r yields a different session key

Key Derivation

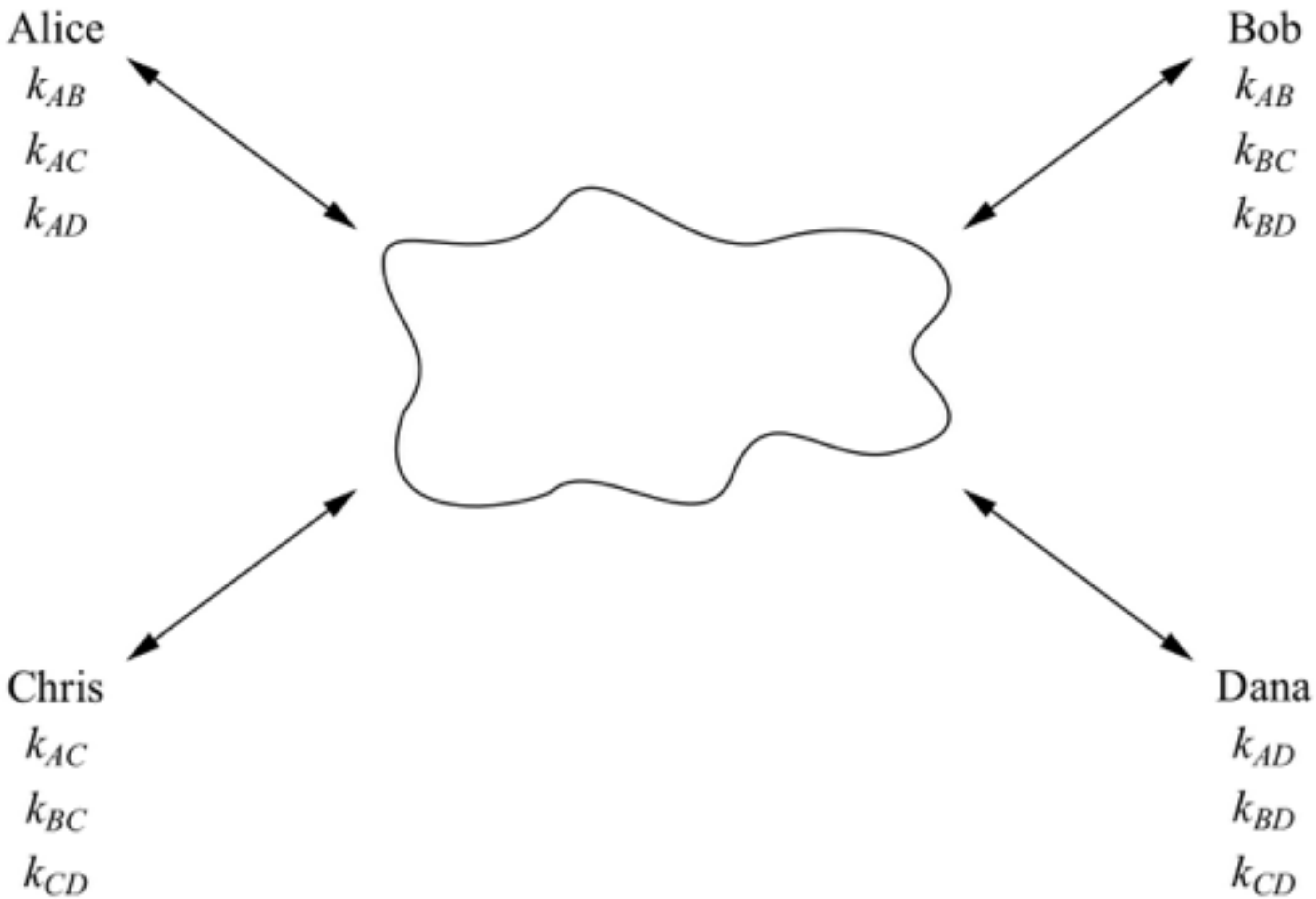
- The key derivation function is a computationally simple function, e.g., a block cipher or a hash function
- Example for a basic protocol:



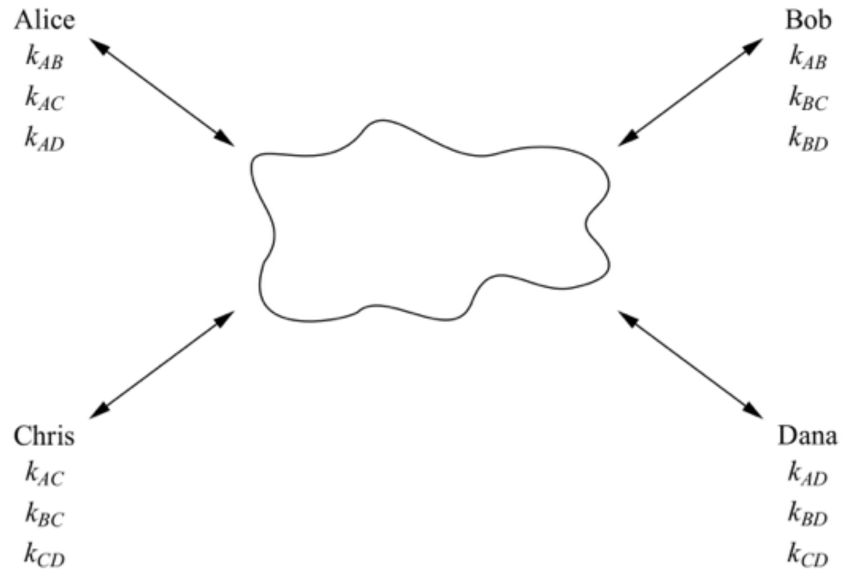
Alice		**Bob**
		generate nonce r
	$\xleftarrow{\quad r \quad}$	
derive session key $K_{ses} = e_{kAB}(r)$		derive session key $K_{ses} = e_{kAB}(r)$

The n^2 Key Distribution Problem

- **n** users. Every user wants to communicate securely with every of the other **n-1** users. Every pair of users needs an individual key pair.



The n^2 Key Distribution Problem



Shortcomings

- There are $n(n-1) \approx n^2$ keys in the system
- There are $n(n-1)/2$ key pairs
- If a new user Esther joins the network, new keys k_{XE} have to be transported via secure channels (!) to each of the existing users

⇒ Only works for small networks which are relatively static

Example: mid-size company with 750 employees

- 750 x 749 = 561,750 keys must be distributed securely

13.2

Key Establishment Using Symmetric-Key Techniques

Key Establishment with Key Distribution Center

- Key Distribution Center (KDC) = Central party, trusted by all users
- KDC shares a **key encryption key (KEK)** with each user
- Principle: KDC sends session keys to users which are encrypted with KEKs

Alice		**KDC**		**Bob**
KEK: k_A		KEKs: k_A , k_B		KEK: k_B
		Generate session key k_{ses}		
	$\xleftarrow{y_A}$	$y_A = e_{KA}(k_{ses})$	$\xrightarrow{y_B}$	
		$y_B = e_{KB}(k_{ses})$		
derive session key $K_{ses} = e_{KA}(y_A)$				derive session key $K_{ses} = e_{KB}(y_B)$

message y $y = e_{Kses}(x)$	$\xrightarrow{y}$	$x = e^{-1}_{Kses}(y)$

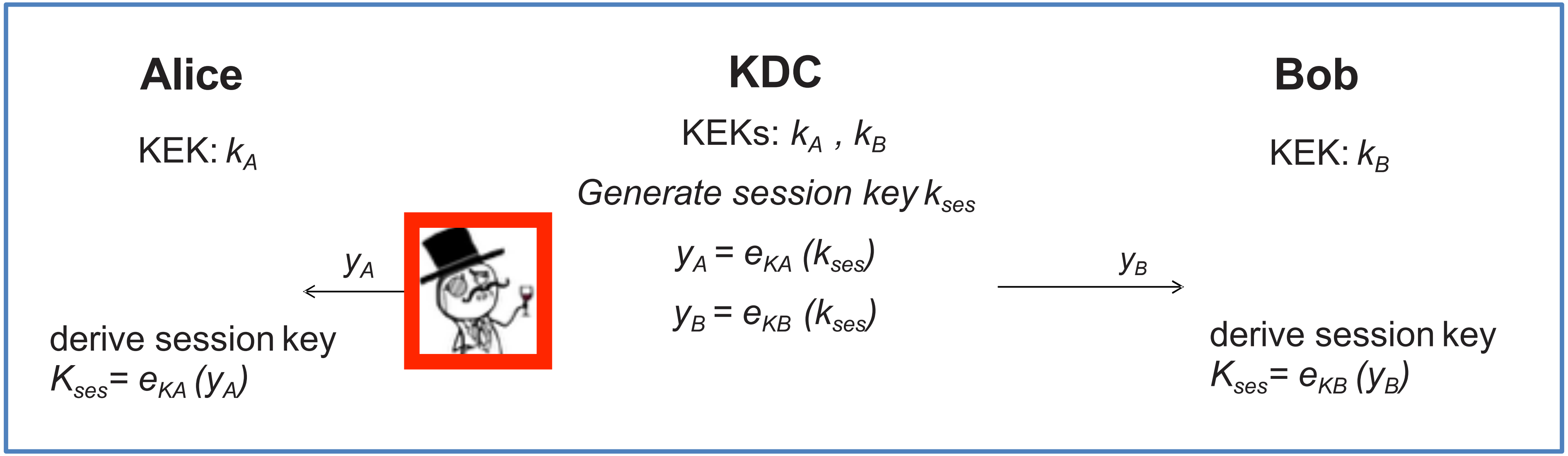
Key Establishment with Key Distribution Center

- Advantages over previous approach:
  - Only *n* long-term key pairs are in the system
  - If a new user is added, a secure key is only needed between the user and the KDC (the other users are not affected)
  - Scales well to moderately sized networks

- *Kerberos* (a popular authentication and key distribution protocol) is based on KDCs

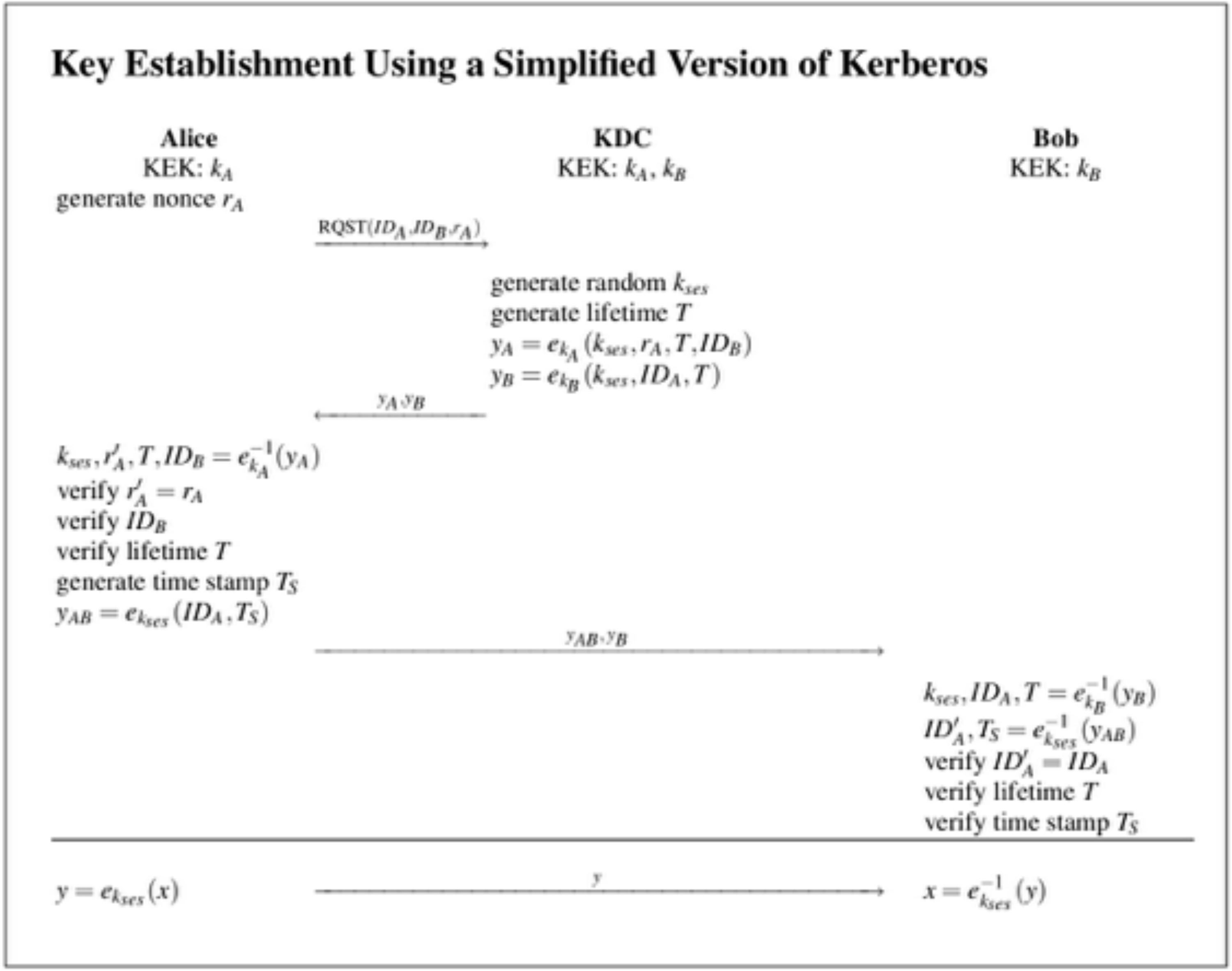
Key Confirmation Attack (MITM)

- MITM attacker sends Alice a fake key
  - (And a fake request to the KDC)
- She can't detect this because there is no ***key confirmation***



Kerberos

- Provides **key confirmation** and **user authentication**
  - Alice sends a random nonce $\boldsymbol{r_A}$ to the server
  - Server encrypts it with their joint KEK $\boldsymbol{k_A}$
  - Alice verifies that the decrypted nonce matches
- Resists **MITM** attacks



Kerberos

- Session keys have a limited lifetime
  - Clocks on all devices must be synchronized within a few minutes

Key Establishment Using a Simplified Version of Kerberos

Alice		**KDC**		**Bob**
KEK: k_A		KEK: k_A, k_B		KEK: k_B
generate nonce r_A				
	$\xrightarrow{RQST(ID_A, ID_B, r_A)}$			
		generate random k_{ses} generate lifetime T $y_A = e_{k_A}(k_{ses}, r_A, T, ID_B)$ $y_B = e_{k_B}(k_{ses}, ID_A, T)$		
	$\xleftarrow{y_A, y_B}$			
$k_{ses}, r'_A, T, ID_B = e^{-1}_{k_A}(y_A)$ verify $r'_A = r_A$ verify ID_B verify lifetime T generate time stamp T_S $y_{AB} = e_{k_{ses}}(ID_A, T_S)$				
	$\xrightarrow{y_{AB}, y_B}$			
				$k_{ses}, ID_A, T = e^{-1}_{k_B}(y_B)$ $ID'_A, T_S = e^{-1}_{k_{ses}}(y_{AB})$ verify $ID'_A = ID_A$ verify lifetime T verify time stamp T_S
$y = e_{k_{ses}}(x)$	$\xrightarrow{y}$			$x = e^{-1}_{k_{ses}}(y)$

Remaining Problems with Symmetric-Key Distribution

- **No *Perfect Forward Secrecy***: If the KEKs are compromised, an attacker can decrypt past messages if he stored the corresponding ciphertext
- **Single point of failure**: The KDC stores all KEKs. If an attacker gets access to this database, all past traffic can be decrypted.
- **Communication bottleneck**: The KDC is involved in every communication in the entire network (can be countered by giving the session keys a long life time)
- **Key confirmation attack (MITM)**

13.3

Key Establishment Using Asymmetric Techniques

Recall: Diffie–Hellman Key Exchange (DHKE)

Alice | **Bob**

Public parameters α, p

Alice:

Choose random private key
$k_{prA} = a \in \{1, 2, \ldots, p\text{-}1\}$

Compute public key
$k_{pubA} = A = \alpha^a \bmod p$

Bob:

Choose random private key
$k_{prB} = b \in \{1, 2, \ldots, p\text{-}1\}$

Compute public key
$k_{pubB} = B = \alpha^b \bmod p$

$$\xrightarrow{\quad A \quad}$$

$$\xleftarrow{\quad B \quad}$$

Alice:

Compute common secret
$k_{AB} = B^a = (\alpha^a)^b \bmod p$

Bob:

Compute common secret
$k_{AB} = A^b = (\alpha^b)^a \bmod p$

- Widely used in practice
- If the parameters are chosen carefully (especially a prime $p > 2^{1024}$), the DHKE is secure against *passive* (i.e., listen-only) attacks
- However: If the attacker can *actively* intervene in the communciation, the man-in-the-middle attack becomes possible

Man-in-the-Middle Attack

- The man-in-the-middle-attack is not restricted to DHKE; it is applicable to any public-key scheme, e.g. RSA encryption. ECDSA digital signature, etc. etc.
- Q: *What is the underlying problem that makes the MIM attack possible?*
- A: The public keys are not **authenticated**
- When Alice receives a public key which is allegedly from Bob, she has no way of knowing whether it is in fact his. (After all, a key consists of innocent bits; it does not smell like Bob's perfume or anything like that)
- Even though public keys can be sent over **insecure channels** (lacking confidentiality), they require **authenticated channels** (with authentication and integrity)

Certificates

- In order to authenticate public keys (and thus, prevent the MIM attack), all public keys are digitally signed by a central trusted authority.
- Such a construction is called *certificate*

certificate = public key + ID(user) + digital signature over public key and ID

- In its most basic form, a certificate for the key k_{pub} of user Alice is:

$$\mathbf{Cert(Alice) = (}\boldsymbol{k_{pub}, ID(Alice), sig_{KCA}}\mathbf{(}\boldsymbol{k_{pub}}\mathbf{,ID(Alice)\ )}$$

- Certificates bind the identity of user to her public key
- The trusted authority that issues the certificate is referred to as ***certifying authority (CA)***
- "Issuing certificates“ means in particular that the CA computes the signature $sig_{KCA}(k_{pub})$ using its (super secret!) private key k_{CA}
- The party who receives a certificate, e.g., Bob, verifies Alice‘s public key using the public key of the CA

Diffie–Hellman Key Exchange (DHKE) with Certificates

CA

$\xleftarrow{Cert(Alice)}$ $\xrightarrow{Cert(Bob)}$

Alice

$k_{prA} = a$

$k_{pubA} = A$

$Cert(Alice) = ((A, ID_A), sig_{KCA}(A, ID_A))$

Bob

$k_{prB} = b$

$k_{pubB} = B = \alpha^b \bmod p$

$Cert(Bob) = ((B, ID_B), sig_{KCA}(B, ID_B))$

$\xrightarrow{Cert(Alice)}$

$\xleftarrow{Cert(Bob)}$

Alice:

verify certificate
$ver_{Kpub,CA}(Cert(Bob))$

if verification is correct:
Compute common secret
$k_{AB} = B^a = (\alpha^a)^b \bmod p$

Bob:

verify certificate
$ver_{Kpub,CA}(Cert(Alice))$

if verification is correct:
Compute common secret
$k_{AB} = A^b = (\alpha^b)^a \bmod p$

Certificates

- Note that verification requires the public key of the CA for $ver_{K_{pub,CA}}$
- In principle, an attacker could run a MITM attack when $k_{pub,CA}$ is being distributed

  $\Rightarrow$ The public CA keys must also be distributed via an authenticated channel!

- Q: So, have we gained anything?

  After all, we try to protect a public key (e.g., a DH key) by using yet another public-key scheme (digital signature for the certificate)?

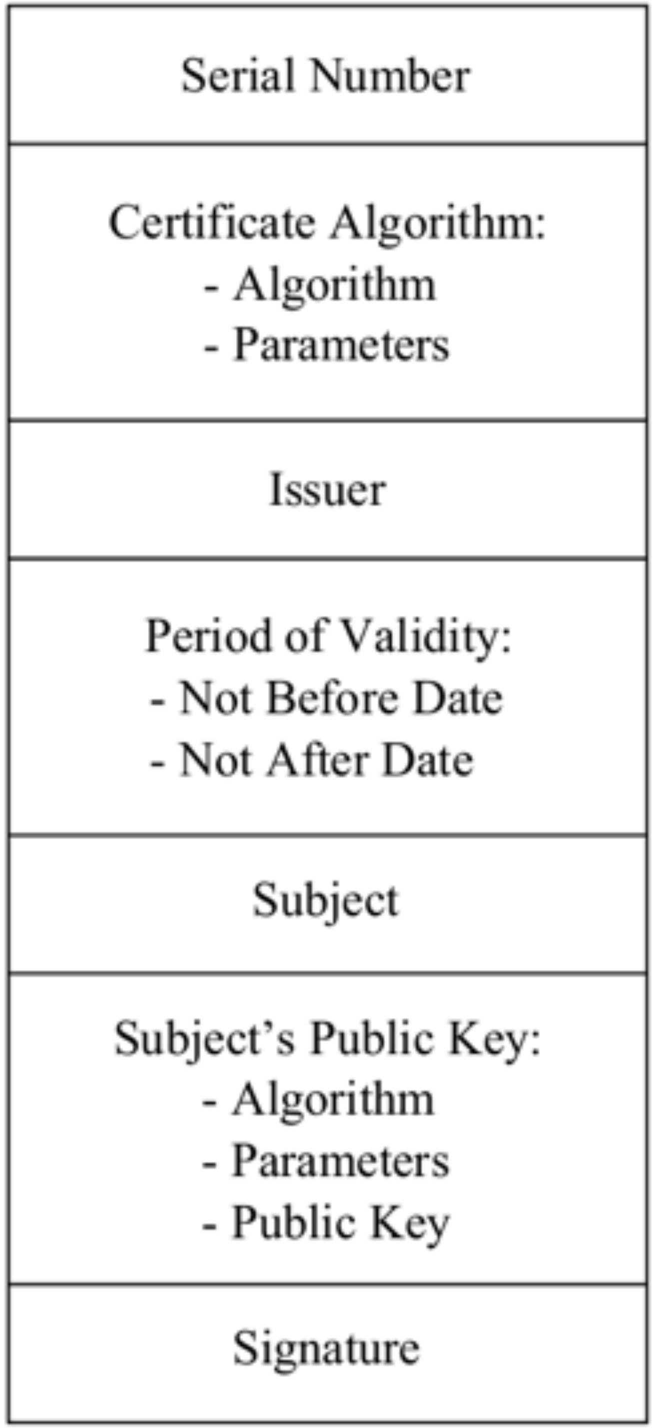
- A: YES! The difference from before (e.g., DHKE without certificates) is that **we only need to distribute the public CA key *once***, often at the set-up time of the system
- Example: Most web browsers are shipped with the public keys of many CAs. The "authenticated channel" is formed by the (hopefully) correct distribution of the original browser software.

Public-Key Infrastructure

- Definition
  - The entire system: Certificate Authorities (CAs) together with the necessary support mechanisms

Certificates in the Real World

- X.509 certificates contain much more information than just a public key and a signature.
- **Signature** at the bottom is computed over all other fields in the certifcate (after hashing of all those fields).
- Every certificate uses **two** public-key schemes
  1. The public key that actually is protected by the signature ("Subject‘s Public Key“ on the right). e.g. public Diffie-Hellman key
  2. The digital signature algorithm used by the CA to sign the certificate data.



Remaining Issues with PKIs

1. Users communicate which other whose certificates are issued by different CAs
   - This requires cross-certification of CAs, e.g.. CA1 certifies the public-key of CA2. If Alice trusts "her“ CA1, cross-certification ensures that she also trusts CA2. This is called a "chain of trust“ and it is said that "trust is delegated“.

2. Certificate Revocation Lists (CRLs)
   - Another real-world problem is that certificates must be revoked, e.g., if a smart card with certificate is lost or if a user leaves an organization. This system is problematic in practice because not all CAs maintain fast, reliable CRL servers, and browsers often fail-open.

Kahoot!