Symmetric Key Establishment

2019. 4. 16

Contents

- Introduction

- Symmetric-key cryptography
  - Block ciphers
  - Symmetric-key algorithms
  - Cipher block modes
  - Stream cipher

- Public-key cryptography
  - RSA
  - Diffie-Hellman
  - ECC
  - Digital signature
  - Public key Infrastructure

- Cryptographic hash function
  - Attack complexity
  - Hash Function algorithm

- Integrity and Authentication
  - Message authentication code
  - Authentication encryption
  - Digital signature

- Symmetric Key establishment
  - Public-key based
  - Key agreement (Diffie-Hellman)
  - server-based

- Key Wrap/Random Number Generation

Key establishment

- establishing symmetric key
  - How are the secret keys in the symmetric key encryption distributed and managed?
- distributing public key
  - When a public key is known in the public domain, how can I trust that the key is really his or her public key to be claimed?
  - For this topic, we already discuss how public keys are distributed in a trusted way in real world.

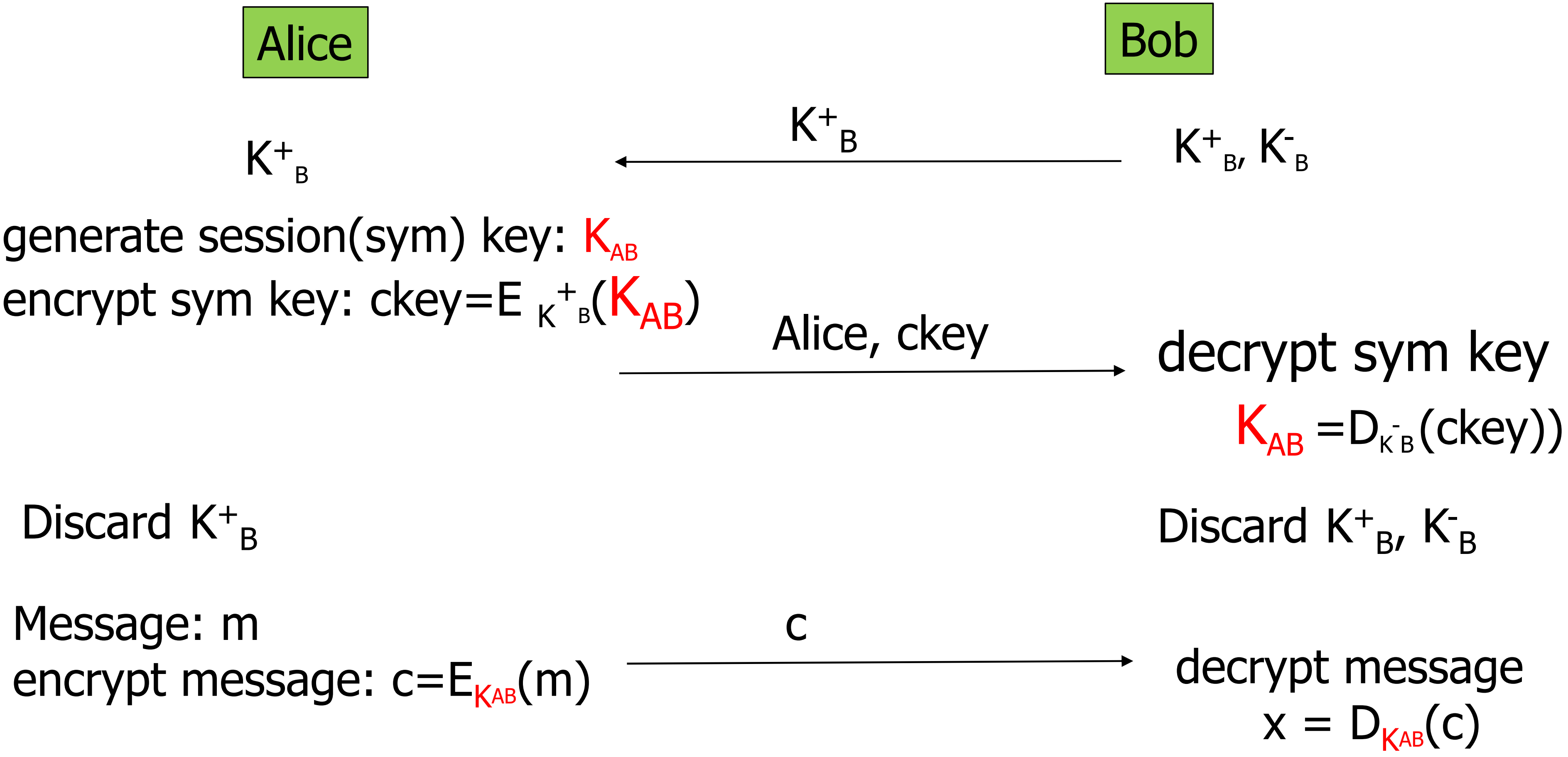
Symmetric key establishment

- Key transportation using public key encryption
  - One of the parties generates a key
  - Then, one party transport the key to the other party.
- Key agreement
  - Key is a function of inputs by two parties
  - Ex, Diffie-Hellman
- Key establishment using symmetric encryption
  - Based on KDC

Symmetric key encryption using public key

- Normally, the public key algorithm is almost never used for encrypting sizable blocks of data because of its a long execution time.
- Typical use of the public key algorithm is to encrypt a symmetric key which does not take much cost.
  - A sender encrypts a symmetric key by the receiver's public key.
  - Then, sends the encrypted symmetric key with its identity.
  - The receivers recovers the sym key by using his private key.
  - Then discards the public and private keys.
  - After that, they can encrypt messages by using the shared symmetric key.

Symmetric key transportation



Alice

Bob

K^+_B

K^+_B

K^+_B, K^-_B

generate session(sym) key: K_{AB}

encrypt sym key: ckey=$E_{K^+_B}(K_{AB})$

Alice, ckey

decrypt sym key

$K_{AB} = D_{K^-_B}(\text{ckey}))$

Discard K^+_B

Discard K^+_B, K^-_B

Message: m

encrypt message: c=$E_{K_{AB}}(m)$

c

decrypt message

$x = D_{K_{AB}}(c)$

Using public key encryption

- The previous simple protocol is not secure against a man-in-the-middle(MIM) attack.
  - How we can prevent this attack will be discussed in the key exchange scheme.
- We already learned that one of the public key applications is to use for establishing symmetric keys.
- Drawback
  - Must trust the public key.
  - To do that, we need PKI.

Session key

- Session key is an ephemeral key to be used for encrypting messages belonging to one session.
- A session key is generated and used during a session. After that, it is thrown away.
- So, a user has a master key which is used permanently until it is updated, and a session key for encryption for temporary use.
- Why do they need session keys, instead of one key?
- How can they have master keys?

Perfect Forward Secrecy

- Consider this “issue”
  - Alice encrypts message with shared key K and sends ciphertext to Bob
  - An attacker records ciphertext and later attacks Alice’s (or Bob’s) computer to recover K
  - Then he decrypts recorded messages
- Perfect forward secrecy (PFS):
  - Even if an attacker gets key K or other secret(s) later, he should not decrypt all past communicated messages.
- Is PFS possible?

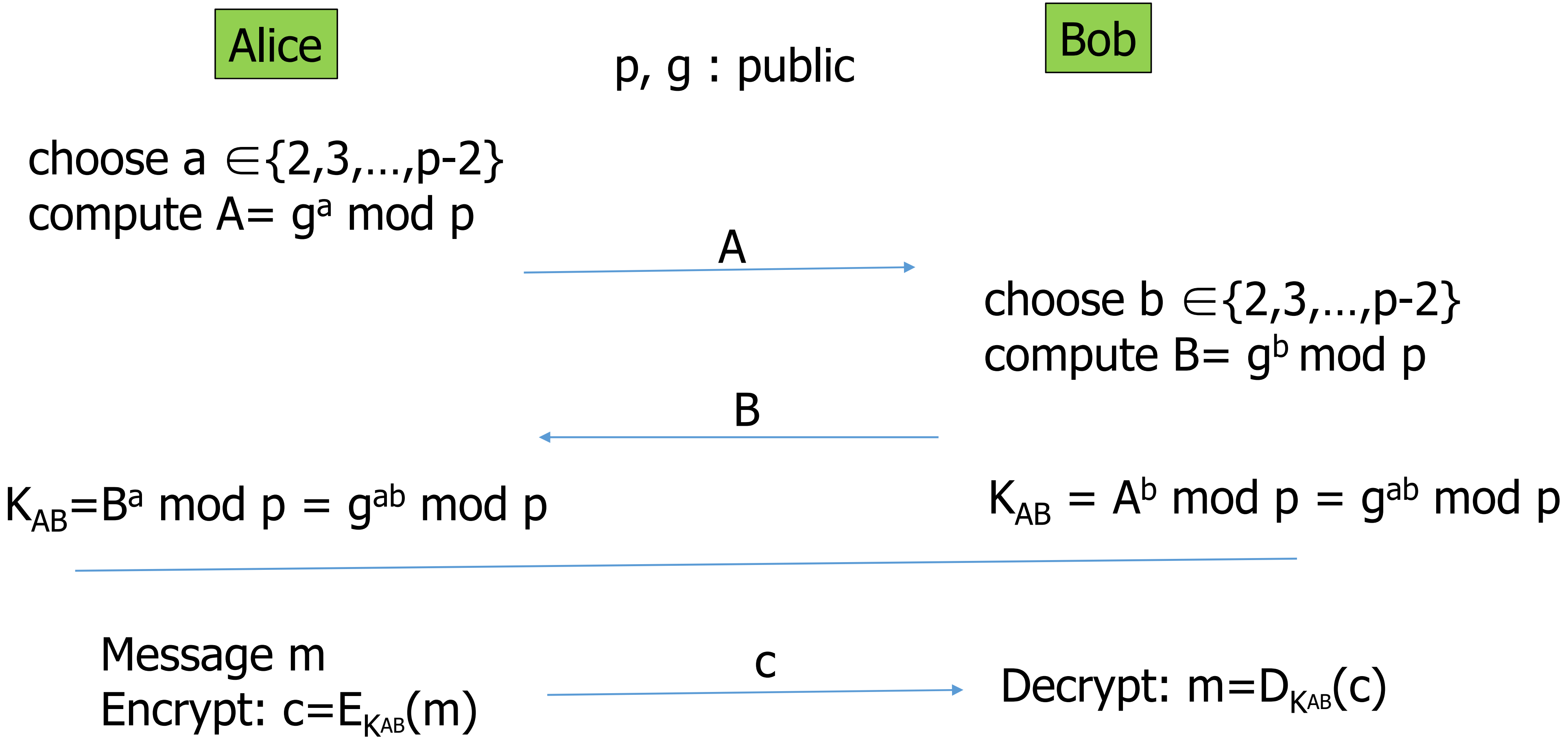
Perfect Forward Secrecy

- Suppose Alice and Bob share a key K
- For perfect forward secrecy, Alice and Bob don’t use K to encrypt.
- Instead they must use a session key K_S and forget it after it’s used.
- Is a session key K_S enough to ensure PFS?

Key agreement

- Use Diffie-Hellman(D-H) or EC-DH algorithm for Alice and Bob to share a secret key.
- D-H key agreement
  - Alice and Bob choose *p*, a large prime numbers p and g, a generator *g* of order p-1, letting them known in public.
  - Then do the procedures in the following slide.
  - The final result, $g^{ab} \bmod p$, can be used directly as a sym key or as secret information to compute a sym key.
  - They destroy a and b after computing a sym key. So, guarantee "Perfect Forward Secrecy (PFS)."

D-H key exchange



Alice

p, g : public

Bob

choose a $\in\{2,3,\ldots,p-2\}$
compute $A= g^a \bmod p$

A

choose b $\in\{2,3,\ldots,p-2\}$
compute $B= g^b \bmod p$

B

$K_{AB}=B^a \bmod p = g^{ab} \bmod p$

$K_{AB} = A^b \bmod p = g^{ab} \bmod p$

Message m
Encrypt: $c=E_{K_{AB}}(m)$

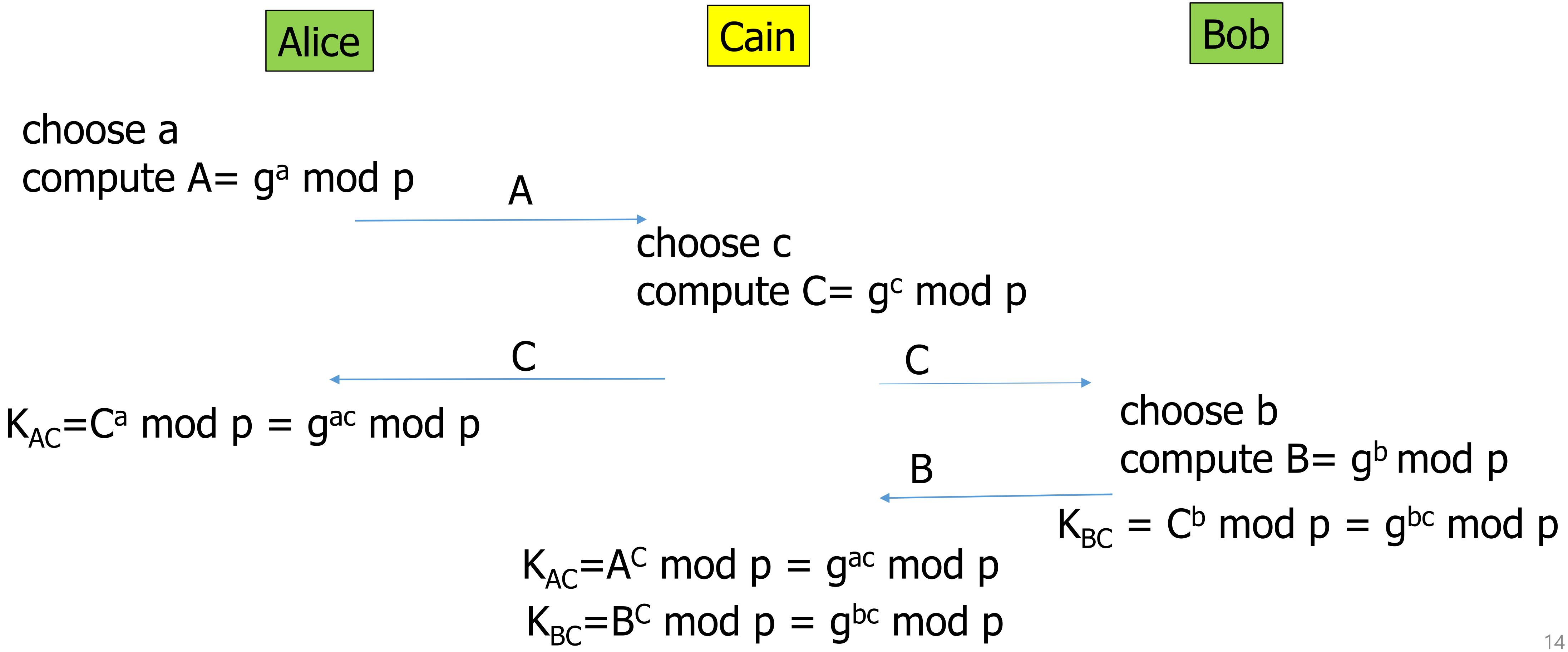
c

Decrypt: $m=D_{K_{AB}}(c)$

Security of D-H key agreement

- We already discussed the security of D-H algorithm.
  - It depends on the parameters, especially the size of p.
- Aside from the algorithm attack, D-H key agreement protocol is subject to the man-in-the-middle attack.

Man-in-the-middle(MIM) attack



Alice

Cain

Bob

choose a
compute A= g^a mod p

A →

choose c
compute C= g^c mod p

← C

C →

K_{AC}=C^a mod p = g^{ac} mod p

choose b
compute B= g^b mod p

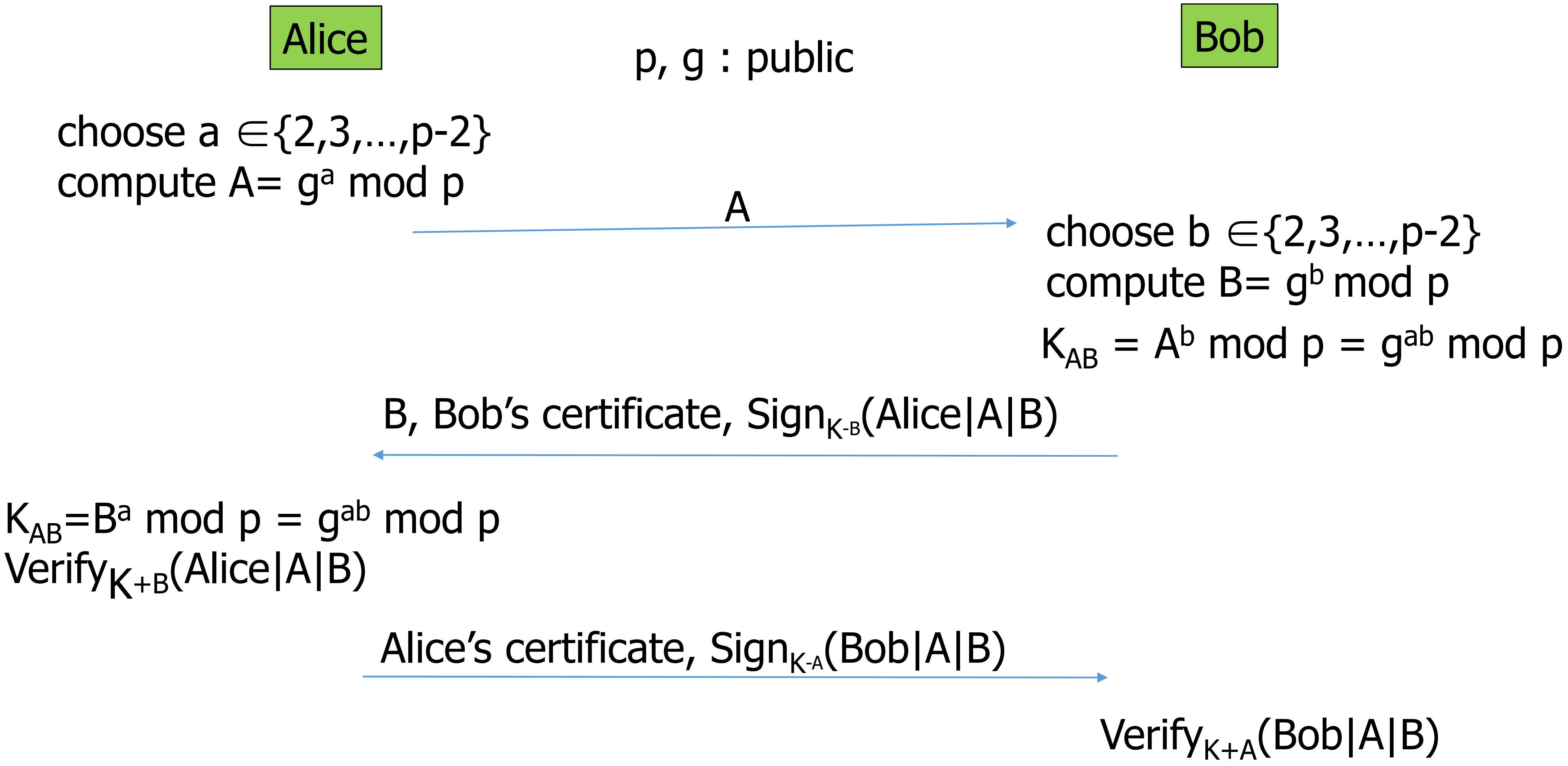
← B

K_{BC} = C^b mod p = g^{bc} mod p

K_{AC}=A^C mod p = g^{ac} mod p
K_{BC}=B^C mod p = g^{bc} mod p

How to prevent MIM attack

- Encrypt DH exchange with symmetric key
  - Sound like a silly answer
- Encrypt DH exchange with public key
- Sign DH values with private key(digital signature)
- Any other?



Alice

p, g : public

Bob

choose a $\in\{2,3,\ldots,p-2\}$
compute A= g^a mod p

A

choose b $\in\{2,3,\ldots,p-2\}$
compute B= g^b mod p
K_{AB} = A^b mod p = g^{ab} mod p

B, Bob's certificate, $\text{Sign}_{K\text{-}B}$(Alice|A|B)

K_{AB}=B^a mod p = g^{ab} mod p
Verify_{K+B}(Alice|A|B)

Alice's certificate, $\text{Sign}_{K\text{-}A}$(Bob|A|B)

Verify_{K+A}(Bob|A|B)

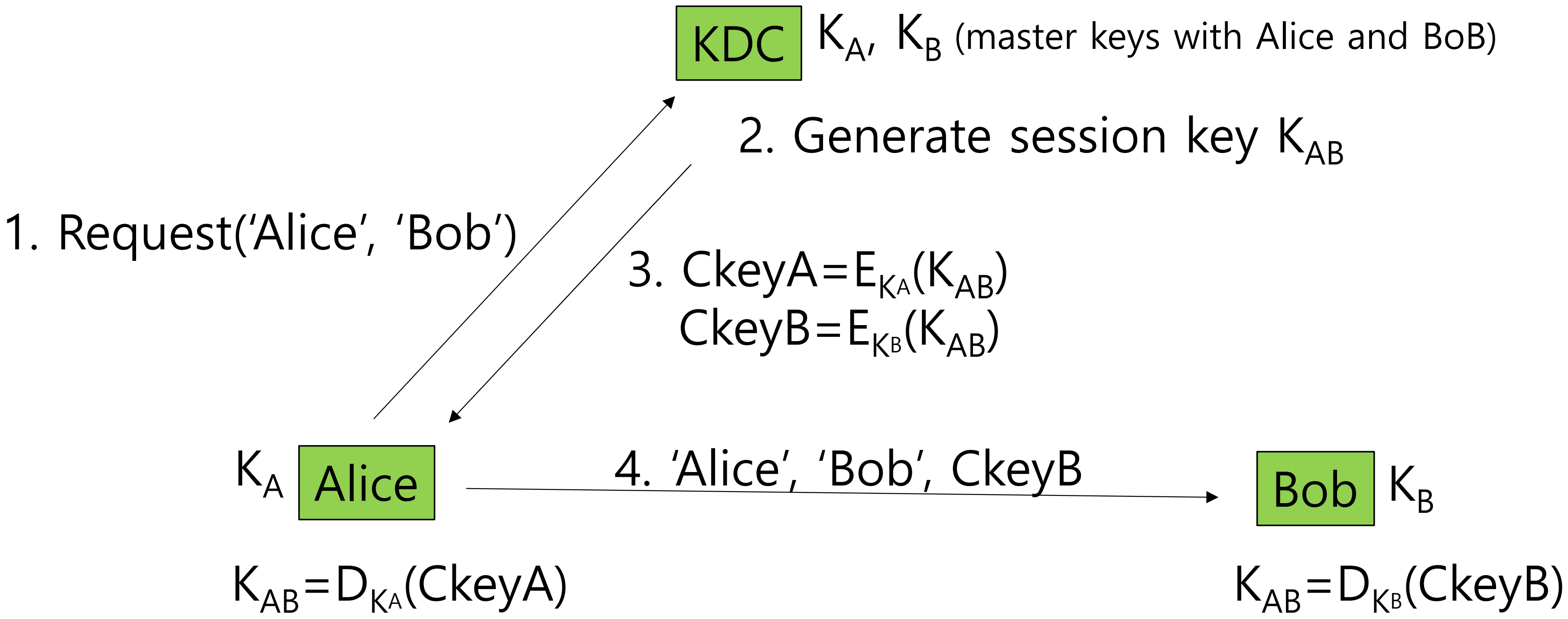
Remark:

- After all, in order to establish symmetric keys, we need public keys, which also bring about secure distribution of public keys.
- Then, the question is how we can establish symmetric keys without resort to public keys.

Key establishment using symmetric key

- Decentralized scheme
  - Establish key pairs between all users at initialization time
  - Drawback:
    - Large number of keys: keys pairs = n(n-1)/2
    - Adding new users is complex
- Centralized scheme
  - A central trusted authority(or authorities) which shares a key(often called master key) with every user distributes a key pair when requested.
  - A central trusted authority is often called a key distribution center(KDC).

simple key establishment using KDC



KDC K_A, K_B (master keys with Alice and BoB)

2. Generate session key K_{AB}

1. Request('Alice', 'Bob')

3. $CkeyA=E_{KA}(K_{AB})$
   $CkeyB=E_{KB}(K_{AB})$

K_A Alice

4. 'Alice', 'Bob', CkeyB

Bob K_B

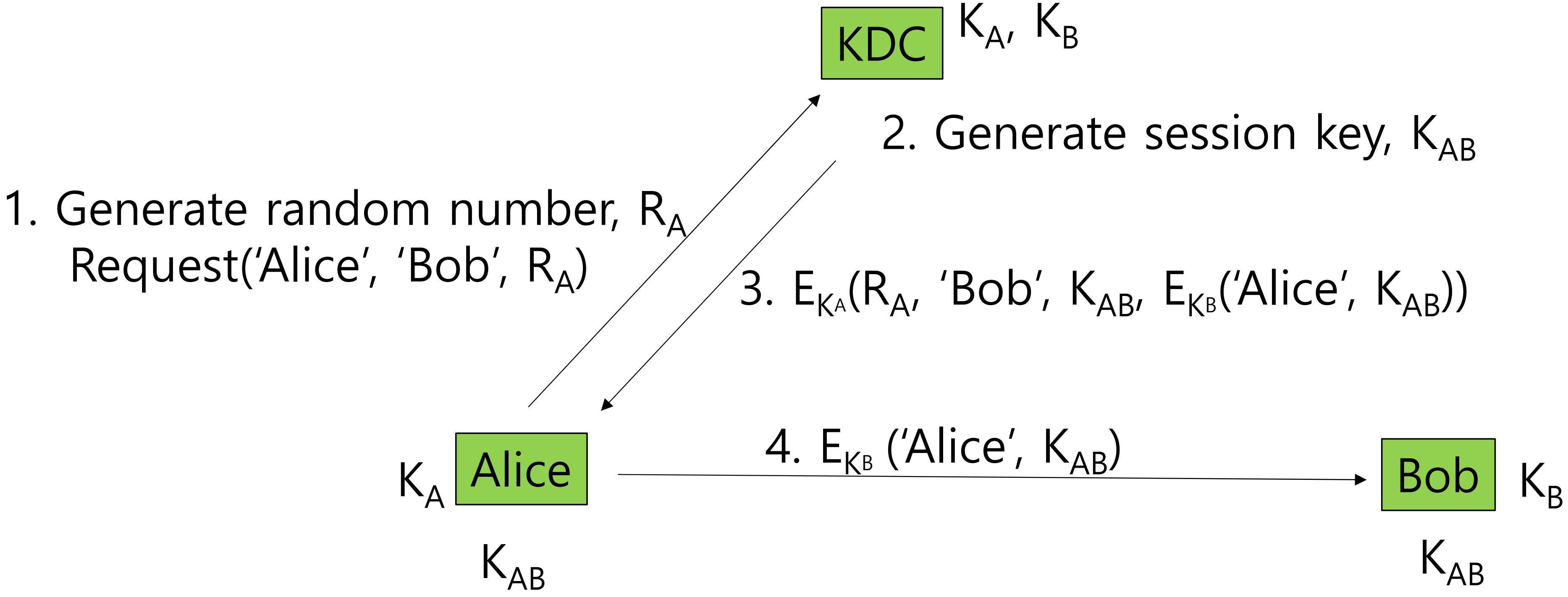
$K_{AB}=D_{KA}(CkeyA)$

$K_{AB}=D_{KB}(CkeyB)$

simple key establishment using KDC

- The keys, K_A and K_B are pre-installed at KDC and users.
- # of keys
  - When n users, there are n keys.
- Adding a new user only requires a secure channel between KDC and a new user at setup time.
- Drawbacks
  - KDC is a single point of failure.
  - No perfect forward secrecy
  - Replay attack

Elaborated establishment using KDC



KDC K_A, K_B

1. Generate random number, R_A
   Request('Alice', 'Bob', R_A)

2. Generate session key, K_{AB}

3. $E_{K_A}(R_A$, 'Bob', K_{AB}, E_{K_B}('Alice', K_{AB}))

4. E_{K_B} ('Alice', K_{AB})

K_A Alice

K_{AB}

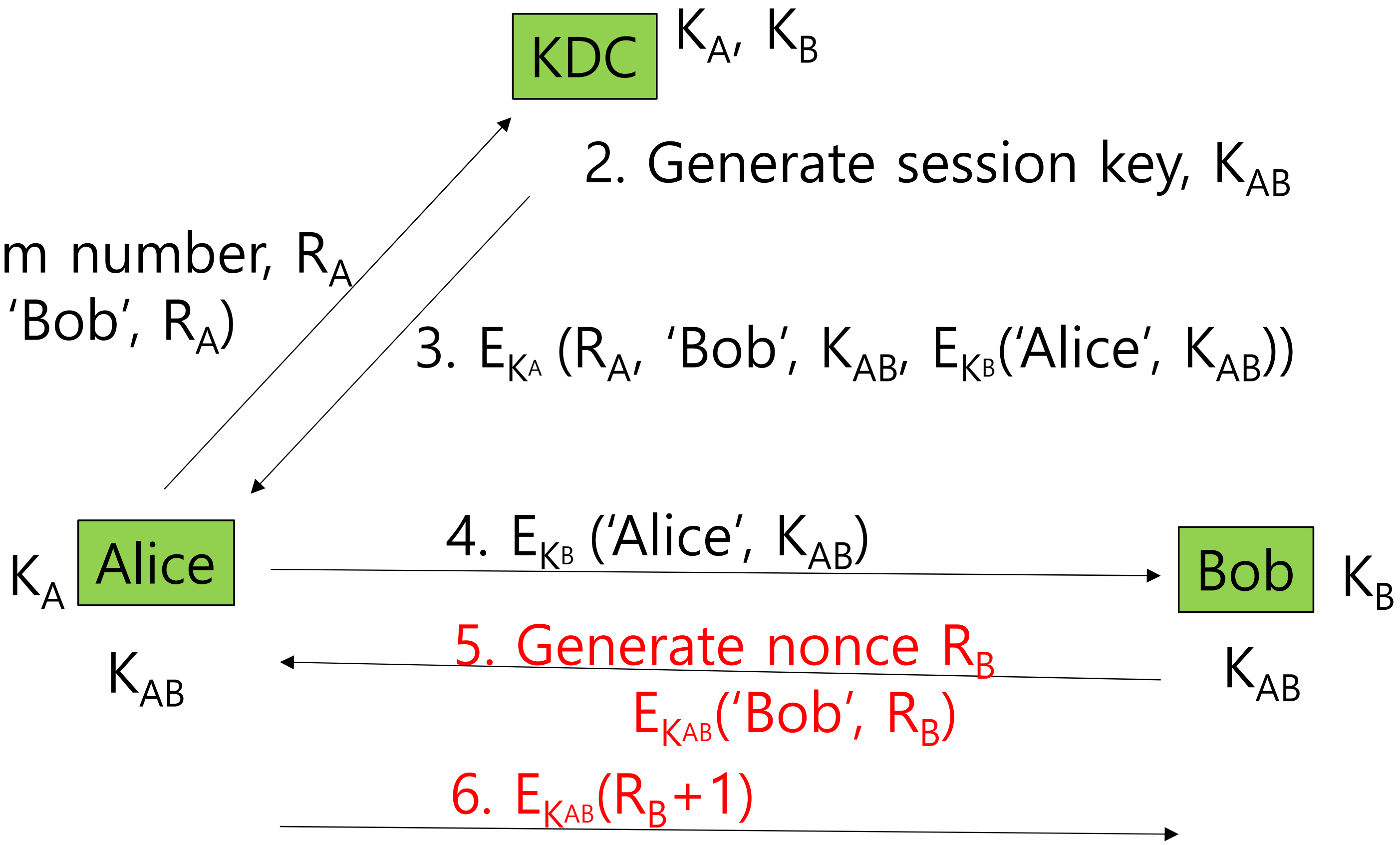
Bob K_B

K_{AB}

Key establishment + mutual authentication

- In the protocol of previous slide, nonce(one time random number) is used to prevent replay attack.
- What about PFS?
- When Bob receives the message, he can be assured the other party is really Alice if he trusts KDC.
- But Bob doesn’t authenticate himself to Alice.
- How can they mutually authenticate themselves?
  - Challenge-response scheme can be used for this purpose.

+ mutual authentication



KDC K_A, K_B

2. Generate session key, K_{AB}

1. Generate random number, R_A
   Request('Alice', 'Bob', R_A)

3. E_{K_A} (R_A, 'Bob', K_{AB}, E_{K_B}('Alice', K_{AB}))

K_A Alice

K_{AB}

4. E_{K_B} ('Alice', K_{AB})

Bob K_B

K_{AB}

5. Generate nonce R_B
   $E_{K_{AB}}$('Bob', R_B)

6. $E_{K_{AB}}$(R_B+1)

Remarks:

- Session key, K_{AB}, can make them authenticate themselves to the other party.
- Nonce R_B is used for preventing replay attack.
- Why $E_{K_{AB}}(R_B+1)$?
  - Someone can reuse $E_{K_{AB}}(R_B)$.
- Timestamp often replaces nonce.
  - But when using timestamp, the clocks at both users must be synchronized within permissible time difference.
- Kerberos is slightly complex version of this protocol.

Kerberos KDC

- Kerberos **Key Distribution Center** or **KDC**
  - KDC acts as the TTP(Trusted Third Party)
  - TTP should be trusted, so it must not be compromised
- KDC shares symmetric key K_A with Alice, key K_B with Bob, key K_C with Carol, etc.
- And a master key K_{KDC} known ***only*** to KDC
- KDC enables authentication as well as establish session keys
  - Session key for confidentiality and integrity

Kerberos Tickets

- KDC issue **tickets** containing info needed to access network resources
- KDC also issues **Ticket-Granting Tickets** (**TGTs)** that are used to obtain tickets
- Each TGT contains
  - Session key
  - User's ID
  - Expiration time
- Every TGT is encrypted with K_{KDC}
  - So, TGT can only be read by the KDC

Kerberized Login

- Alice enters her password
- Then Alice's computer does following:
  - Derives K_A from Alice's password
  - Uses K_A to get TGT for Alice from KDC
- Alice then uses her TGT (credentials) to securely access network resources
- **Plus:** Security is transparent to Alice

Kerberized Login

Alice

password

Computer

request TGT

$E_{K_A}(S_A,\text{TGT})$

KDC

- Key K_A = h(Alice's password)
- KDC generates a session key S_A
- Alice's computer decrypts S_A and TGT
  - Then it forgets K_A
- TGT = $E_{K_{KDC}}$("Alice", S_A)

Alice Requests “Ticket to Bob”

Alice

I want to talk to Bob

Computer

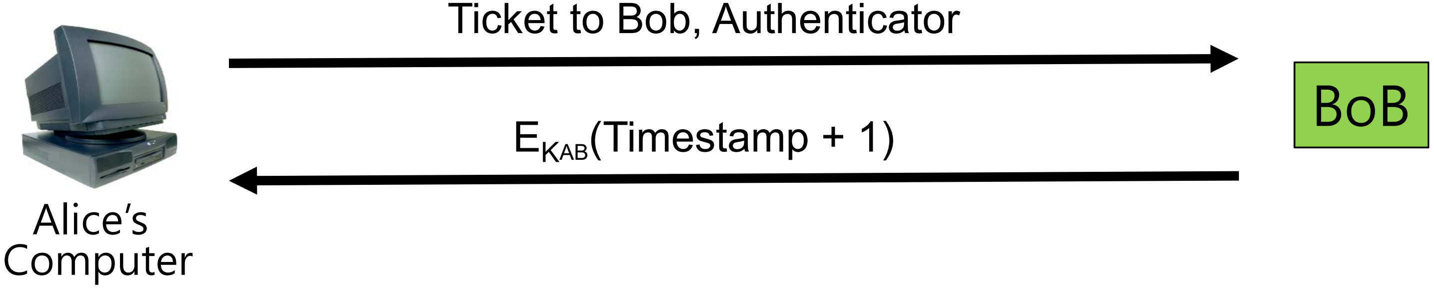
REQUEST

REPLY

KDC

- REQUEST = (TGT, Authenticator)
  - authenticator = E_{SA}(Timestamp)
- REPLY = E_{SA} (“Bob”, K_{AB}, Ticket to Bob)
  - Ticket to Bob = E_{K_B}(“Alice”, K_{AB})
- KDC gets S_A from TGT to verify timestamp

Alice Uses Ticket to Bob



Ticket to Bob, Authenticator

$E_{K_{AB}}$(Timestamp + 1)

BoB

Alice's Computer

- Ticket to Bob = E_{K_B}(“Alice”, K_{AB})
- Authenticator = $E_{K_{AB}}$(Timestamp)
- Bob decrypts “Ticket to Bob” to get K_{AB} which he then uses to verify timestamp

Remark:

- Key S_A used in authentication for Alice to KDC
- Timestamps for replay protection
  - Reduce the number of messages—like a nonce that is known in advance
  - But, "time" is a security-critical parameter
- Why does KDC use a TGT?
  - KDC doesn't need to remember any information about Alice and Bob.
  - **stateless** KDC is major feature of Kerberos

Key management

- In Kerberos, K_A = h(Alice's password)
- Could instead generate random K_A
  - Compute K_h = h(Alice's password)
  - And Alice's computer stores $E_{K_h}(K_A)$
- Then K_A need not be changed when Alice changes her password
  - But $E_{K_h}(K_A)$ must be stored on computer
- This alternative approach is often used
  - But not in Kerberos

Kerberos Questions

- When Alice logs in, KDC sends $E_{K_A}(S_A, TGT)$ where $TGT = E_{K_{KDC}}(“Alice”, S_A)$, why is TGT encrypted with K_A?
  - Extra work for no added security!
- In Alice’s “Kerberized” login to Bob, can Alice authenticate herself?
- Why is “ticket to Bob” sent to Alice?
  - Why doesn’t KDC send it directly to Bob?

Key Wrap Algorithm

Key wrap

- Even when a user encrypts message by using symmetric key algorithm, he has two keys; one is called key encryption key(KEK) which is used for encrypting the content encryption key(CEK) which is used for encrypting message.
  - And then send encrypted(key wrapped) CEK and encrypted message.(It is one possibility of Key Wrap application.)
  - In other application, we can store the key-wrapped CEK in a disk.
- Overall, the Key Wrap can be considered to be one method of key management.

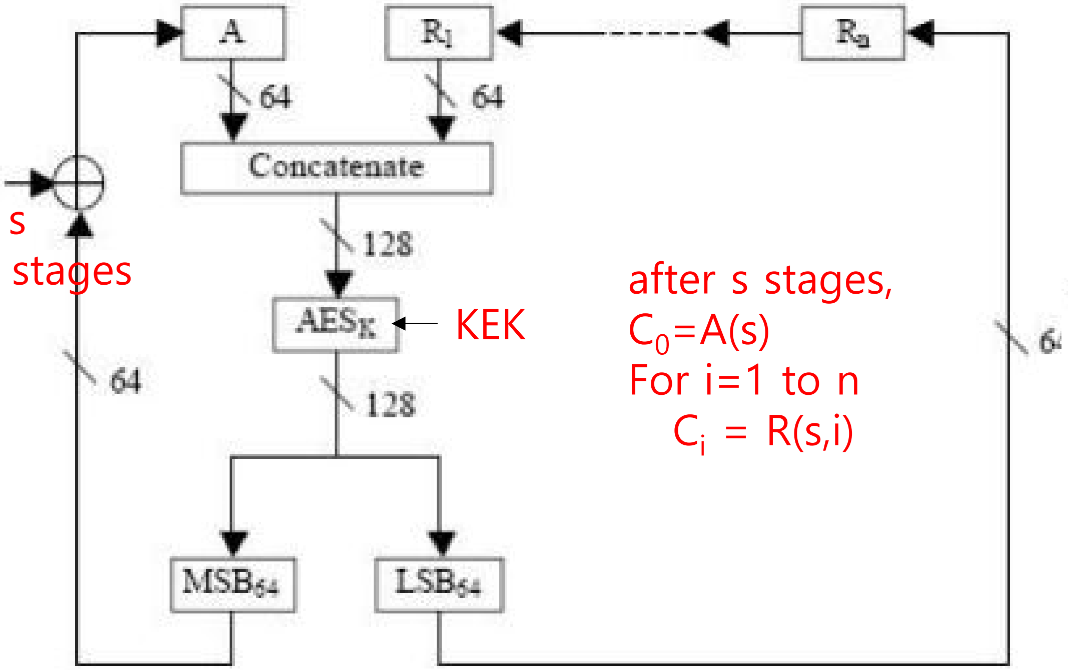
Types of Key Wrap mode of operation

- In actual implementation, KEK encrypts the CEK with other data, which is called a key data (or key material).
  - Key Wrap(CEK) = E_{KEK}(CEK + other data)
  - In this case, the length of key data is longer than the block length of KEK (128 bits for AES).
  - So, we apply a different mode of operation for key wrapping.
    - Key Wrap(CEK) = KW_{KEK} (CEK + other data)
- Types of Key Wrap algorithms
  - AESKW(AES key wrapping algorithm)
  - TDKW (TDES key wrapping algorithm)
    - Similar to AESKW except for using 3DES instead of AES
  - AKW1
  - AKW2

AESKW

Input: $(P_0,\ldots,P_n)$: key data including CEK
Output : ciphertext $(C_0,\ldots,C_n)$
Initialize:
  $A[0] = IV$
  for $i=1$ to n
    $R(0,i) = P_i$

Input: $(C_0,\ldots,C_n)$: key data including CEK
Output : ciphertext $(P_0,\ldots,P_n)$, IV
Initialize:
  $A[s] = C_0$
  for $i=1$ to n
    $R(s,i) = C_i$

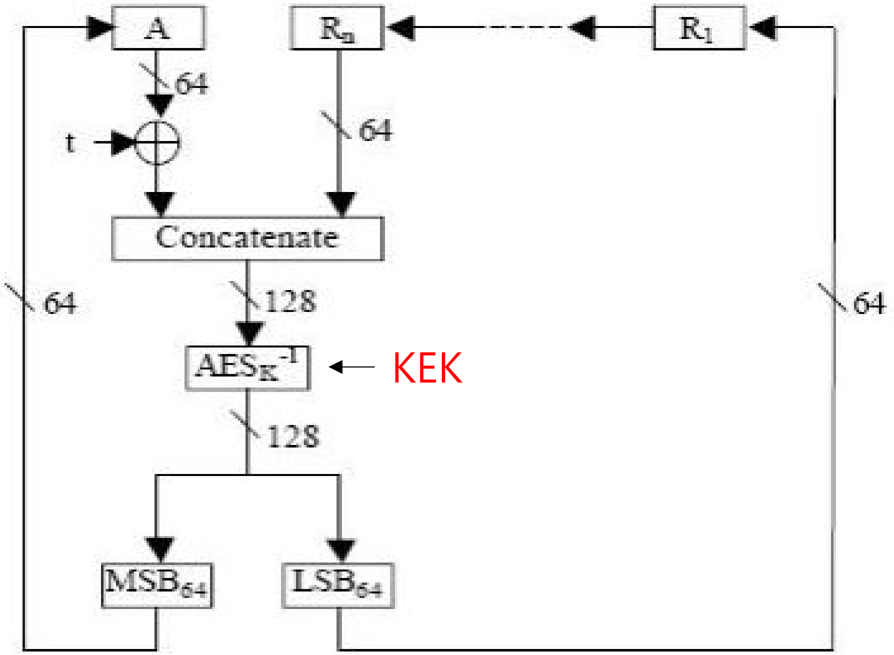


t=1 to s
S: # of stages

← KEK

after s stages,
$C_0=A(s)$
For $i=1$ to n
  $C_i = R(s,i)$

Kek Wrap



← KEK

Kek Unwrap

Why using KW mode of operation?

- Key material is longer than the block size of encryption algorithm. (ex, 128bits for AES)
  - If we use the block mode of operation, the first block influences only on the first block of ciphertext, subsequently the next blocks only influences on the next blocks of ciphertext.
- But the KW mode of operation make the data of blocks be interspersed in all blocks of ciphertext, making more security.

Simplified use of AESKW

Alice

Bob

KEK_{AB}

KEK_{AB}

generate CEK_{AB}
encrypt CEK_{AB} : $Ckey=KW_{KEK_{AB}}(CEK_{AB})$
Message: m
encrypt message: $c=E_{CEK_{AB}}(m)$

(Ckey, c)

decrypt Ckey : $CEK_{AB}=KW^{-1}_{KEK_{AB}}(Ckey)$
decrypt message: $m=D_{CEK_{AB}}(c)$

Purpose of key wrapping

- For more security?
  - In my opinion, there is no point of key wrapping for providing more security.
  - If KEK is revealed, so is the message.
- But there is one advantage:
  - Suppose Bob maintains encrypted data communicated up to now.
  - Even if KEK is revealed, he doesn’t need to change the CEK.
  - Instead, Alice re-encrypts the same CEK with new KEK and sends the newly encrypted CEK to Bob.

Random Number Generation (RNG)

Application of random numbers

- Random numbers used to generate **keys**
  - Symmetric keys
  - RSA: Prime numbers
  - Diffie Hellman: secret values
- Random numbers used for nonces
  - Sometimes a sequence is OK
  - But sometimes nonces must be random
- Random numbers also used in simulations, statistics, etc., where numbers need to be “statistically” random

Types of RNG: TRNG

- True RNG
  - Random numbers are generated from physical process in real life.
    - Eg, coin flipping, lottery, thermal noise, mouse movement, radioactive decay, lava lamp, etc.
- What is “random”?
  - In statistics, a sequence of numbers without any correlation or bias – “statistical randomness”
  - General definition: a sequence of numbers (events) has no regularity(order) and does not follow an intelligible pattern or combination, so unpredictable.

Types of RNG: PRNG

- Pseudo RNG (PRNG)
  - Random numbers are computed, i.e. they are deterministic.
  - Typical algorithm for computing PRNG
    - S_0=seed, $S_{i+1} = F(S_i)$
  - Eg, RAND() function in ANSI C
    - S_0=12345, $S_{i+1} = 1103515245\ S_i + 12345 \pmod{2^{31}}$

Types of RNG

- Cryptography PRNG (CPRNG)
  - CPRNGs are PRNG with one additional property; generated numbers are unpredictable.
  - Given n output bits

    $S_i, S_{i+1}, \ldots, S_{i+n-1}$

    it is computationally infeasible to generate S_n.
  - So, the number is generated by an artificial algorithm like PRNG. But, the generated number is not deterministic (predictable).

CPRNG based on Hash Function

- Defined by NIST SP 800-90 and ISO 18031
- The algorithm uses the crypto hash function H, generating n-bits random number.

```
data = IV (seed)
W = null string
Seedleng : bit length of IV
For i = 1 to m (n < m x the length of hash value)
    $w_i$ = H(data)
    W = W || $w_i$
    data = (data +1) mod $2^{seedleng}$
Return leftmost n bits of W
```

Example of bad random number use

- Online version of Texas Hold 'em Poker developed by ASF Software, Inc.
- Random numbers used to shuffle the deck.
- Program did not produce a random shuffle. Did it cause a serious problem or not?

(source: Information Security, Mark Stamp)

How many instances of card shuffle?

- There are 52! > 2^{225} possible shuffles
- The poker program used "random" 32-bit integer to determine the shuffle
  - Only 2^{32} distinct shuffles could occur
- Code used Pascal pseudo-random number generator (PRNG): Randomize()
- Seed value for PRNG was function of number of milliseconds since midnight
- Less than 2^{27} milliseconds in a day
  - So, less than 2^{27} possible shuffles

- PRNG re-seeded with each shuffle
- By synchronizing clock with server, number of shuffles that need to be tested $< 2^{18}$
- Could then test all 2^{18} in real time
  - Test each possible shuffle against “up” cards
- Attacker knows **every card** after the first of five rounds of betting!