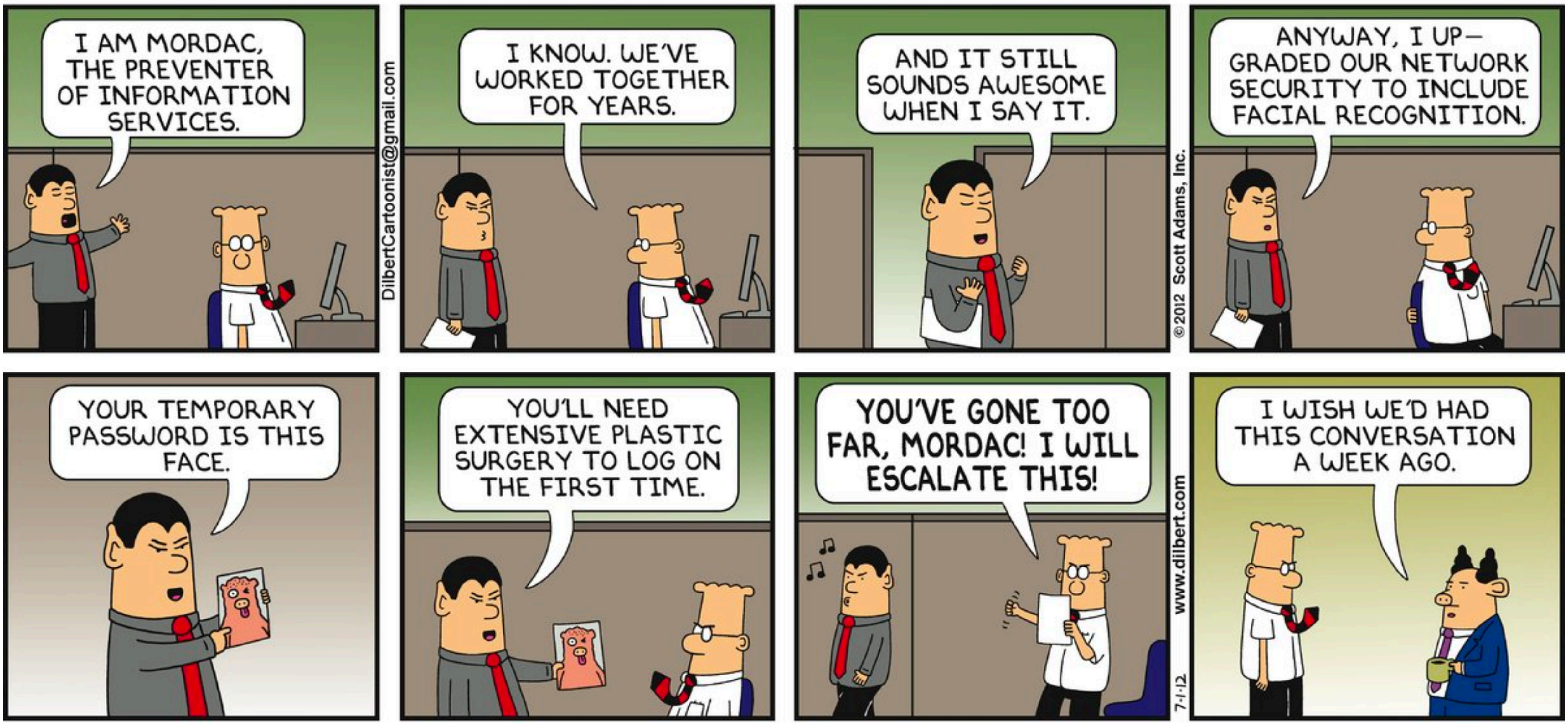
Key Distribution

February 15, 2022



Homework 1 and Today

- Homework 1
    - available on **Blackboard**
    - based on cryptography lectures, **requires Python or Java programming**
    - due **February 20th** (Sunday) at 11:59pm

- Today:
    - digital signatures
    - **key distribution**
      Where do keys come from?
      Where do they go?
      How do they get there?

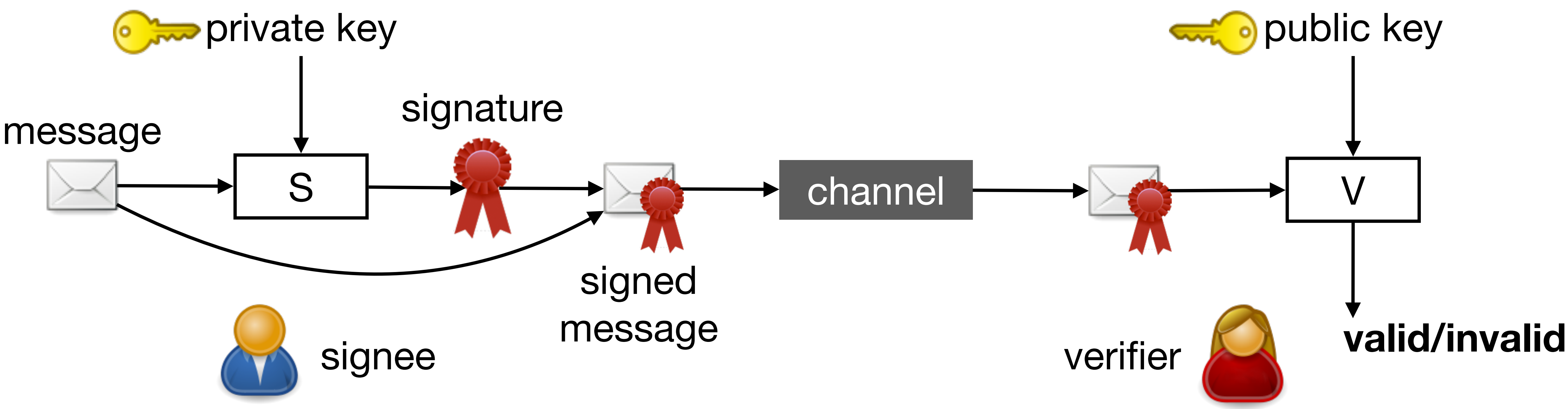
Feedback: https://forms.gle/JGbNCmCsU69iWaTv8

Digital Signatures

Motivation for Digital Signatures

- Message authentication does not protect the sender and receiver from each other
  - receiver can forge a message and claim that it is from the sender
  - sender can deny sending a message and claim that it was forged by the receiver
- Non-repudiation:
  sender cannot deny that it has sent a message
- Digital signature
  ≈ message authentication + non-repudiation
  - provide integrity and authenticity protection as well as non-repudiation
  - similar to traditional signatures: signee cannot deny signing a document
  - in many countries, digital signatures have legal significance

Digital Signature



- Signee knows the private key → can sign
- Verifier knows the public key → can verify
  - public key can be published so that anyone can verify
- Attacker (i.e., forger) does not know the private key → cannot sign

Digital Signature Schemes

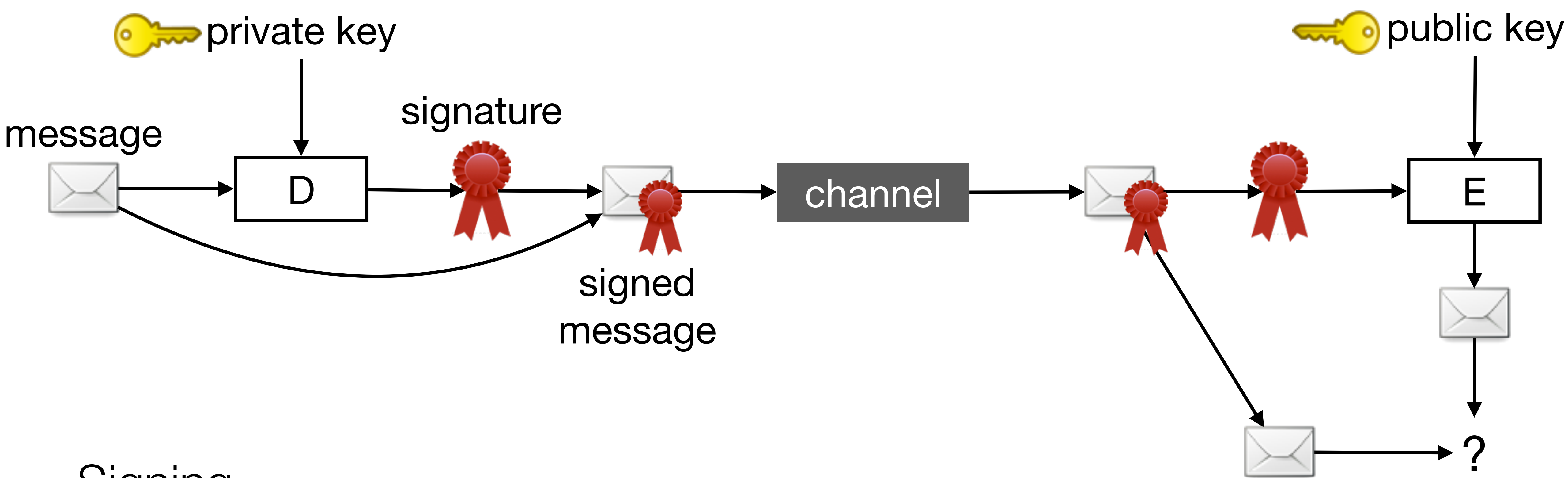
Algorithms:

- Key generation **G()**: randomized algorithm, outputs key pair (PU, PR)
- Signature **Sign(PR, M)**: takes private key PR and message M, outputs signature S
- Verification **Verify(PU, M, S)**: takes public key PU, message M, and signature S, outputs accept/reject

Public-key encryption:

- Key generation **G()**: randomized algorithm, outputs key pair (PU, PR)
- Decryption **D(PR, C)**: takes private key PR and ciphertext C, outputs plaintext M
- Encryption **E(PU, M)**: takes public key PU and plaintext M, outputs ciphertext C

Digital Signatures Using Public-Key Encryption

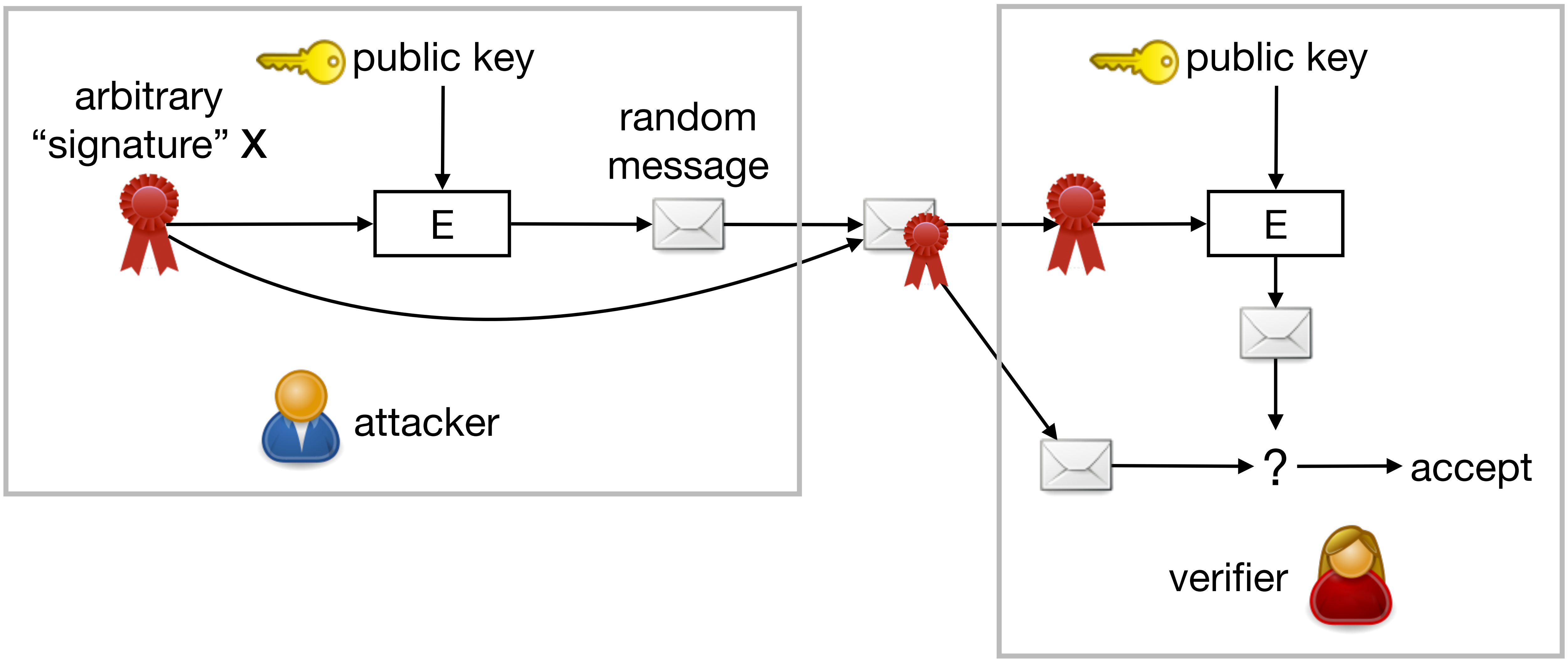


- Signing
  - treat the message as a ciphertext, “decrypt” it using the private key
  - signature is the resulting “plaintext”
- Verification
  - treat the signature as a plaintext, encrypt it using the public key
  - verify if the resulting “ciphertext” is equal to the message

Simple Forgery Attack

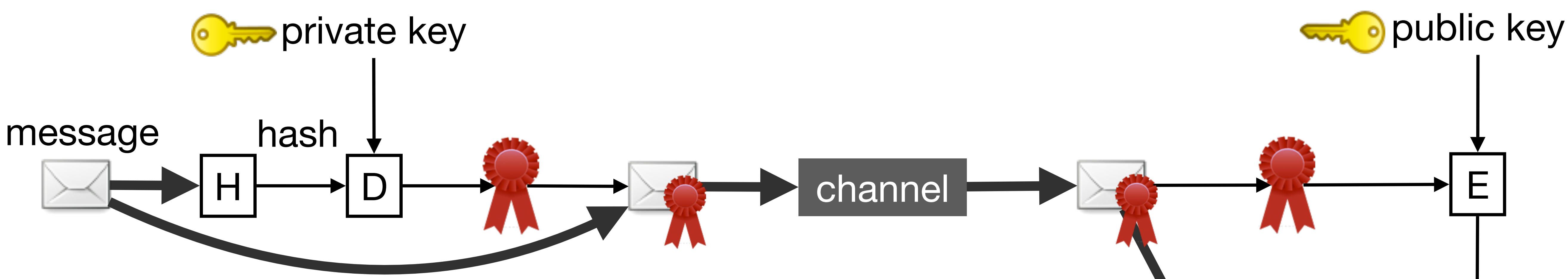
- Attacker can forge a signature for a random message
  - pick an arbitrary value **X**, and use it as a signature

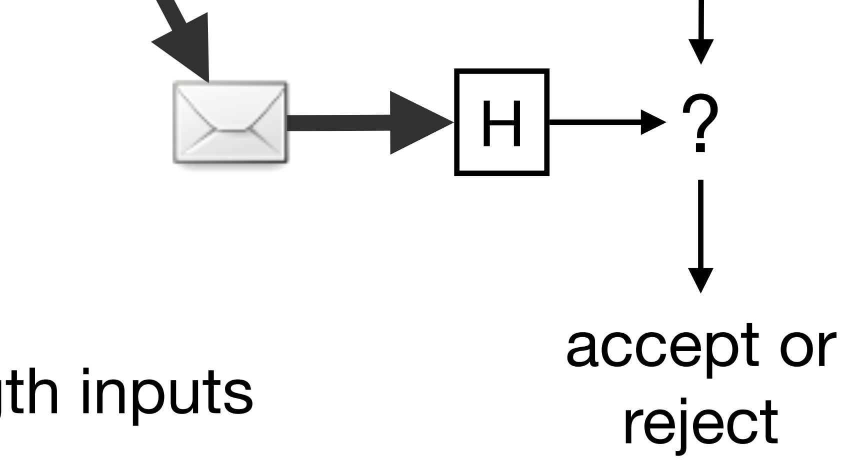
    → signature for message **E(PU, X)** is **X**



Hash-then-Sign

- *Idea*: sign a cryptographic hash of the message





- Advantages
  - **compatibility**:
    most public-key encryption algorithms take fixed-length inputs
  - **efficiency**: signature will be shorter and faster to compute
  - **security**: prevents existential forgery (attacker cannot compute forged message for an arbitrary signature using only the public-key)

Cryptographic Hash Function

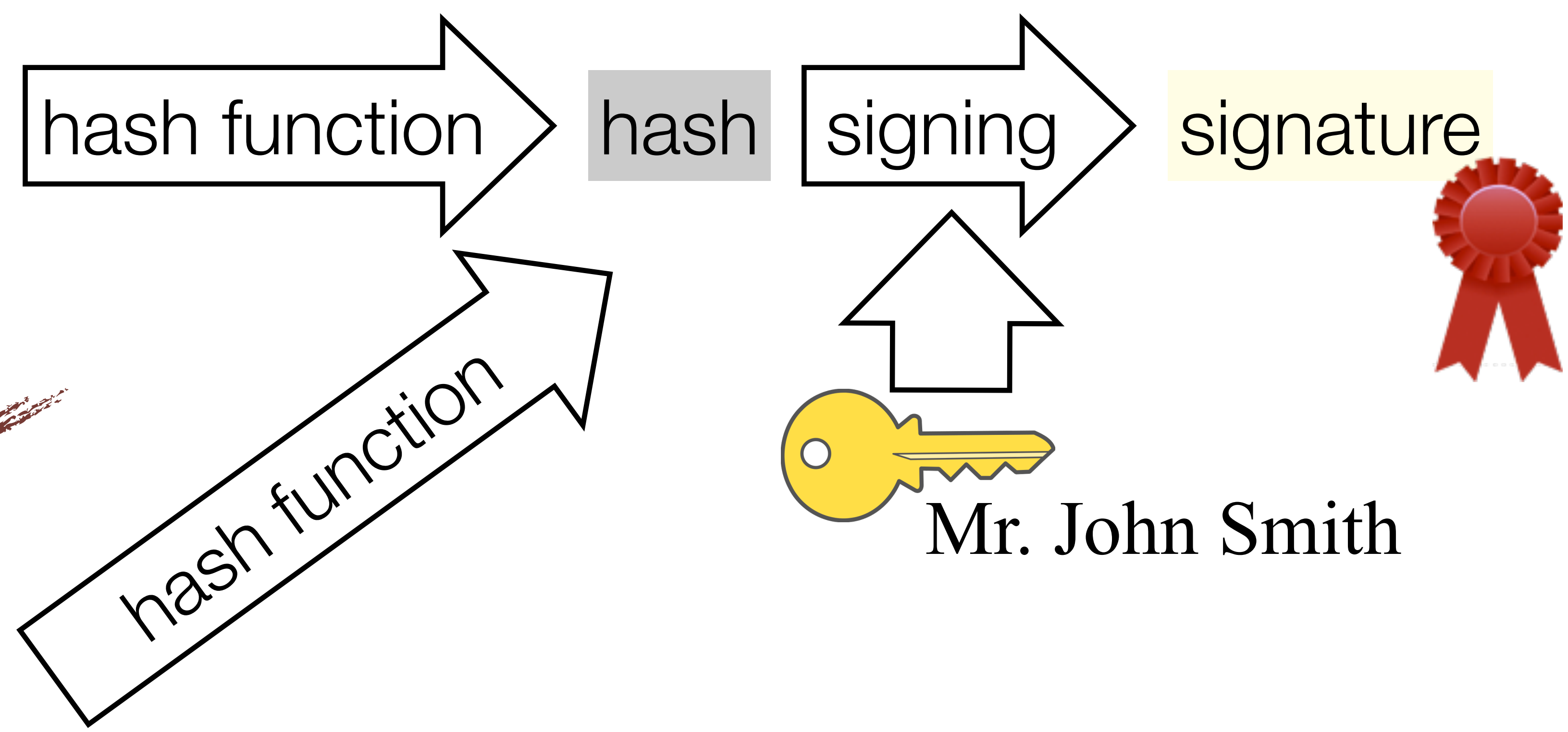
- **One-way** → prevents existential forgery with public-key encryption
- **Collision-resistant**

Contract

Mr. John Smith agrees to sell his yacht to Mr. John Doe in exchange for $2,000,000.

~~**Contract**~~

~~Mr. John Smith agrees to sell his yacht to Mr. John Doe in exchange for $2,000.~~



RSA Signatures

- Very widely used with SHA-256 (and other versions of SHA)
  - *example:* SSL/TLS
- Standard: **PKCS #1** by RSA Laboratories, republished as RFC 3447
  - RSASSA-PKCS1-v1_5
    - older standard
  - RSASSA-PSS
    - PSS = Probabilistic Signature Scheme:
      adds randomized padding (called salt) to the message
    - provably secure (given that RSA is secure)

Digital Signature Algorithm (DSA)

- Digital Signature Standard:
  FIPS (Federal Information Processing Standard) 186
    - introduced in 1993, updated multiple times
    - latest version includes RSA, DSA, and elliptic-curve signatures
- Digital Signature Algorithm
    - proposed by NIST in 1991
    - designed for signature, cannot be used for encryption
    - efficient variant of the ElGamal signature scheme (much smaller signatures, modular arithmetic operations with lower moduli)
- Elliptic Curve Digital Signature Algorithm (ECDSA)
    - based on elliptic curve cryptography
    - shorter keys and increased efficiency

Digital Signatures Conclusion

- Digital signature
  ≈ message authentication + **non-repudiation**
  - provides integrity and authenticity protection as well as non-repudiation
- Based on asymmetric-key cryptography
  → much **slower** than message authentication
- Algorithms
  - RSA
  - DSA
  - ECDSA

Summary of Cryptographic Primitives

Types of Cryptographic Primitives

	Symmetric-key	Asymmetric-key	
Confidentiality	Block ciphers Stream ciphers	Asymmetric-key encryption	
Integrity	Message authentication	Digital signatures	Hash functions

Cryptographic Primitives
Lessons Learned

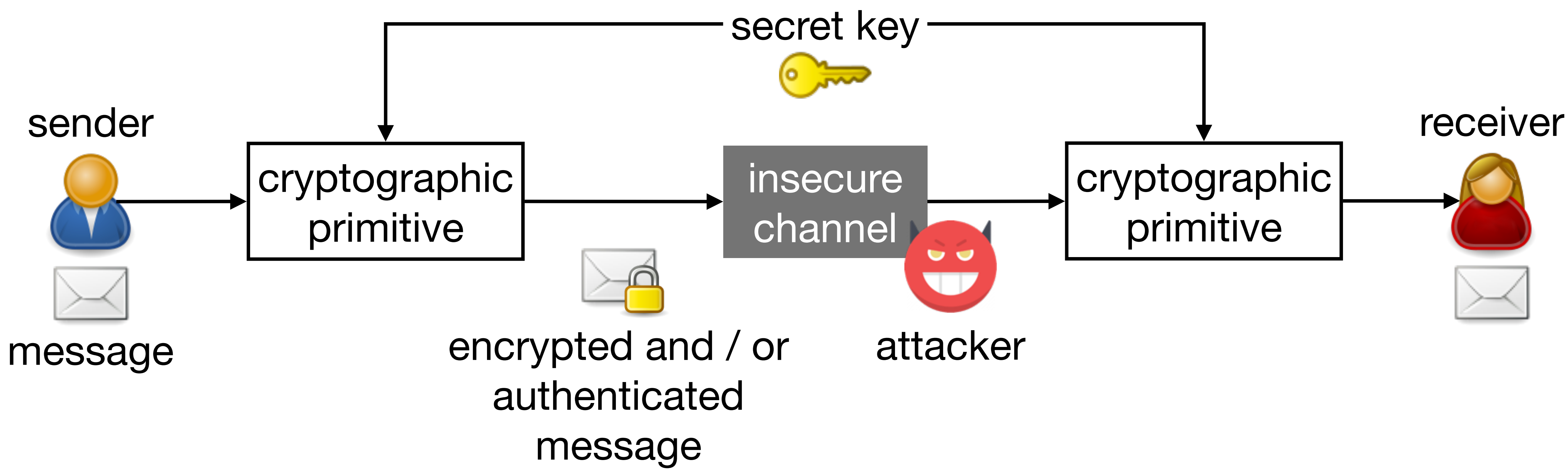
- Obscurity is not security
  - *example*: A5/1 cipher (GSM) was designed in secret, but was eventually broken
- Security of practical cryptographic primitives is not proven
  - symmetric primitives are built on design principles, asymmetric primitives are built on mathematical problems that are believed to be hard
- Nonetheless, widely-used cryptographic primitives are rarely broken
  - cryptographic primitives are much more trustworthy than software, users, etc.
- However, even secure primitives may be used, implemented, or combined in insecure ways
  - *example*: earlier versions of the SSL/TLS protocol had some weaknesses and very vulnerable implementations
- Security is a process not a product
  - key lengths and algorithms must be upgraded from time to time

Key Distribution

How can parties exchange or agree on a secret key?

Key Distribution

- Symmetric-key cryptography
  - much **more efficient** than asymmetric-key cryptography



- However, to use symmetric-key cryptography
  - communication parties must **share the same key**
  - unauthorized parties **must not know the key**

Key Freshness

- Secret keys may **become insecure when used for a long time**
  - more ciphertexts encrypted using the same key
    → easier for the attacker to recover the key
  - *examples*:
    - most stream ciphers produce pseudorandom sequences that repeat eventually
    - block ciphers with 64-bit blocks in CBC mode are likely to output the same block after ~34 GB of data → reveals XOR of corresponding plaintext blocks
- **Key freshness requirement**: renew (i.e., change) secret key frequently
  - *example*:
    SSH protocol usually requires a new key after 1 hour or 2^{32} packets (rekeying)
- Problem:

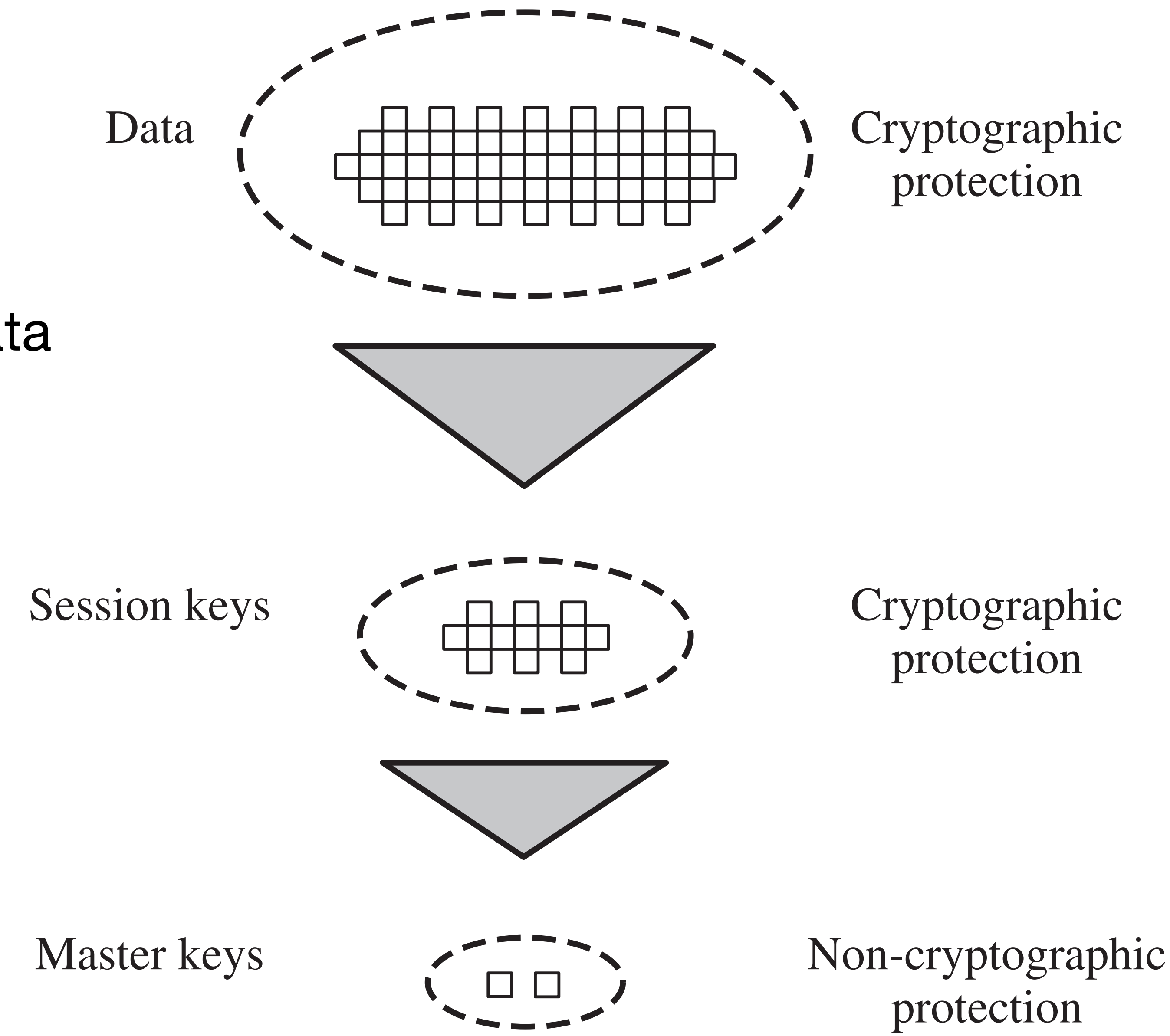
secret keys have to be renewed frequently ↔ setting up a secret-key is a complex operation

Secret-Key Hierarchy

- Session key
  - renewed frequently (*e.g.*, one key for each logical connection)
  - used to encrypt and authenticate data
- Master key
  - renewed infrequently
  - used to distribute session keys

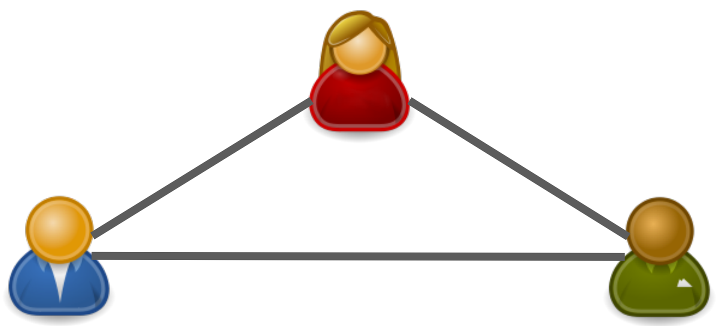
<u>Questions:</u>

- *What are the master keys (e.g., symmetric or asymmetric key)?*
- *Who have the master keys?*
- *How to obtain a session key from a master key?*

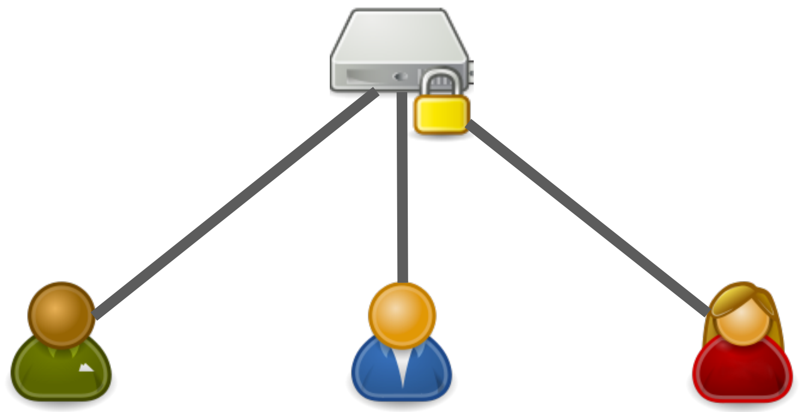


Secret-Key Distribution Approaches

1. Decentralized
    - each pair of communication parties share a secret master key



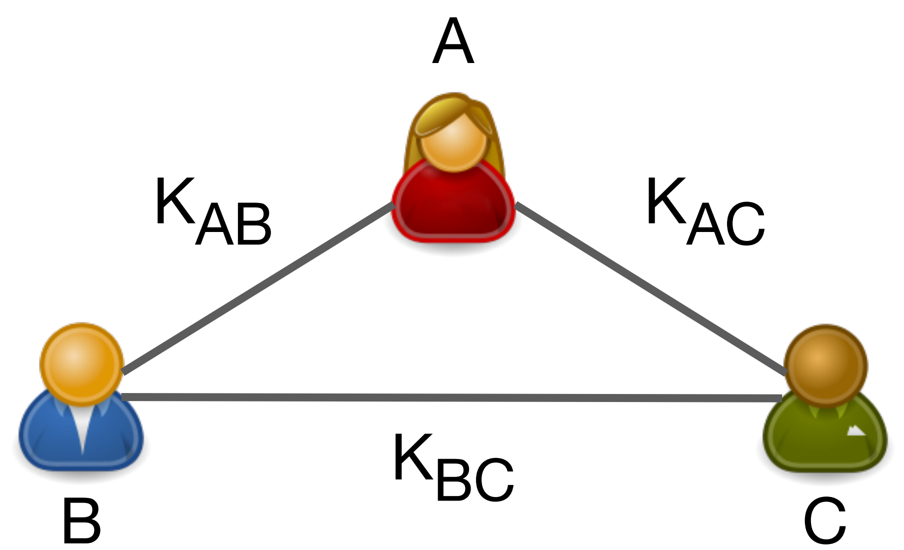
2. Key Distribution Center (KDC)
    - KDC shares a secret master key with each of the communication parties



3. Public-key cryptography
    - one communication party needs to have the public key of the other

Decentralized Key Distribution

- Each pair of communication parties has to share a secret master key

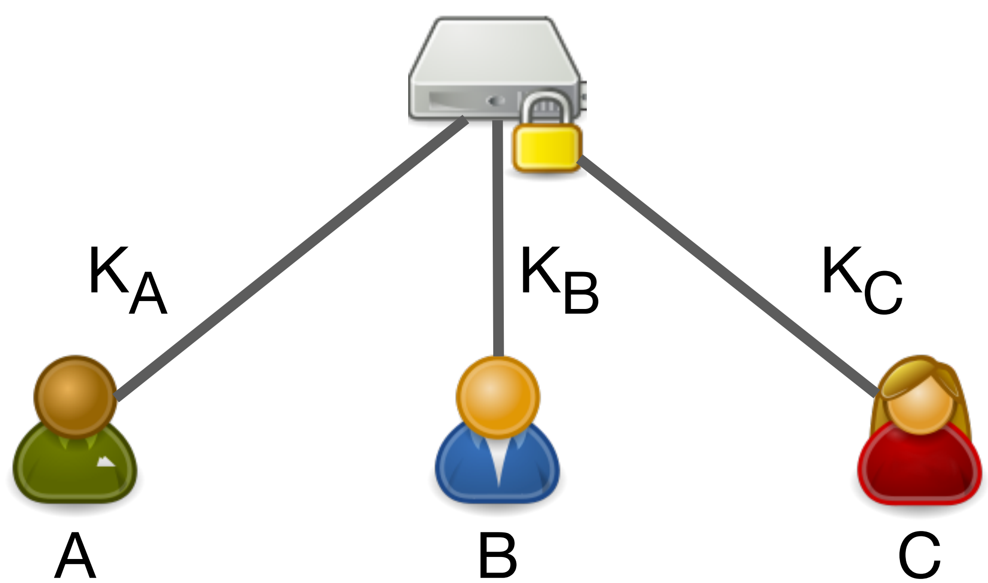


N communication parties
→ N · (N - 1) / 2 pairs

- Master key needs to be set up for each pair manually
  - any pair can then exchange or agree on session keys easily
- May work for securing small, local networks
  - *example*: physically delivering the key for each pair
- However, it does not scale well
  - especially difficult in a wide-area distributed system

Key Distribution Based on KDC

- Key Distribution Center (KDC)
  - acts as a **trusted third party**:
    all communication parties trust the KDC
  - each party **X** shares a secret master key $\mathbf{K_X}$ with the KDC



N communication parties
→ only **N** master keys

Key-Distribution Protocols

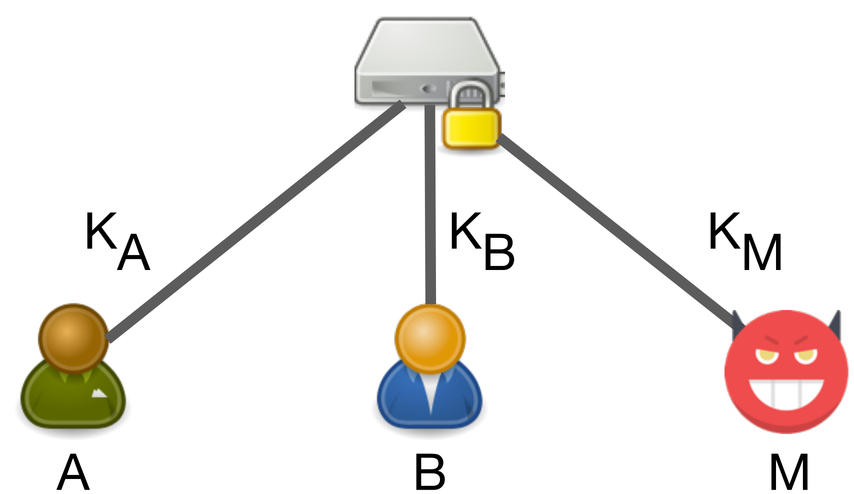
How to obtain a session key from master keys?

Assumptions and Adversary Model

- Cryptographic primitives are secure
- Each master key is known only by the KDC and the corresponding communication party
- Every non-malicious participant follows the protocol

Adversary

- may be a legitimate protocol participant (i.e., insider)
- has full control over the communication channels
- may have old, compromised session keys

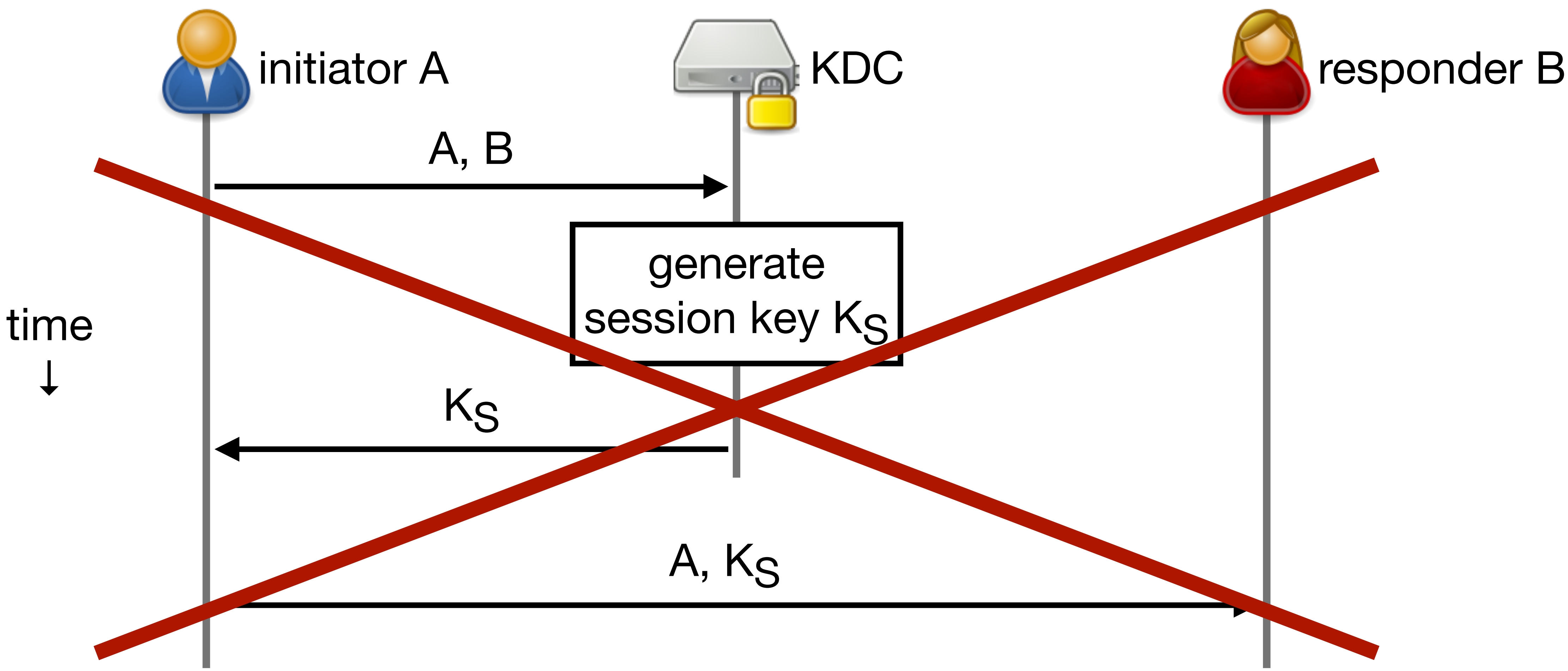


Key Distribution Objectives

- **Effectiveness**: both parties should learn the session key
- **Implicit key authentication**: no other parties (except for the trusted third party) should know the key
- **Key freshness**: both parties should be able to verify that the key was freshly generated

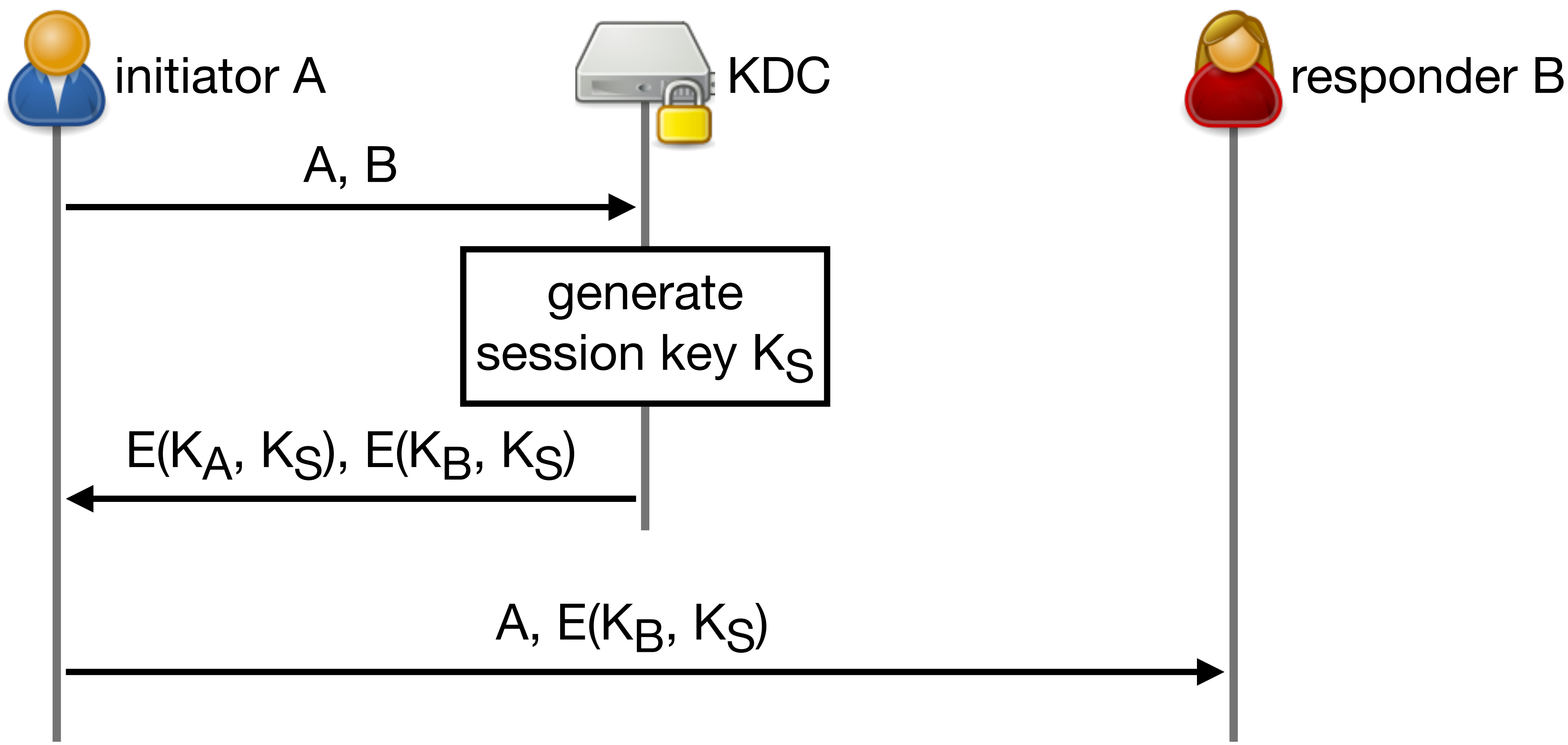
(**Key confirmation**: both parties should be able to verify that the other party also has the key)

Basic Key Transport



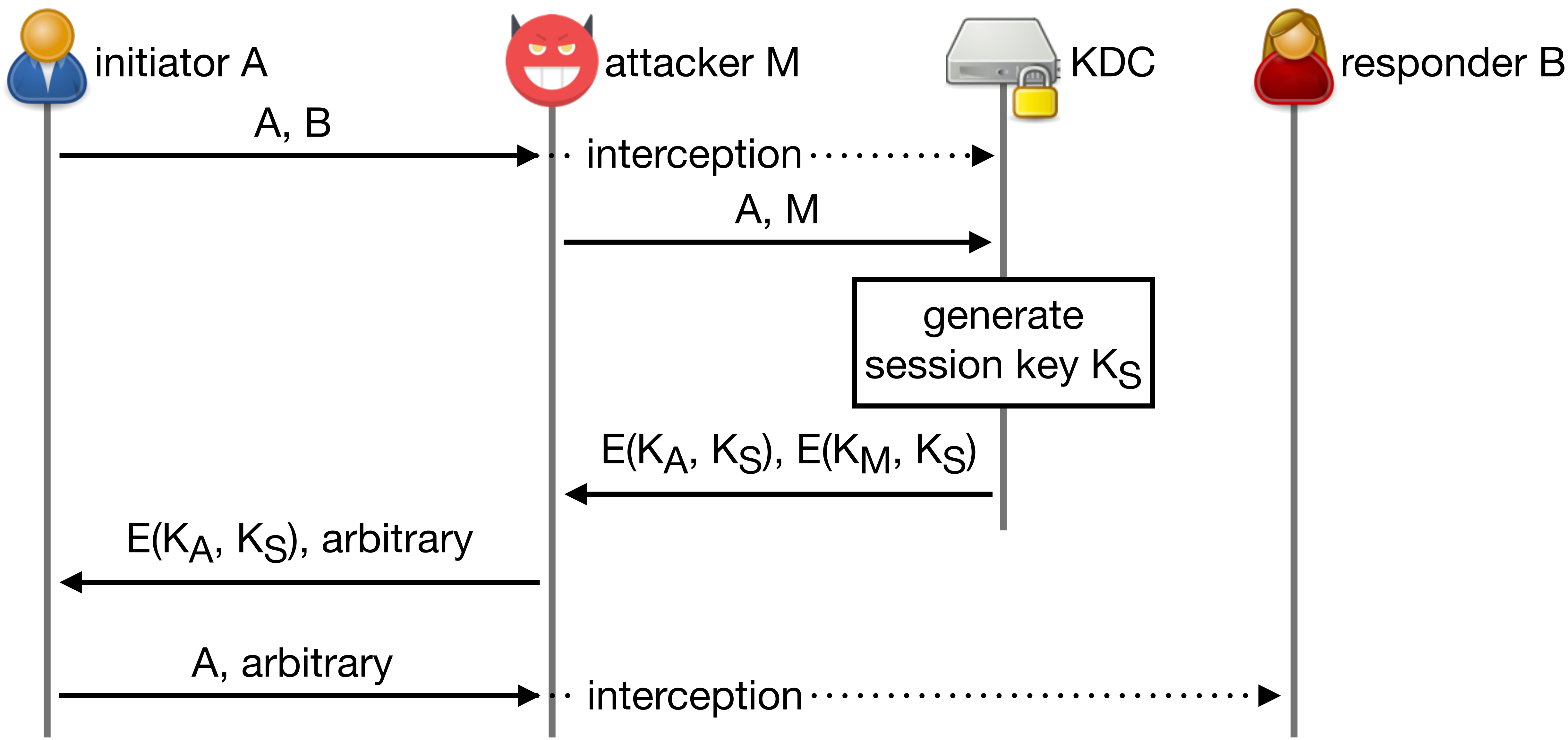
- Attacker can eavesdrop the session key K_S

Basic Key Transport with Encryption



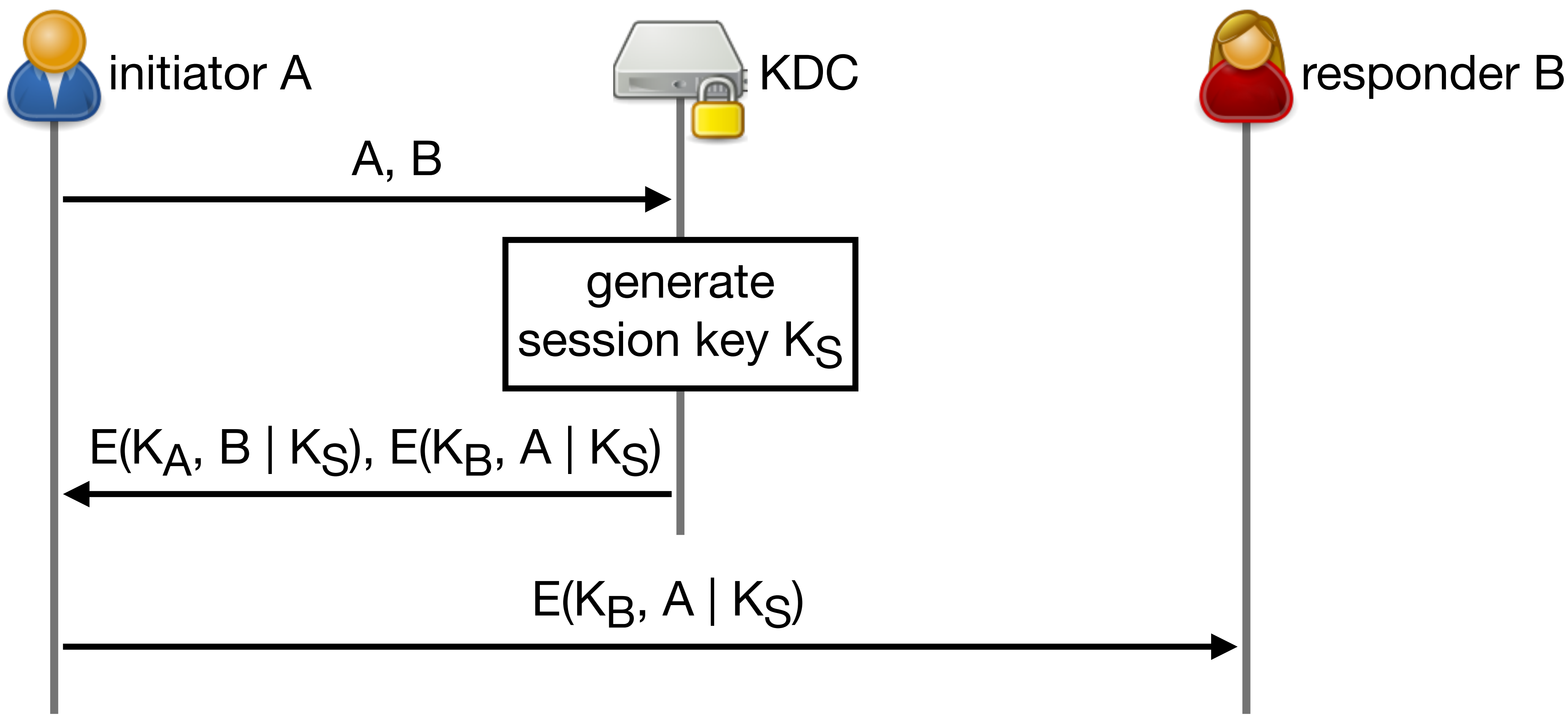
- Attacker cannot eavesdrop the session key K_S
- However, a man-in-the-middle attacker can impersonate B

Man-in-the-Middle Attack



- A thinks that it shares a secret key with B, but it actually shares a key with the attacker M

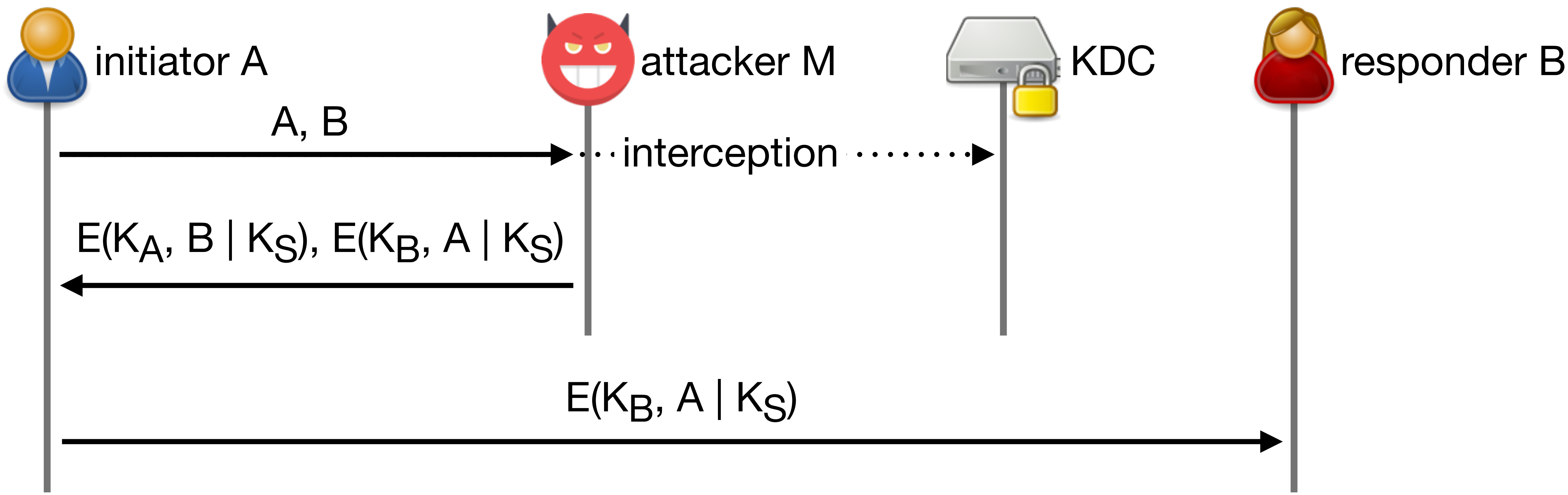
Basic Key Transport with Encryption and Identifiers



- Attacker cannot impersonate protocol participants
- However, a man-in-the-middle attacker may replay old session keys

Replaying Old Session Key

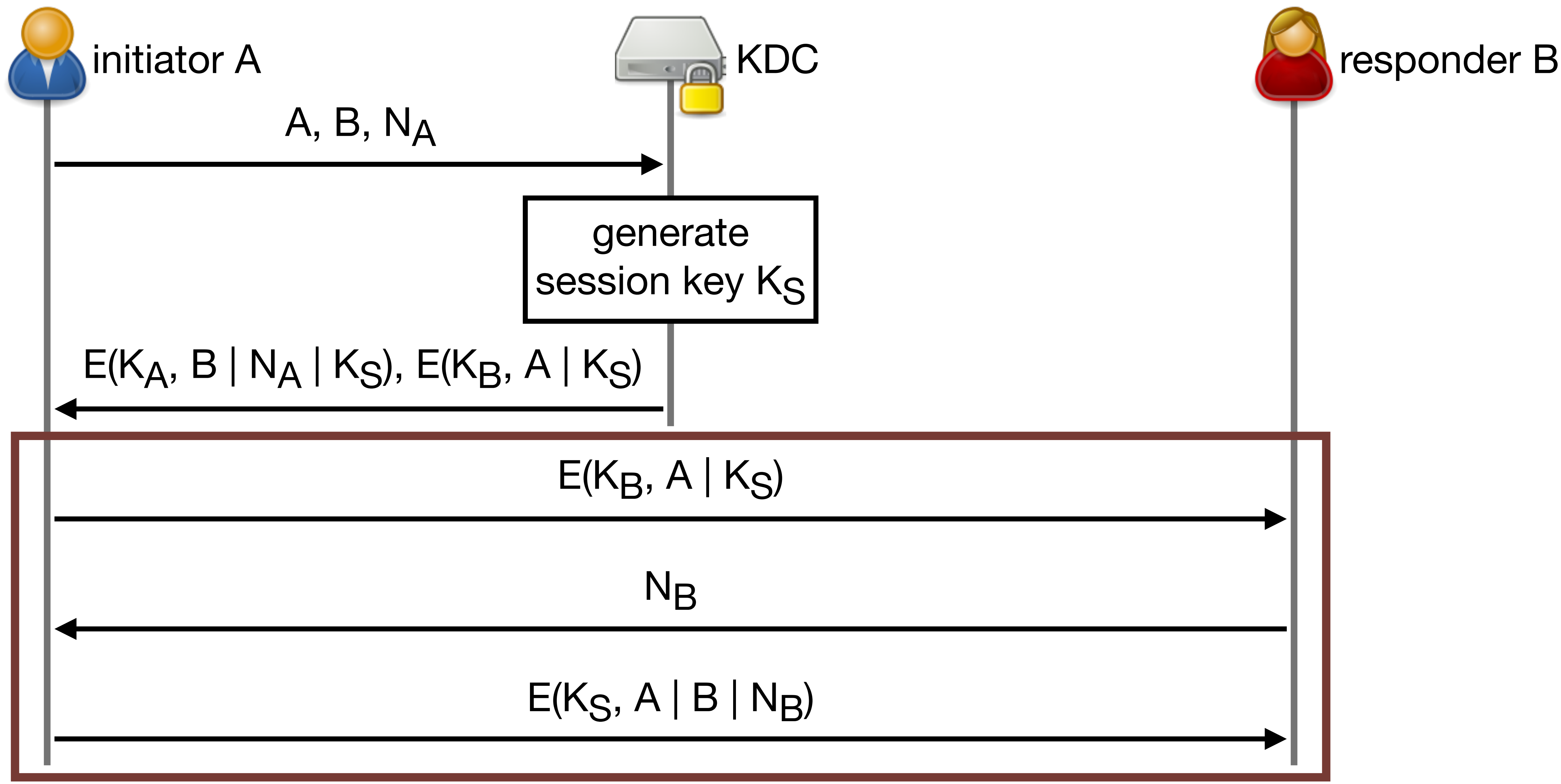
- Suppose that the attacker has observed the distribution of an old session key K_S



- Key freshness is not guaranteed by the protocol
  - neither **A** nor **B** can tell if the session key K_S was generated recently
- Attacker can force A and B to use the old key indefinitely

Key Transport with Identifiers and Nonces

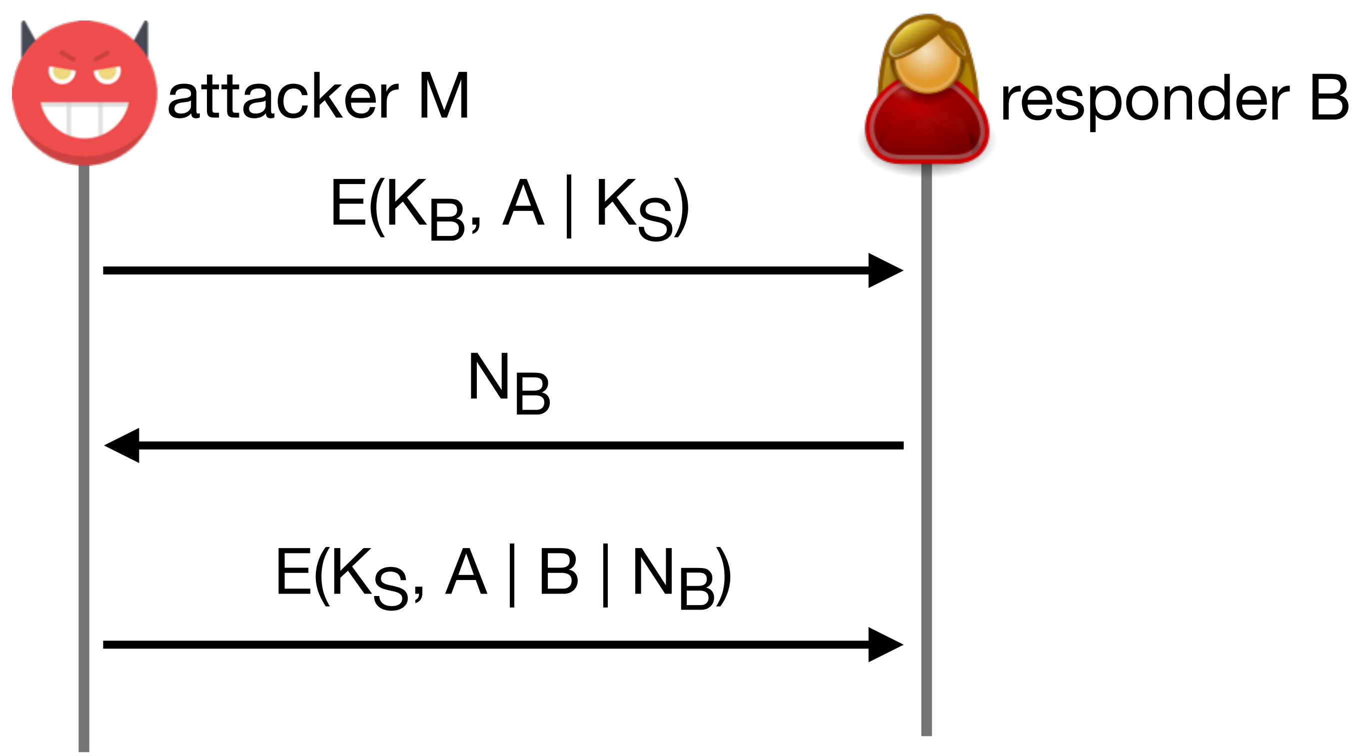
- **Nonce:** number used once



- Replaying an old, compromised session key is possible

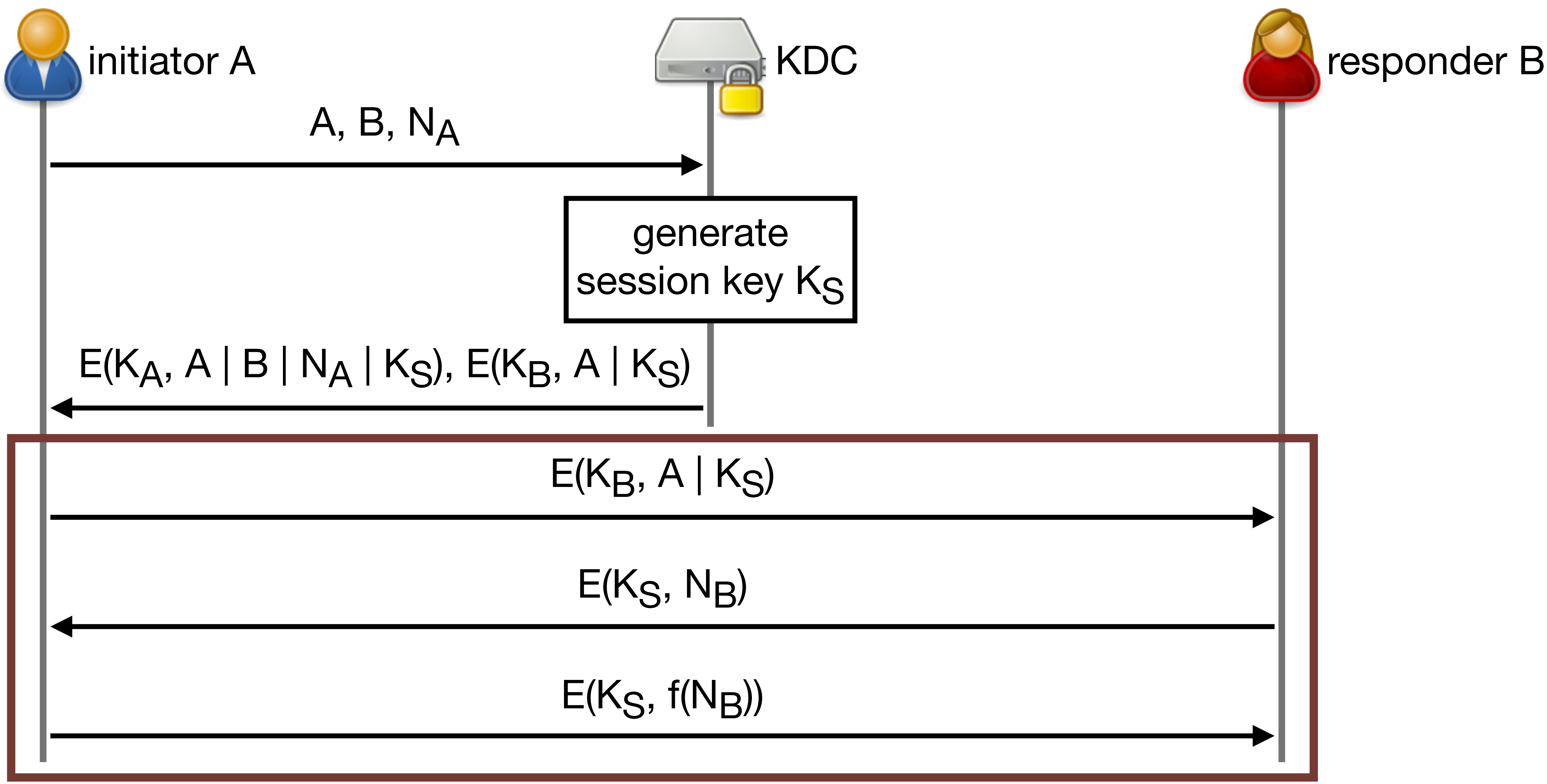
Replaying an Old Session Key

- Suppose that the attacker has compromised an old session key K_S



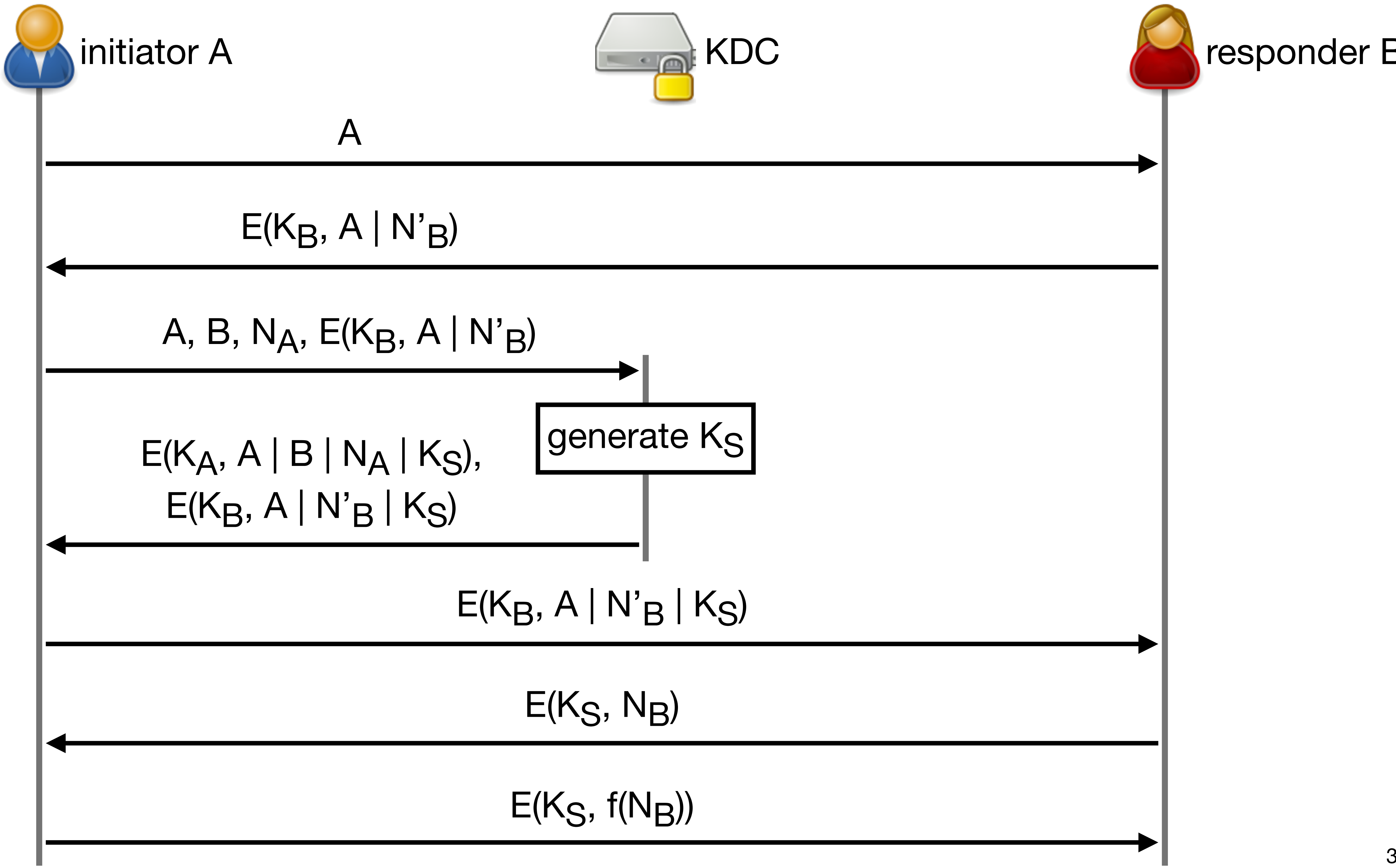
- Key freshness is still not guaranteed by the protocol

Needham-Schroeder Symmetric-Key Protocol



- f: some mathematical function (*e.g.*, subtracting one)
- Replaying an old, compromised session key is still possible

Extended Needham-Schroeder Protocol



Kerberos Network Authentication Protocol

- Allows nodes to communicate over a non-secure network and to prove their identities to each other
- Similar to the extended Needham-Schroeder protocol, but uses **timestamps** instead of nonces
  - in addition to timestamps, messages may also contain lifetimes
    → can limit usage time
- Windows 2000 and later versions use Kerberos as the default authentication for clients that want to join a Windows domain

The mythological Kerberos

Next lecture:

Public-Key Distribution and Certificates