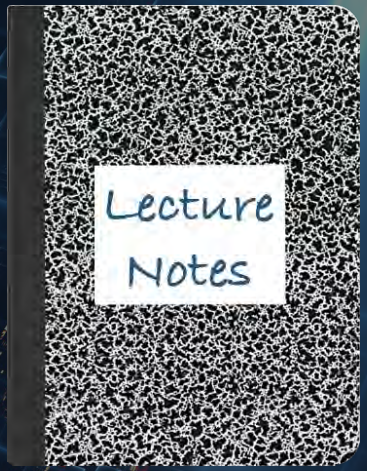
CS 417 – DISTRIBUTED SYSTEMS




# Week 12: Security in Distributed Systems

## Part 2: Data Integrity

Paul Krzyzanowski

© 2022 Paul Krzyzanowski. No part of this content may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

# Integrity: Goals

Use cryptographic techniques to detect that data has not been modified

Integrity mechanisms can help to

- Detect data corruption
- Detect malicious data modification
- Prove ownership of data

# Message Integrity

## How do we detect that a message has been tampered?

- A cryptographic hash acts as a checksum

*A hash is a small, fixed amount of information that lets us have confidence that the data used to create the hash was not modified*

- We associate a hash with a message
  - we're not encrypting the message
  - we're concerned with *integrity*, not *confidentiality*


- If two messages hash to different values, we know the messages are different

$$H(M) \neq H(M')$$

# Cryptographic hash functions

Also called *digests* or *fingerprints*

## Properties

- Arbitrary length input → **fixed-length output**
- **Deterministic**: you always get the same hash for the same message
- **One-way function** (**pre-image resistance**, **or *hiding*** )
  - Given *H*, it should be difficult to find ***M*** such that ***H=hash(M)***
- **Collision resistant**
  - Infeasible to find any two different strings that hash to the same value:
    Find ***M, M'*** such that ***hash(M) = hash(M')***
- **Output should not give any information about any of the input**
  - Like cryptographic algorithms, relies on ***diffusion***
- **Efficient**
  - Computing a hash function should be computationally efficient

# Hash functions are the basis of integrity

- Not encryption
- Can help us to detect:
  - Masquerading

    Insertion of message from a fraudulent source
  - Content modification

    Changing the content of a message
  - Sequence modification

    Inserting, deleting, or rearranging parts of a message
  - Replay attacks

    Replaying valid sessions

# Some Popular Hash Functions

**MD5**
- 128 bits
- Linux passwords used to use this
- **Rarely used** now since weaknesses were found

**SHA-1**
- 160 bits – was widely used: checksum in Git & torrents
- Google demonstrated a *collision attack* in Feb 2017
  - … Google had to run >9 quintillion SHA-1 computations to complete the attack
  - ... but already being phased out since weaknesses were found earlier
- Was for message integrity in GitHub (SHA-256 fully supported as of 2023)

**SHA-2**

*Believed to be secure*
- Designed by the NSA; published by NIST
- Variations based on bit length: SHA-224, SHA-256, SHA-384, SHA-512
- Linux passwords use SHA-512
- Bitcoin uses SHA-256

**SHA-3**

*Believed to be secure*
- 256 & 512 bit

**Blowfish**
- Used for password hashing in OpenBSD

**3DES**
- Linux passwords used to use this

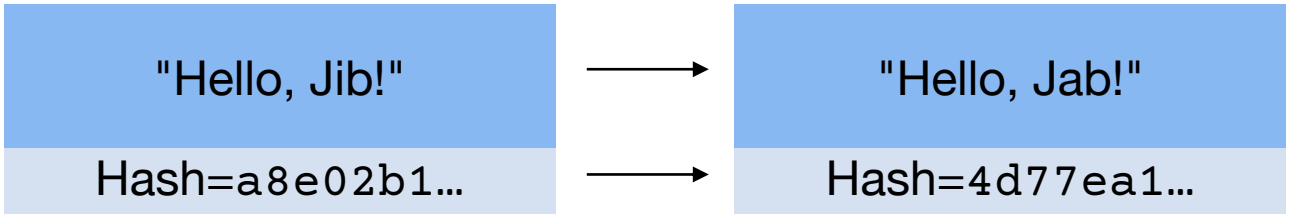
← Derived from ciphers ←

# Tamperproof Integrity: Message Authentication Codes and Digital Signatures

# Message Integrity: MACs

- We rely on hashes to assert the integrity of messages

- An attacker can create a new message *M′* & a new hash and replace *H(M)* with *H(M′)*




- So, let's create a checksum that relies on a key for validation:

  Message Authentication Code (MAC) = *hash(M, key)*

# Message Authentication Codes (MAC)

Hash of message and a symmetric key:
An intruder will not be able to replace the hash value

- *You need to have the key and the message to recreate the hash*

MACs provide message **integrity**

- The hash assures us that the original message has not been modified
- The encryption of the hash assures us that an attacker could not have re-created the hash

# Digital Signatures

Create a hash that anyone can verify but *only* the owner can create:

Hash of message encrypted with the owner's private key

- Alice encrypts the hash with her private key
- Bob validates by decrypting it with her public key & comparing with a hash of the message

Digital signatures add **non-repudiation**

- Only Alice could have created the signature because only Alice has her private key

# Digital Signature Primitives

1. **Key generation**: { signing_key, verification_key } := **gen_keys**(key_size)

   signing_key = private_key, *k*

   verification_key = public_key, *K*

2. **Signing**: signature := **sign**(message, private_key)

   signature := **sign**(message, private_key)

   ⇒ signature := $E_k$(hash(message))

   The signature uses a *hash(message)* instead of the *message*

   - We'd like the signature to be a small, fixed size
   - We are not hiding the contents of the message
   - We trust hashes to be collision-free

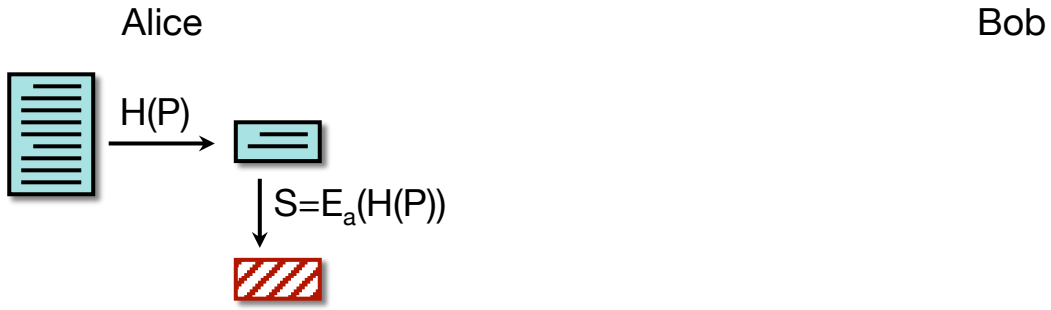
3. **Validation**: verify(verification_key, message, signature)

   $D_K$(signature) $\stackrel{?}{=}$ hash(message)

# Digital signatures

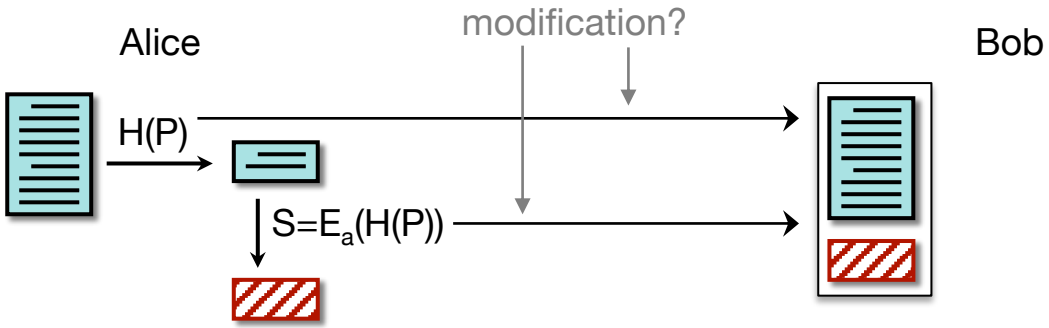

Alice generates a hash of the message, H(P)

# Digital signatures: public key cryptography



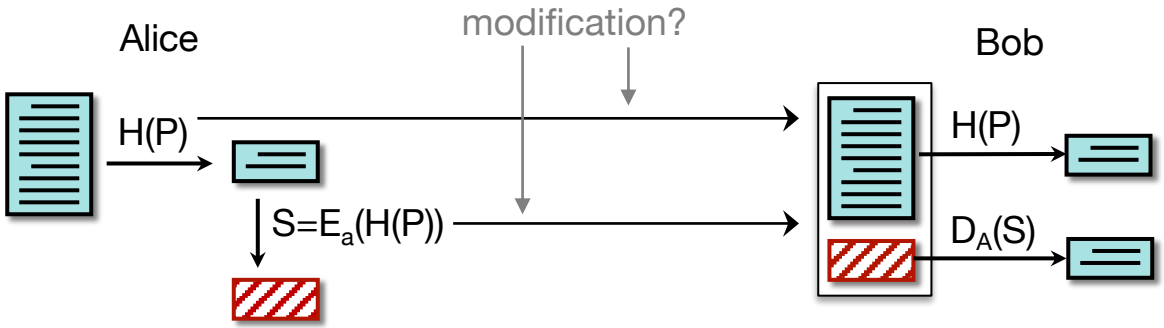

Alice encrypts the hash with her private key
This is her signature.

# Using Digital Signatures



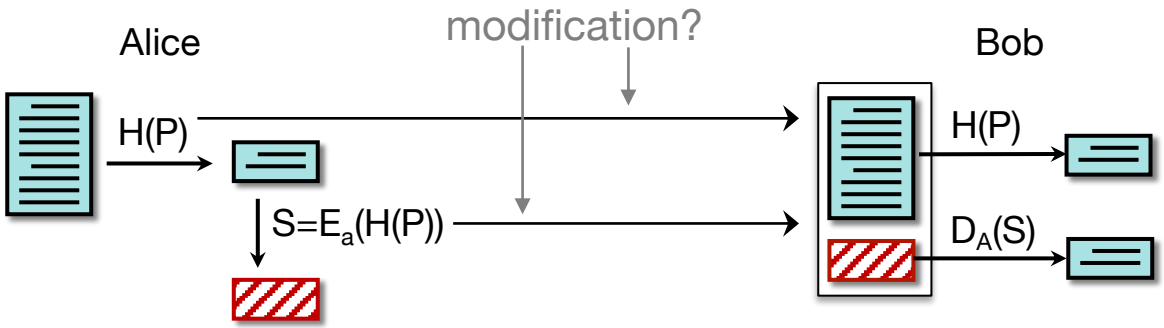

Alice sends Bob the message & the encrypted hash

# Using Digital Signatures




1. Bob decrypts the hash using Alice's public key
2. Bob computes the hash of the message sent by Alice

# Using Digital Signatures



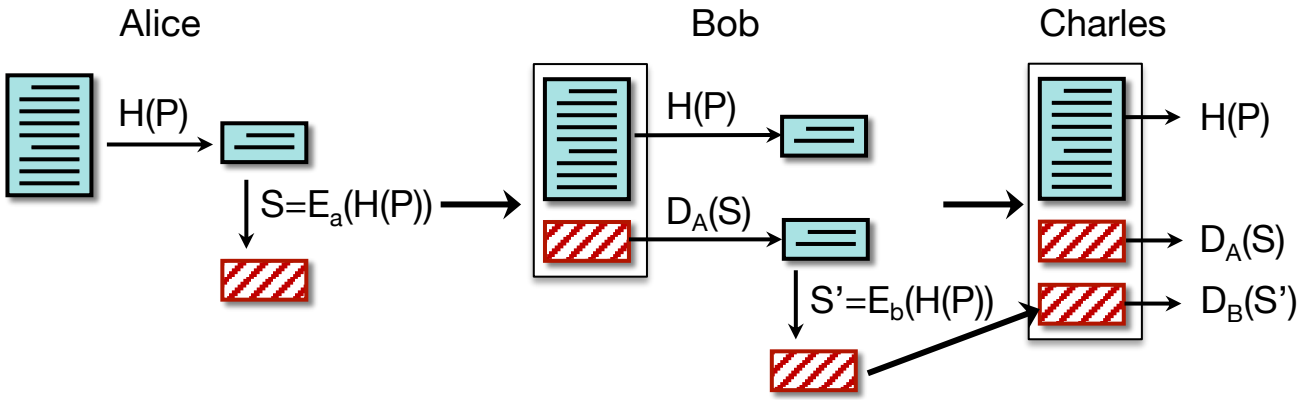

If the hashes match, the signature is valid
⇒ the encrypted hash *must* have been generated by Alice

# Digital signatures & non-repudiation

- **Digital signatures provide non-repudiation**
  - Only Alice could have created the signature because only Alice has her private key

- **Proof of integrity**
  - The hash assures us that the original message has not been modified
  - The encryption of the hash assures us that an attacker could not have re-created the hash

# Digital signatures: multiple signers




Charles:

- Generates a hash of the message, H(P)
- Decrypts Alice's signature with Alice's public key
  - Validates the signature: $D_A(S) \stackrel{?}{=} H(P)$
- Decrypts Bob's signature with Bob's public key
  - Validates the signature: $D_B(S) \stackrel{?}{=} H(P)$

# Digital Signature Algorithms

While encrypting a hash with a private key produces a digital signature, there are dedicated algorithms that use public key cryptography to produce and validate signatures

- **RSA** – based on RSA cryptography & keys
  - Difficulty based on factoring products of primes
- **DSA** (Digital Signature Algorithm)
  - Developed by NIST (National Institute of Standards and Technology)
  - Difficulty based on discrete logarithms & modular exponentiation
- **ECDSA**, **EdDSA**: Elliptic Curve Digital Signature Algorithm
  - Difficulty based on discrete logarithms on elliptic curves
  - (EdDSA = Edwards curve DSA – uses Twisted Edwards curves)
  - Newest & fastest signature algorithm

# Covert AND authenticated messaging

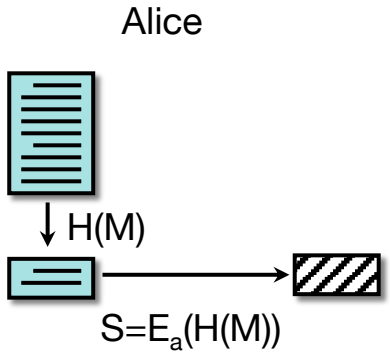
If we want to keep the message secret

- combine encryption with a digital signature

Use a **session key**:

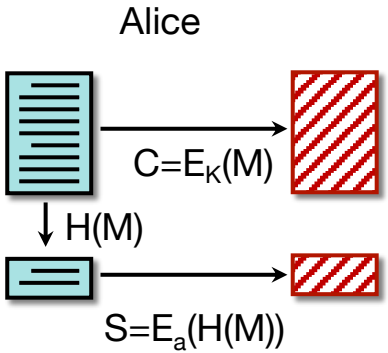
- Pick a random key, *K*, to encrypt the message with a symmetric algorithm
- Encrypt *K* with the public key of each recipient
- For signing, encrypt the hash of the message with sender's private key

# Covert and authenticated messaging



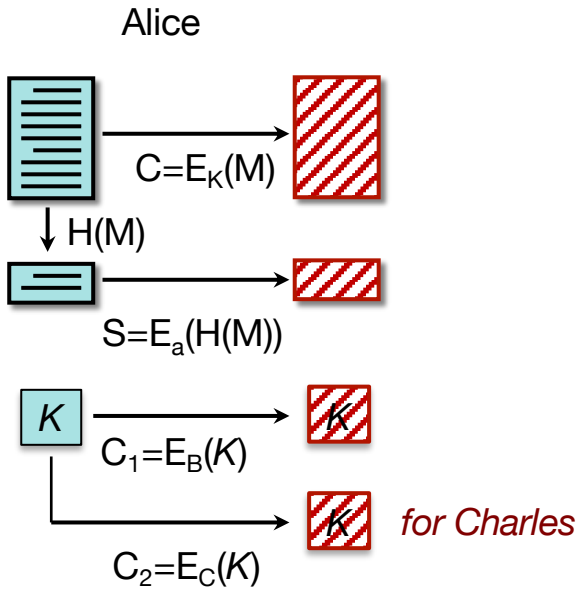

Alice generates a digital signature by
encrypting the message with her private key

# Covert and authenticated messaging



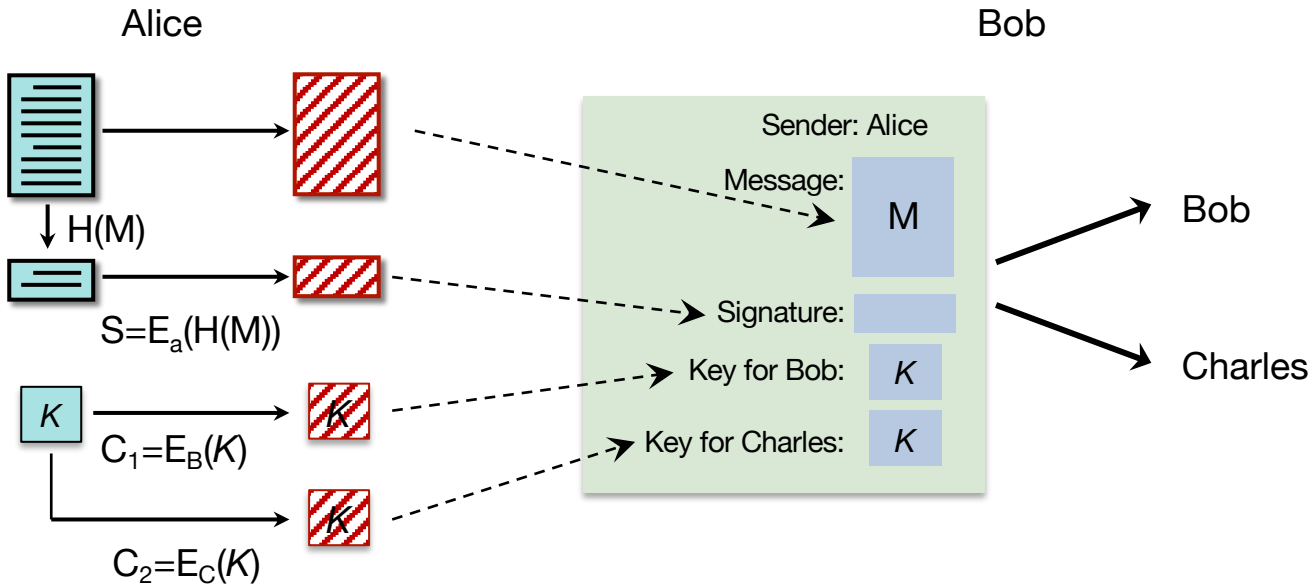

Alice picks a random key, *K*, and encrypts the message *M* with it using a symmetric cipher

# Covert and authenticated messaging




Alice encrypts the session key for each recipient of this message using their public keys

# Covert and authenticated messaging




The aggregate message is sent to Bob & Charles

# The End