Computer Security

07. Cryptography: Public keys, Integrity

Paul Krzyzanowski

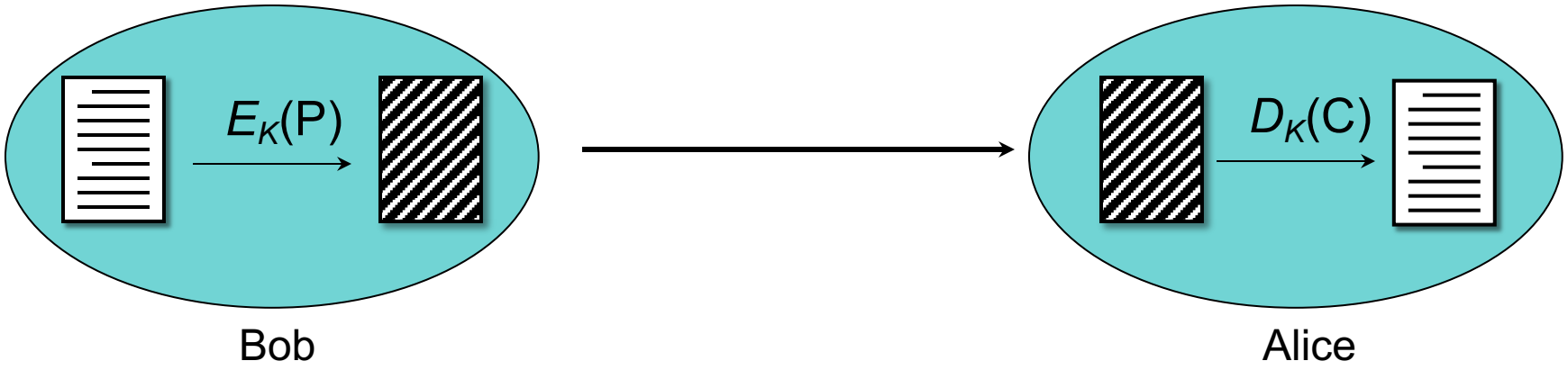
Rutgers University

Fall 2019

Key Distribution

Communicating with symmetric cryptography

- Both parties must agree on a secret key, *K*
- Message is encrypted, sent, decrypted at other side

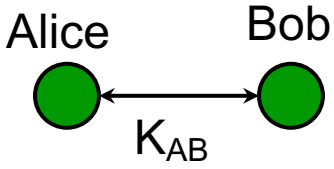


Key distribution must be secret

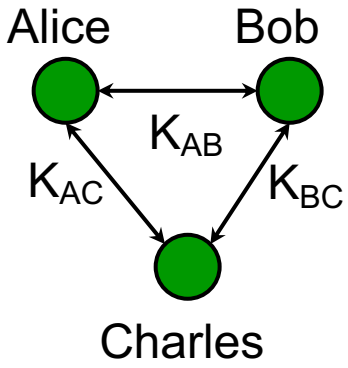
- Otherwise messages can be decrypted
- Users can be impersonated

Key explosion

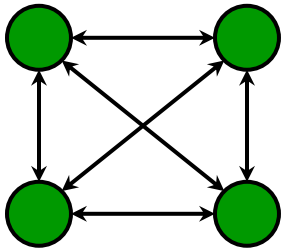
Each pair of users needs a separate key for secure communication



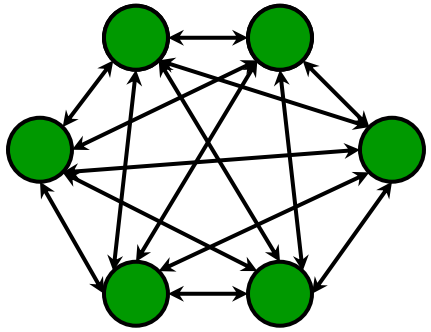
2 users: 1 key



3 users: 3 keys



4 users: 6 keys



6 users: 15 keys

100 users: 4,950 keys

1000 users: 399,500 keys

n users: $\frac{n(n-1)}{2}$ keys

Key distribution

Secure key distribution is the biggest problem with symmetric cryptography

Public Key Cryptography

Public-key algorithm

Two related keys.

$$C = E_{K1}(P) \qquad P = D_{K2}(C)$$
$$C' = E_{K2}(P) \qquad P = D_{K1}(C')$$

K_1 is a public key

K_2 is a private key

Examples:

RSA, Elliptic curve algorithms
DSS (digital signature standard)

Trapdoor functions

Public key cryptography relies on trapdoor functions

- Trapdoor function
  - Easy to compute in one direction
  - Inverse is difficult to compute without extra information

Example:

96171919154952919 is the product of two prime #s
What are they?

But if you're told that one of them is 100225441
then it's easy to compute the other: 959555959

RSA Public Key Cryptography

Ron Rivest, Adi Shamir, Leonard Adleman created the first public key encryption algorithm in 1977

Each user generates two keys:

- Private key (kept secret)
- Public key (can be shared with anyone)

Difficulty of algorithm based on the difficulty of factoring large numbers

- Keys are functions of a pair of large (~300 digits) prime numbers

RSA algorithm: key generation

1. Choose two random large prime numbers p, q
2. Compute the product $n = pq$ and $\phi = (p - 1)(q - 1)$
   n is part of the private key. Length(n) is the key length
3. Choose the **public exponent**, e, such that:
   $1 < e < \phi$ and gcd(e, ϕ) = 1
   [e and $(p - 1)(q - 1)$ are relatively prime]
4. Compute the **secret exponent**, d such that:
   $ed = 1 \bmod \phi$
   $d = e^{-1} \bmod ((p - 1)(q - 1))$
5. Public key = (e, n)
   Private key = (d, n)
   Discard p, q, ϕ

See https://www.di-mgt.com.au/rsa_alg.html

RSA Encryption

Key pair: public key = (e, n)
private key = (d, n)

Encrypt

- Divide data into numerical blocks < n
- Encrypt each block:

$$c = m^e \bmod n$$

Decrypt

$$m = c^d \bmod n$$

RSA security

The security of RSA encryption rests on the difficulty of factoring a large integer

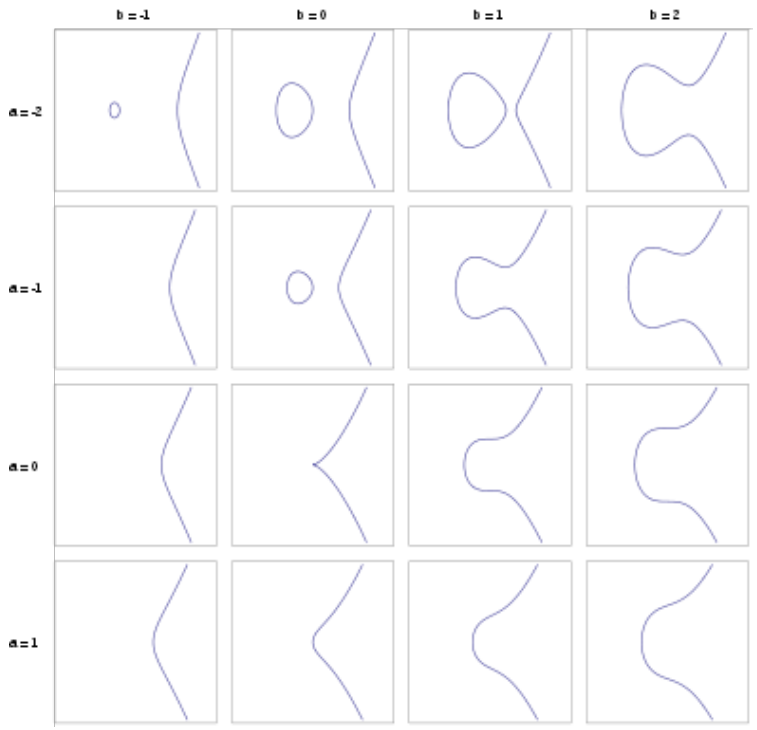
Public key = { *modulus*, *exponent* }, or {*n*, *e*}

- The *modulus* is the product of two primes, *p*, *q*
- The private key is derived from the same two primes

Elliptic Curve Cryptography

Alternate approach: elliptic curves

$$y^2 = x^3 + ax + b$$



Catalog of elliptic curves
https://en.wikipedia.org/wiki/Elliptic_curve

Using discrete numbers, pick

- A prime number as a maximum (modulus)
- A curve equation
- A public base point on the curve
- A random private key
- Public key is derived from the private key, the base point, and the curve

To compute the private key from the public,

- We would need an elliptic curve discrete logarithm function
- This is difficult and is the basis for ECC's security

ECC vs. RSA

- RSA is still the most widely used public key cryptosystem
  - Mostly due to inertia & widespread implementations
  - Faster for decryption
  - Simpler implementation
- ECC offers higher security with fewer bits than RSA
  - ECC is faster for key generation & encryption
  - Uses less memory
  - NIST defines 15 standard curves for ECC
    - But many implementations support only a couple (P-256, P-384)

https://www.keylength.com/en/4/

http://https://www.enisa.europa.eu/publications/algorithms-key-size-and-parameters-report-2014

Key length

Unlike symmetric cryptography, not every number is a valid key with RSA and ECC

Comparable complexity:

- 3072-bit RSA = 256-bit elliptic curve = 128-bit symmetric cipher
- 15360-bit RSA = 521-bit elliptic curve = 256-bit symmetric cipher

For long-term security

The European Union Agency for Network and Information Security (ENISA) and the National Institute for Science & Technology (NIST) recommend:

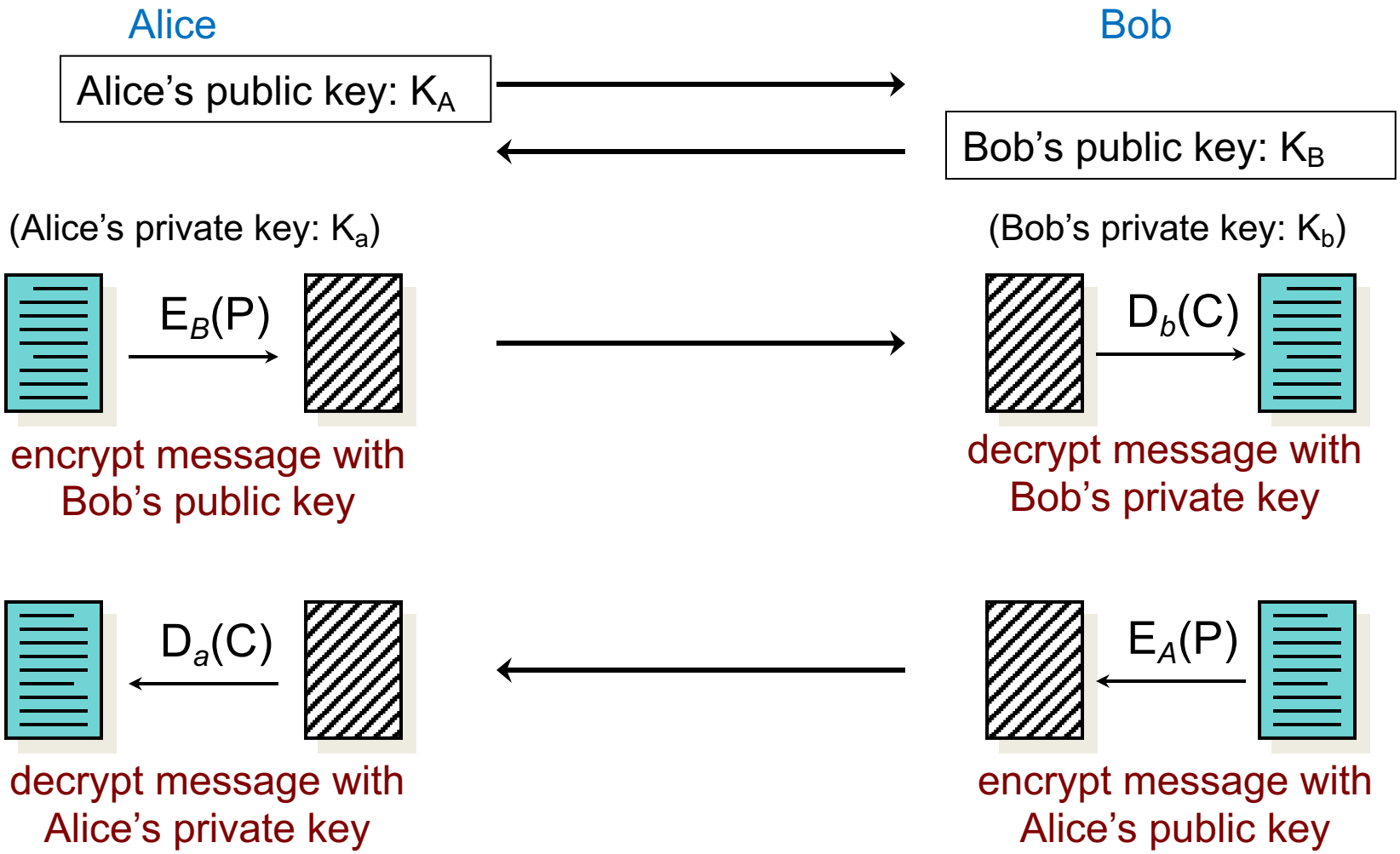
- AES: 256-bit keys
- RSA: 15,360-bit keys
- ECC: 512 bit-keys

Communication with public key algorithms

Different keys for encrypting and decrypting

- No need to worry about key distribution

Communication with public key algorithms



RSA isn't good for communication

Calculations are very expensive relative to symmetric algorithms

Common speeds:

Algorithm	Bytes/sec
AES-128-ECB	148,000,000
AES-128-CBC	153,000,000
AES-256-ECB	114,240,000
RSA-2048 encrypt	3,800,000
RSA-2048 decrypt	96,000

- AES ~1500x faster to decrypt; 40x faster to encrypt than RSA

If anyone learns your private key, they can read all your messages

Key Exchange

Diffie-Hellman Key Exchange

Key distribution algorithm

- Allows two parties to exchange keys securely
- *Not* public key encryption
- Based on difficulty of computing discrete logarithms in a finite field compared with ease of calculating exponentiation

Allows us to negotiate a secret **common key** without fear of eavesdroppers

Diffie-Hellman Key Exchange

- All arithmetic performed in a field of integers modulo some large number
- Both parties agree on
  - a **large prime number *p***
  - and a number $\alpha < p$
- Each party generates a public/private key pair

  Private key for user *i*: X_i

  Public key for user *i*: $Y_i = \alpha^{X_i} \bmod p$

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B

K = (Bob's public key) (Alice's private key) mod p

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B
- Bob computes

$$K = Y_A^{X_B} \bmod p$$

K' = (Alice's public key) (Bob's private key) mod p

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- expanding:

$$\begin{aligned} K &= Y_B^{X_A} \bmod p \\ &= (\alpha^{X_B} \bmod p)^{X_A} \bmod p \\ &= \alpha^{X_B X_A} \bmod p \end{aligned}$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B
- Bob computes

$$K = Y_A^{X_B} \bmod p$$

- expanding:

$$\begin{aligned} K &= Y_B^{X_B} \bmod p \\ &= (\alpha^{X_A} \bmod p)^{X_B} \bmod p \\ &= \alpha^{X_A X_B} \bmod p \end{aligned}$$

$$\boldsymbol{K = K'}$$

K is a *common key*, known *only* to Bob and Alice

Hybrid Cryptosystems

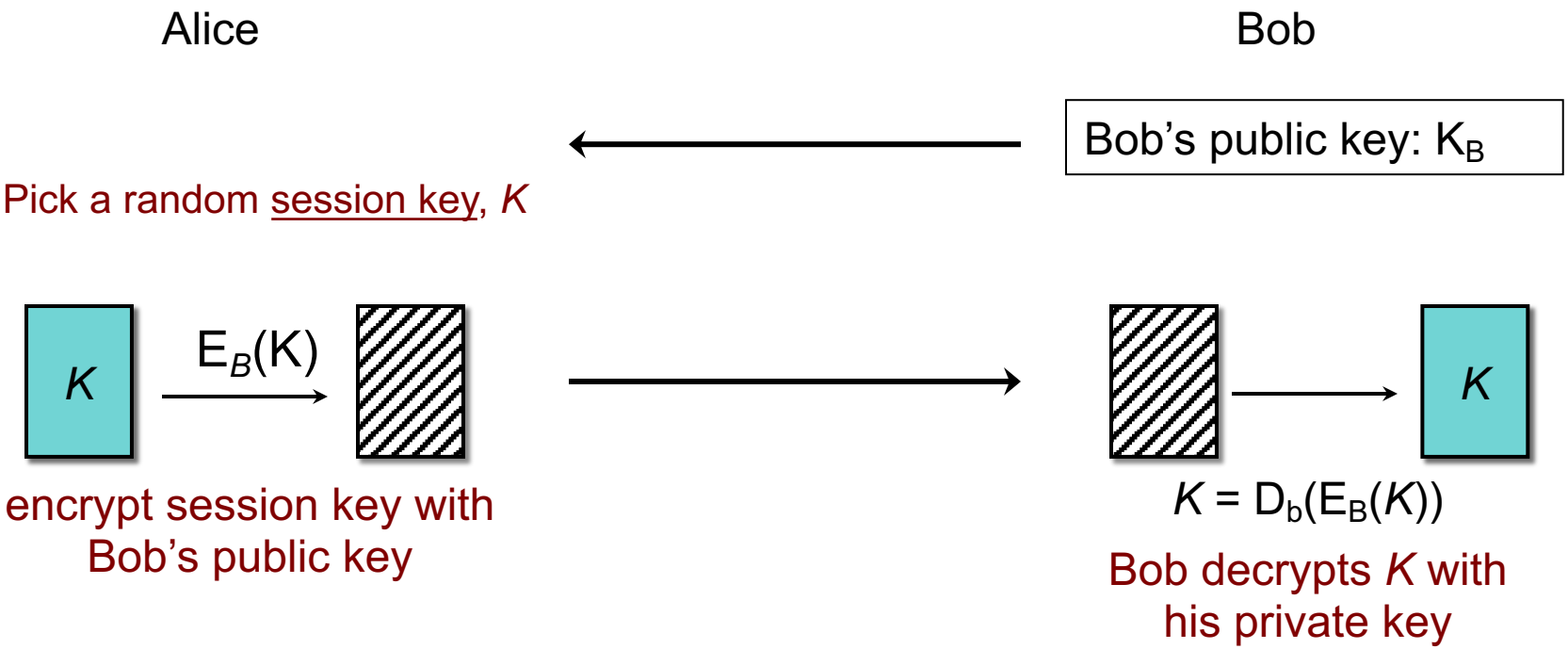
Hybrid Cryptosystems

- Session key: randomly-generated key for one communication session
- Use a public key algorithm to send the session key
- Use a symmetric algorithm to encrypt data with the session key

Public key algorithms are almost never used to encrypt messages

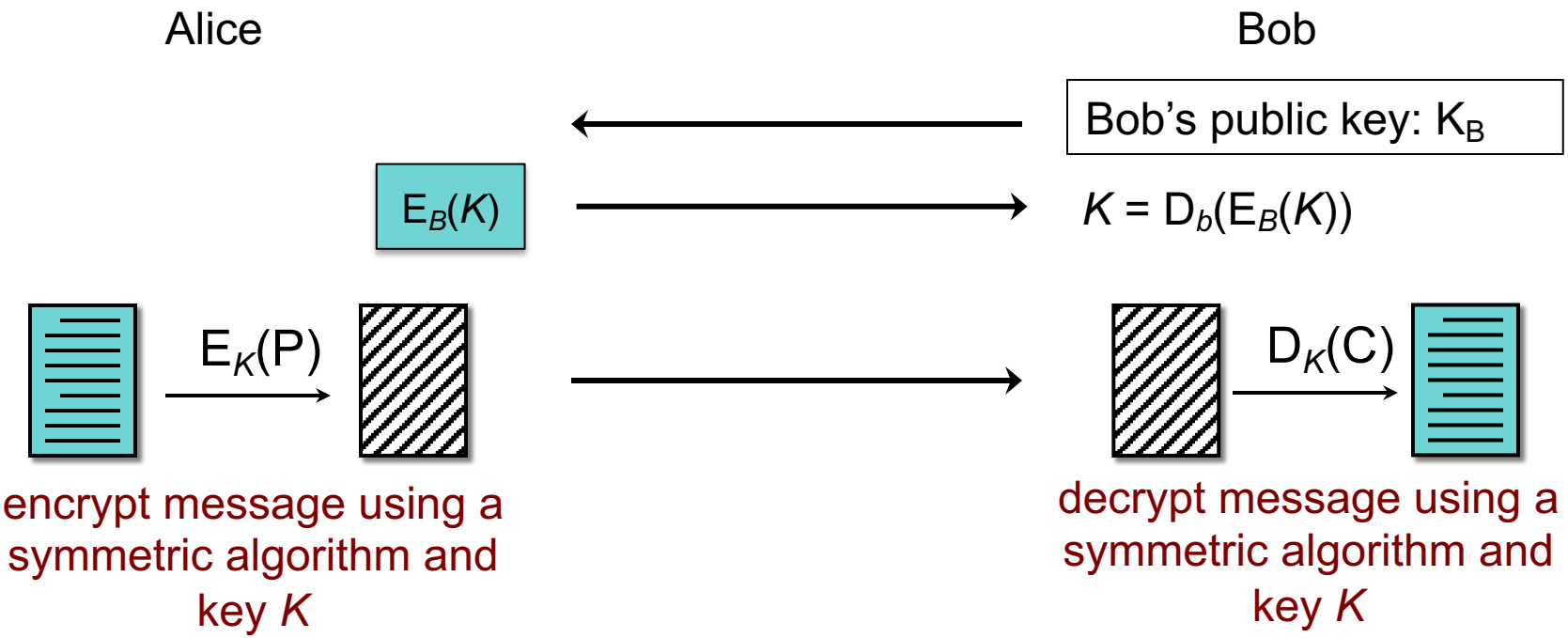
- MUCH slower; vulnerable to *chosen-plaintext attacks*
- RSA-2048 approximately 55x slower to encrypt and 2,000x slower to decrypt than AES-256

Communication with a hybrid cryptosystem

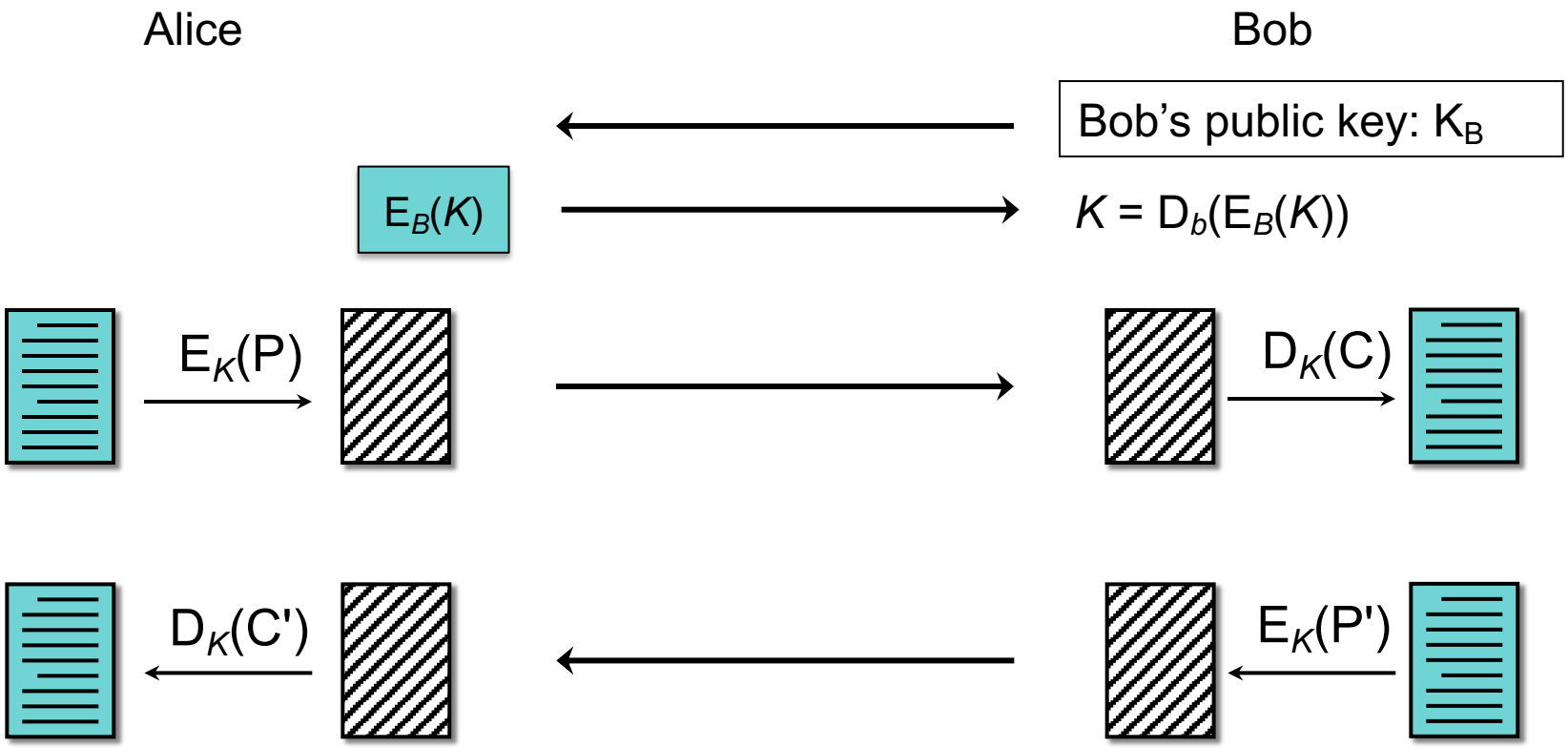


Now Bob knows the secret session key, K

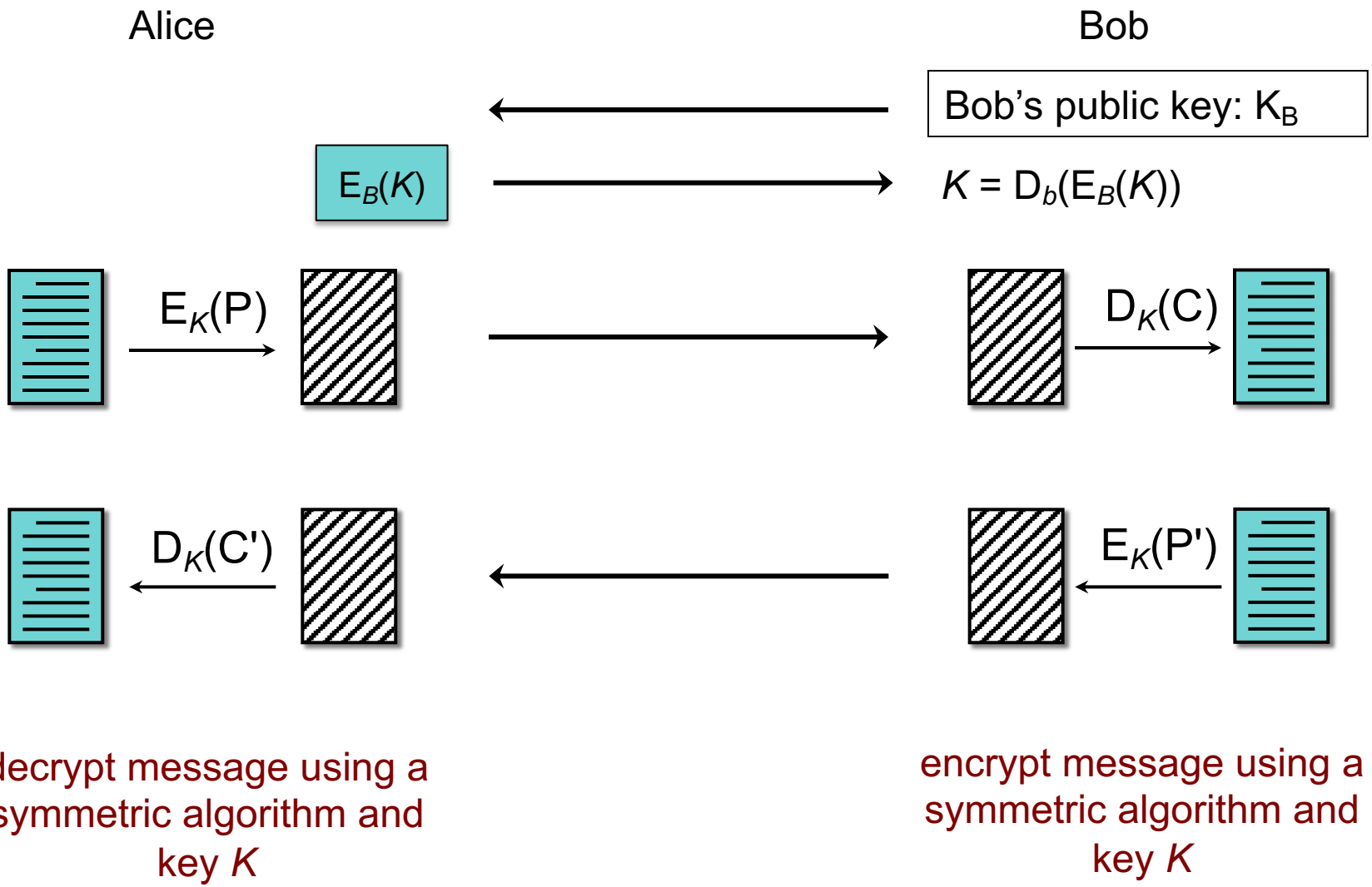
Communication with a hybrid cryptosystem



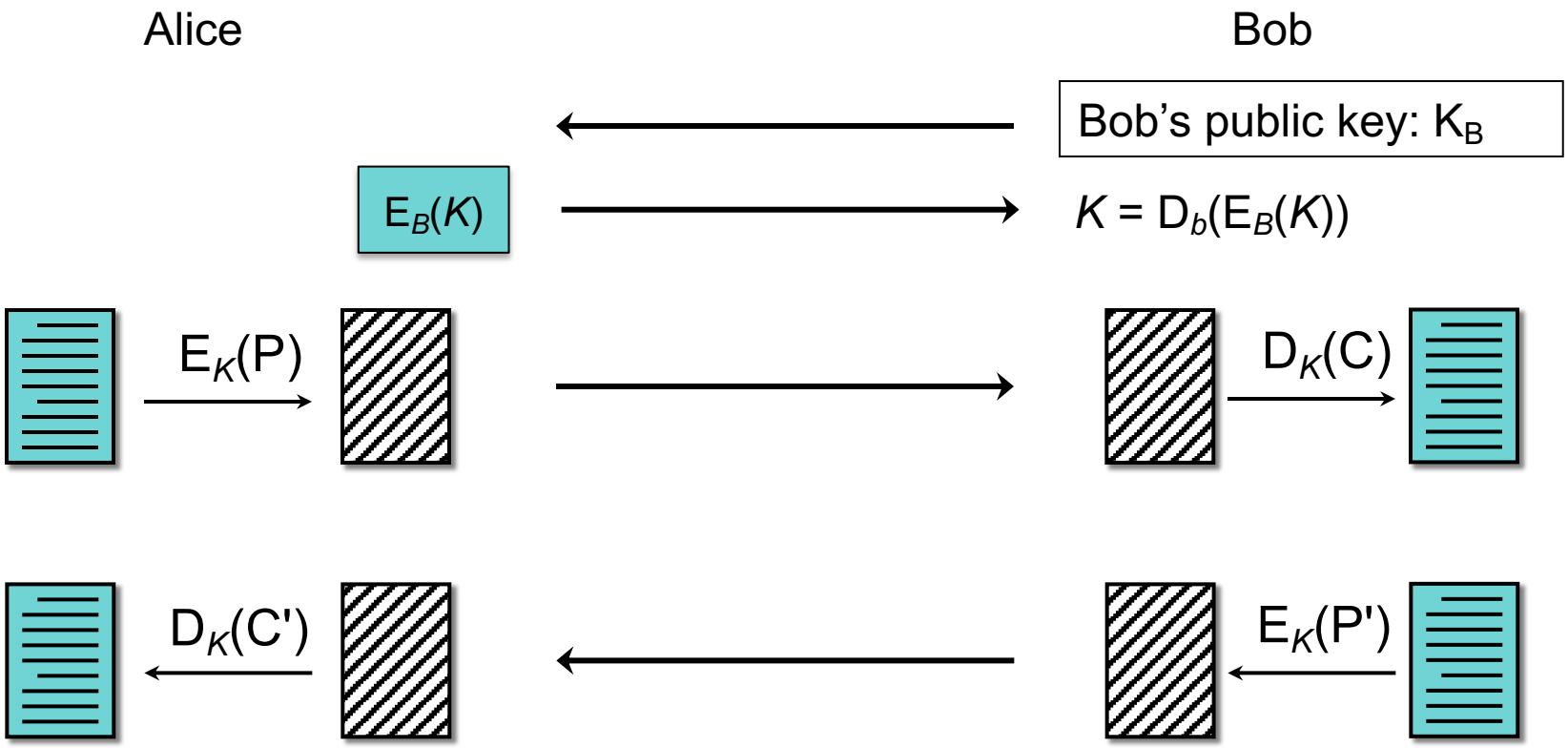
Communication with a hybrid cryptosystem



Communication with a hybrid cryptosystem

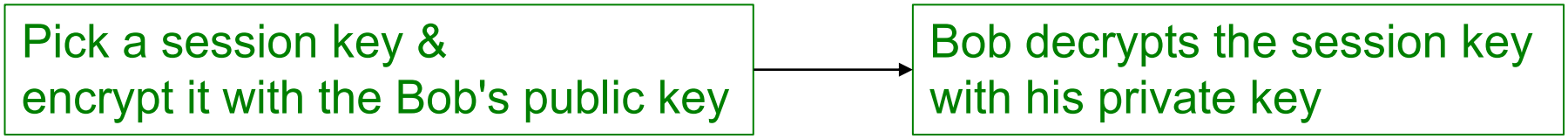


Communication with a hybrid cryptosystem



Forward Secrecy

Forward Secrecy



Suppose an attacker steals Bob's private key

- Future messages can be compromised
- He can go through past messages & decrypt old session keys

Security rests entirely on the secrecy of Bob's private key

- If Bob's private key is compromised, all recorded past traffic can be decrypted

Forward Secrecy

Forward secrecy

- Compromise of long-term keys does not compromise past session keys
- There is no one secret to steal that will compromise multiple messages

Diffie-Hellman key exchange is commonly used for key exchange

- Generate a set of keys per session
- Use the derived common key as the encryption/decryption key

  …or as a key to encrypt a session key
- Not recoverable as long as private keys are thrown away

  Unlike RSA keys, Diffie-Hellman makes key generation simple

Keys must be **ephemeral**

- Client & server will generate new Diffie-Hellman parameters for each session – all will be thrown away after the session

Diffie-Hellman is preferred over RSA for key exchange to achieve forward secrecy – generating Diffie-Hellman keys is a rapid, low-overhead process

Cryptographic systems: summary

- Symmetric ciphers
  - Based on “SP networks” = substitution & permutation sequences
- Asymmetric ciphers – public key cryptosystems
  - Based on trapdoor functions
    Easy to compute in one direction; difficult to compute in the other direction without special information (the trapdoor)
- Hybrid cryptosystem
  - Pick a random session key
  - Use a public key algorithm to send
  - Use a symmetric key algorithm to encrypt traffic back & forth
- Key exchange algorithms (more to come later)
  - Diffie-Hellman
  - Public key

Enables secure communication without knowledge of a shared secret

Looking ahead

RSA cryptography in the future

- Based on the difficulty of factoring products of two large primes
- Factoring algorithms get more efficient as numbers get larger
  - As the ability to decrypt numbers increases, the key size must therefore grow even faster
  - This is not sustainable (especially for embedded devices)
- ECC is a better choice for most applications

Quantum Computers

Once (if) real quantum computers can be built, they can

– Factor efficiently
  - Shor's algorithm factors numbers exponentially faster
  - RSA will not be secure anymore

– Find discrete logarithms & elliptic curve discrete logarithms efficiently
  - Diffie-Hellman key exchange & ECC will not be secure

Symmetric cryptography is largely immune to attacks

- Some optimizations are predicted: crack a symmetric cipher in time proportional to the square root of the key space size: $2^{n/2}$ vs. 2^n
  – Use 256-bit AES to be safe

2016: NSA called for a migration to "post-quantum cryptographic algorithms" – but no agreement yet on what those will be

2019: Narrowed submissions down to 26 leading candidates

Quantum Computers

Quantum computing is not faster at everything

There are only four types of algorithms currently knows where quantum computing offers an advantage

- Researchers are developing algorithms that are based on problems quantum computers do not help with
- 2017: IBM presented CRYSTALS (Cryptographic Suite for Algebraic Lattices) – based on a category of equations called *lattice problems*
  - Example: add 3 out of a set of 5 numbers
  - Give the sum to a friend and ask them to determine which numbers were added
  - Try this if someone picks 500 out of 1,000 numbers with 1,000 digits each
- 2019: IBM demonstrated the use of this algorithm in a tape drive

https://www.scientificamerican.com/article/new-encryption-system-protects-data-from-quantum-computers/

Message Integrity

McCarthy's Spy Puzzle (1958)

The setting:

- Two countries are at war
- One country sends spies to the other country
- To return safely, spies must give the border guards a password

Conditions

- Spies can be trusted
- Guards chat – information given to them may leak

McCarthy's Spy Puzzle

Challenge

- How can a border guard authenticate a person without knowing the password?

- Enemies cannot use the guard's knowledge to introduce their own spies

Solution to McCarthy's puzzle

Michael Rabin, 1958

- Use a one-way function, $B = f(A)$
  - Guards get B
    - Enemy cannot compute A if they know A
  - Spies give A, guards compute f(A)
    - If the result is B, the password is correct.

- Example function:
  - Middle squares
    - Take a 100-digit number (A), and square it
    - Let B = middle 100 digits of 200-digit result

One-way functions

- Easy to compute in one direction
- Difficult to compute in the other

Examples:

Factoring:

$pq = N$	EASY
find p,q given N	DIFFICULT

} Basis for RSA

Discrete Log:

$a^b \bmod c = N$	EASY
find b given a, c, N	DIFFICULT

} Basis for Diffie-Hellman & Elliptic Curve

Example of a one-way function

Example with an 18 digit number

A = 289407349786637777

A^2 = 837566141105253089484453382035017 29

Middle square, B = 1105253089484453 38

Given A, it is easy to compute B

Given B, it is difficult to compute A

"Difficult" = no known short-cuts; requires an exhaustive search

Cryptographic hash functions

Cryptographic hash functions

Also known as a 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 authentication

- 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

Hash Algorithms

Use iterative structure like block ciphers do … but use no key

- Example:
  - Secure Hash Algorithm, **SHA-1**
    - Designed by the NSA in 1993; revised in 1995
    - US standard for use with NIST Digital Signature Standard (DSS)
    - Produces 160-bit hash values
    - Chosen prefix collision attacks demonstrated May 2019
- Successors
  - **SHA-2** (2001)
    - Produces 224, 256, 384, or 512-bit hashes
    - Approved for use with the NIST Digital Signature Standard (DSS)
  - **SHA-3** (2015)
    - Can be substituted for SHA-2
    - Improved robustness

Example: SHA-1 Overview

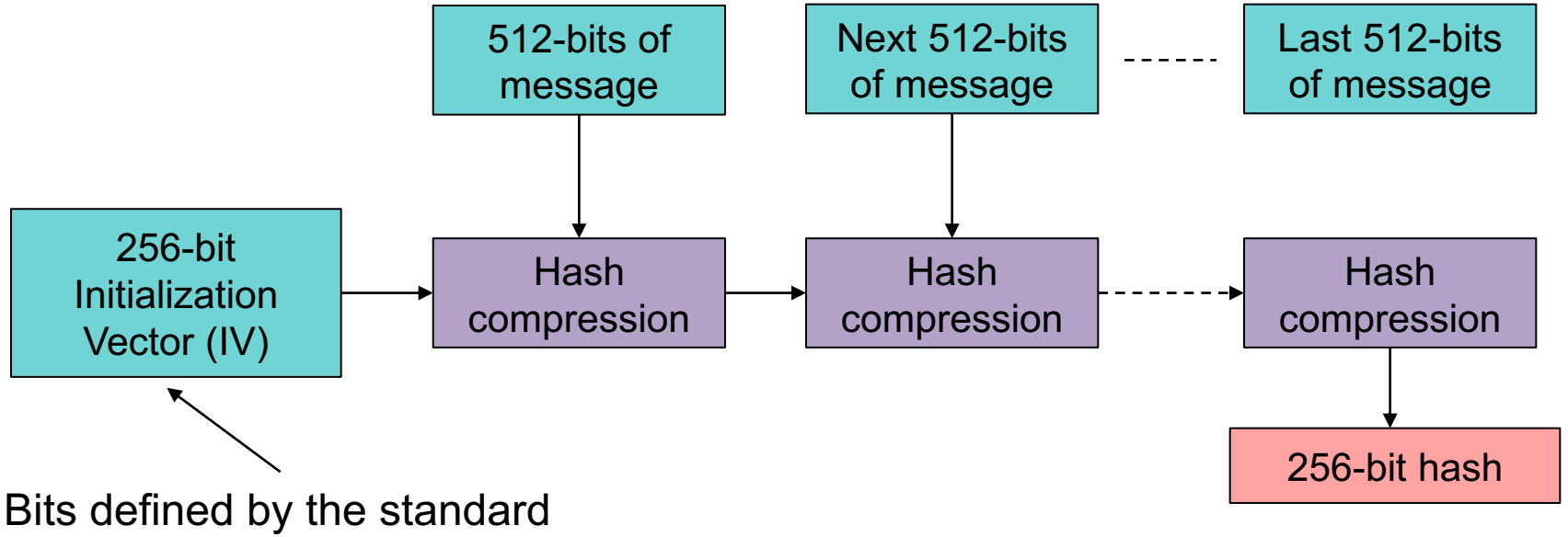
- Prepare the message
  - Append the bit 1 to the message
  - Pad message with 0 bits so its length = 448 mod 512
  - Append length of message as a 64-bit big endian integer
- Use an Initialization Vector (IV) = 5-word (160-bit) buffer:

```
a = 0x67452301   b = 0xefcdab89   c = 0x98badcfe
d = 0x10325476   e = 0xc3d2e1f0
```

- Process the message in 512-bit chunks
  - Expand the 16 32-bit words into 80 32-bit words via XORs & shifts
  - Iterate 80 times to create a hash for this chunk
    - Various sets of ANDs, ORs, XORs, and shifts
  - Add this hash chunk to the result so far

See https://www.saylor.org/site/wp-content/uploads/2012/07/SHA-1-1.pdf

SHA-2 Overview



Popular hash functions

- MD5 *R.I.P.*
  - 128 bits (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
  - Used for message integrity in GitHub
- SHA-2 – *believed to be secure*
  - Designed by the NSA; published by NIST
  - SHA-224, SHA-256, SHA-384, SHA-512
    - e.g., Linux passwords used MD5 and now SHA-512
  - SHA-256 used in bitcoin
- SHA-3 – *believed to be secure*
  - 256 & 512 bit
- Derivations from ciphers:
  - Blowfish (used for password hashing in OpenBSD)
  - 3DES – used for old Linux password hashes

Linux commands

sha1sum: create a SHA-1 hash

```
echo "hello, world!" | sha1sum
e91ba0972b9055187fa2efa8b5c156f487a8293a  -
```

sha3sum: create a 256-bit SHA-3 hash

```
echo "hello, world!" | sha3sum
c3d69513b79e0cdf3aa2b4afa38a5ffde144310109029e0e1aa57eb6  -
```

md5sum: create an MD5 hash

```
echo "hello, world!" | md5sum
910c8bc73110b0cd1bc5d2bcae782511  -
```

openssl: create a 512-bit SHA-3 hash (many other options available)

```
echo "hello, world!" | openssl dgst -sha3-512
(stdin)=8fc33b84ff22559082893fdc73f6877e590eb67533441fe5e48cd6d8a1
1aaf8d6270f82ef437c2c758000d65b09b45116b9c0eb3f3162149b13ca98c8cc8
c90f
```

Hash Collisions

Hashes are *collision resistant*, but collisions can occur

wikipedia

Pigeonhole principle

- If you have 10 pigeons & 9 compartments, at least one compartment will have more than one pigeon
- A hash is a fixed-size small number of bits (e.g., 256 bits = 32 bytes)
- Every possible permutation of an arbitrary number of bytes cannot fit into every permutation of 32 bytes!

Collisions: The Birthday Paradox

How many people need to be in a room such that the probability that two people will have the same birthday is > 0.5?

Your guess before you took a probability course: 183

This is true to the question of "how many people need to be in a room for the probability that someone else will have the same birthday as *one specific student*?"

Answer: 23

$$p(n) = 1 - \frac{n! \cdot \binom{365}{n}}{365^n}$$

Approximate solution for # people required to have a 0.5 chance of a shared birthday, where m = # days in a year

$$n \approx \sqrt{2 \times m \times 0.5}$$

The Birthday Paradox: Implications

- Searching for a collision with a pre-image (known message) is *A LOT* harder than searching for two messages that have the same hash

- Strength of a hash function is approximately ½ (# bits)
  - 256-bit hash function has a strength of approximately 128 bits
  - But that's a huge space!

    $2^{128} = 3.4 \times 10^{38}$

  - It's not feasible to try that many messages in the hope of finding a collision
    - BTW … the odds of winning the Powerball lottery are only 1:2.9×10^{8}

Message Integrity

How do we detect that a message has been tampered?

- A cryptographic hash acts as a checksum
- 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

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

Hash Pointers

We can use **hash pointers** instead of pointers in data structures

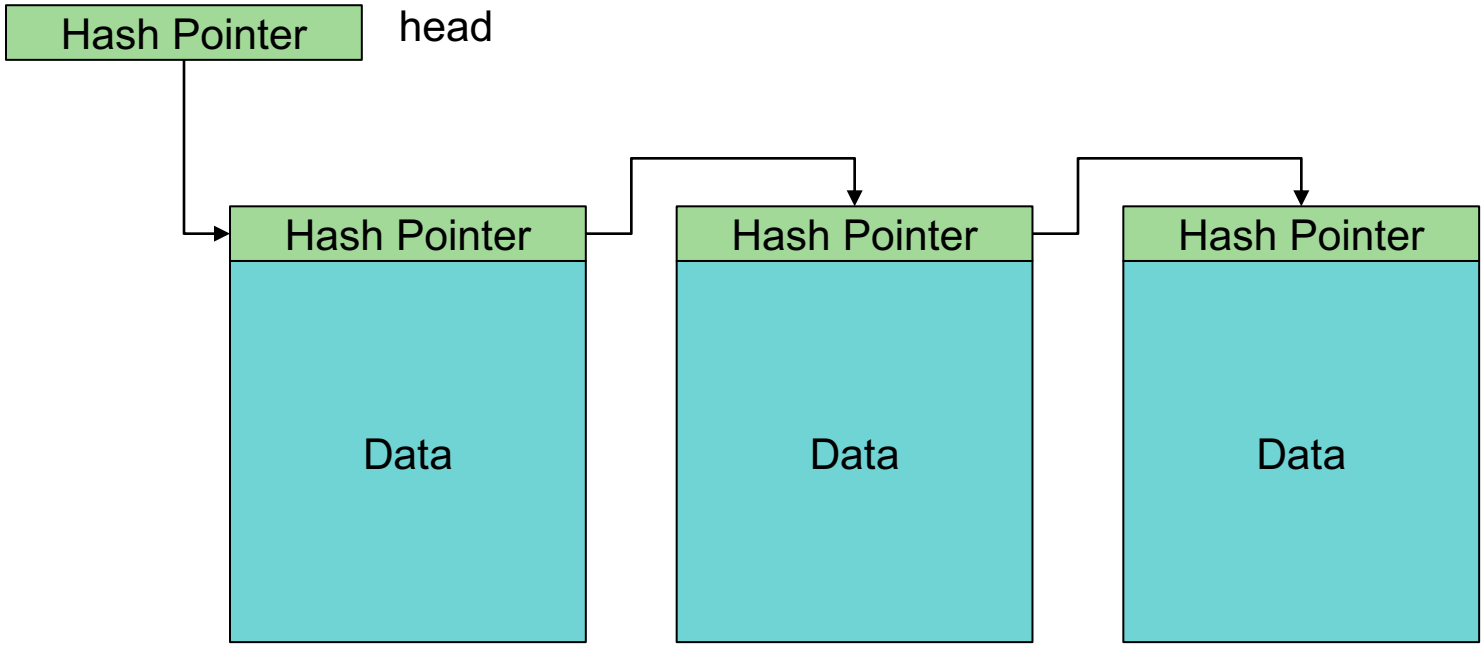
Hash pointer = { pointer, *hash*(data) }

- pointer = whatever we use to identify where the object is:
  memory location, file name, object ID, server/object, …
- *hash*(data) = hash function applied to the data being pointed to

This allows us to verify that the information we're pointing to has not changed

- Before using that data, do a hash(data) and see if it matches the hash in the hash pointer

Hash Pointers: Linked Lists = blockchain

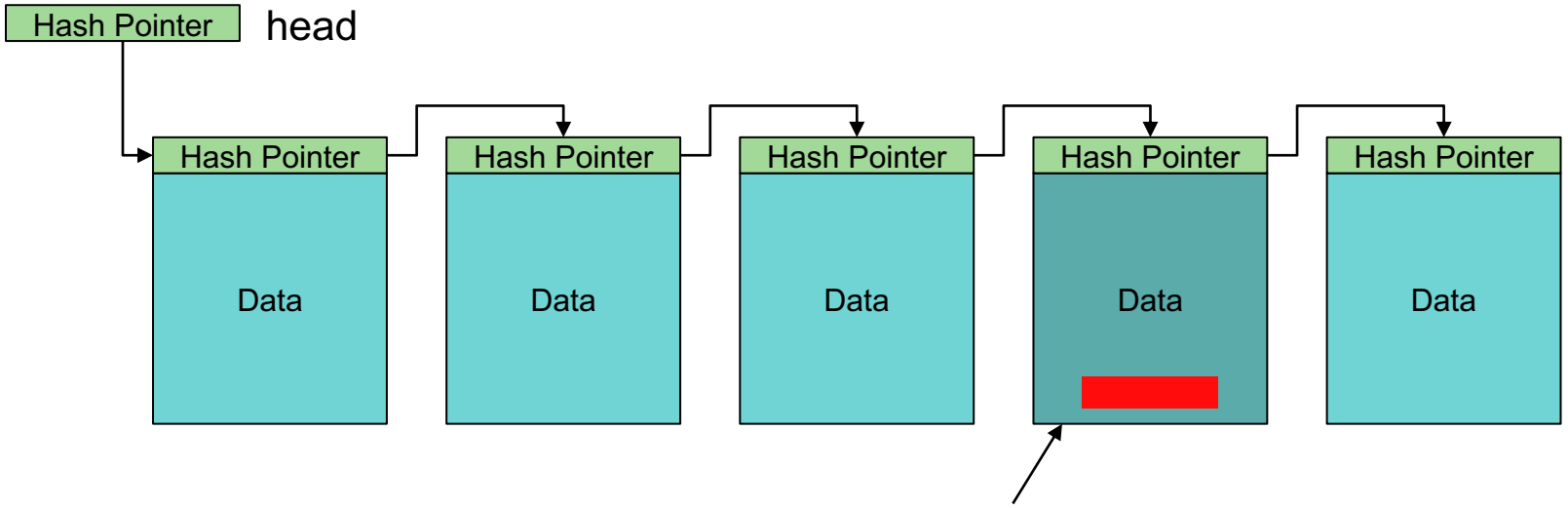


Add new data blocks to the end of the list

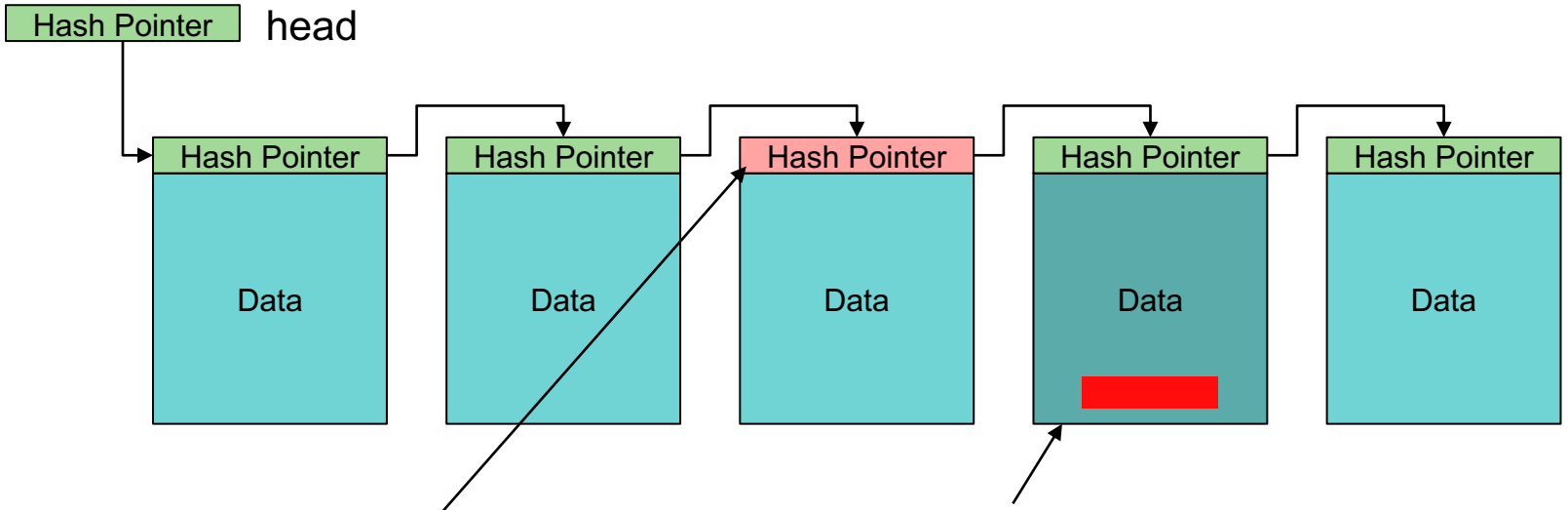
– Each hash pointer contains a pointer & a hash of the entire data structure to which it is pointing: the application data and the hash pointer

- Tamper Evident Log = blockchain

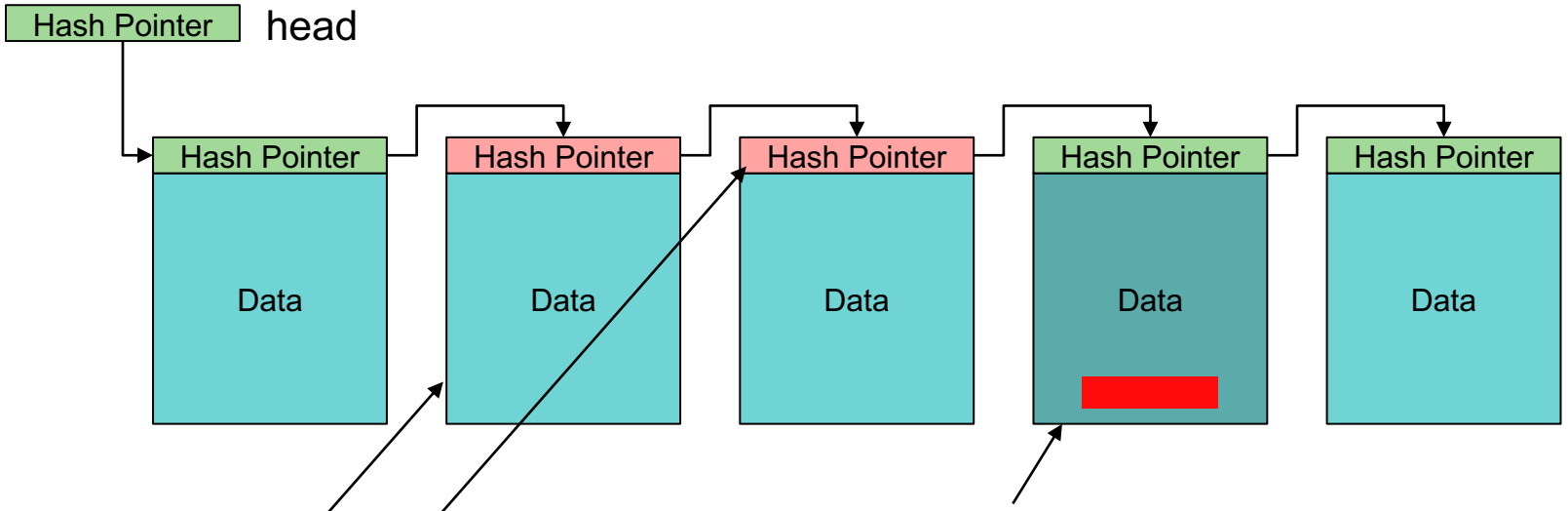
Tamper detection



Tamper detection



Tamper detection

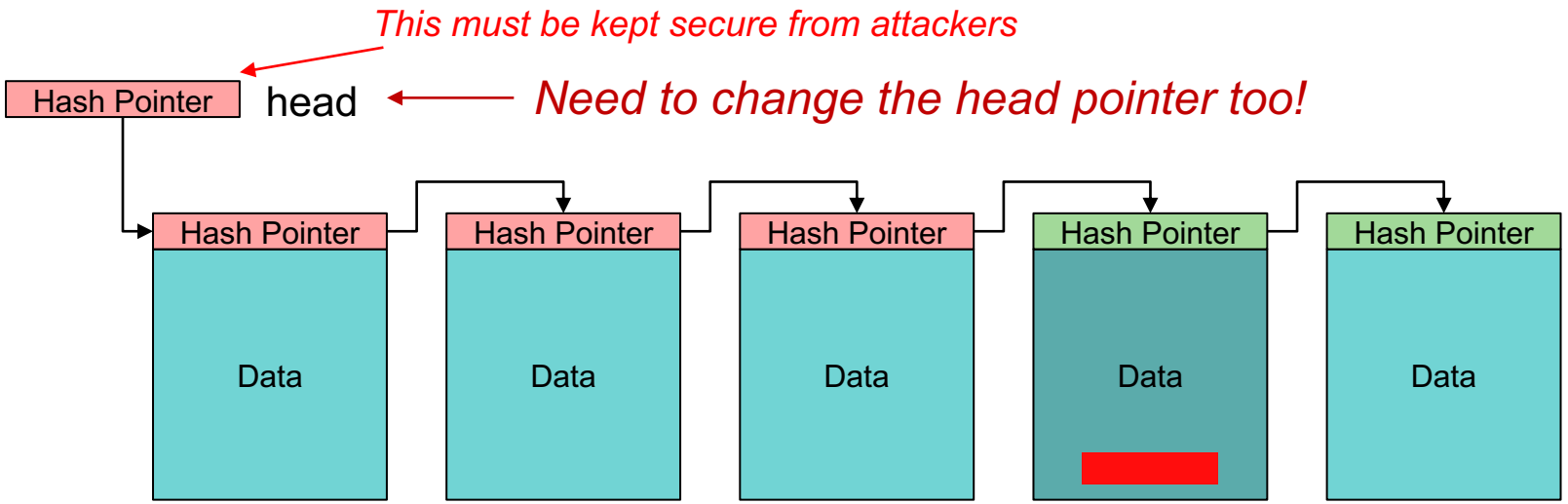


Suppose an adversary wants to data in this block

Then this hash pointer needs to be changed
But that will change the hash of this data block

So this hash pointer needs to be changed too
But now this data block hashes to a different value …

Tamper detection

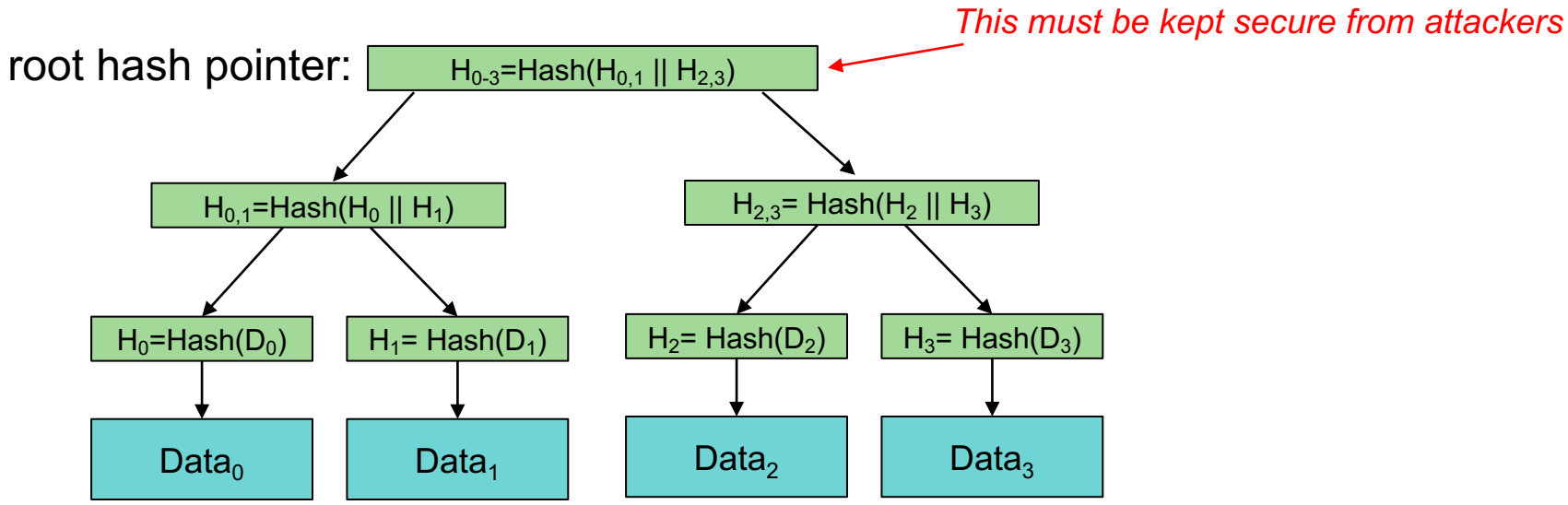


The adversary will have to change all hash pointers back to the root.

If we're holding on to the head of the list so an adversary cannot modify it, then we will be able to detect tampering.

Merkle Trees: Binary trees with hash pointers

Merkle Tree Hash pointer = { left_subtree, right_subtree, *hash*(left || right) }



- Another tamper-resistant structure
- Only need to examine $O(\log_2 n)$ hashes to validate data

a || *b* means *a* concatenated with *b*

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 & 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-based MAC

We can create a MAC from a cryptographic hash function

HMAC = Hash-based Message Authentication Code

$HMAC(m, k) = H((opad \oplus k) \,||\, H(ipad \oplus k) \,||\, m))$

Where

H = cryptographic hash function

opad = outer padding 0x5c5c5c5c … (01011100…)

ipad = inner padding 0x36363636… (00110110…)

k = secret key

m = message

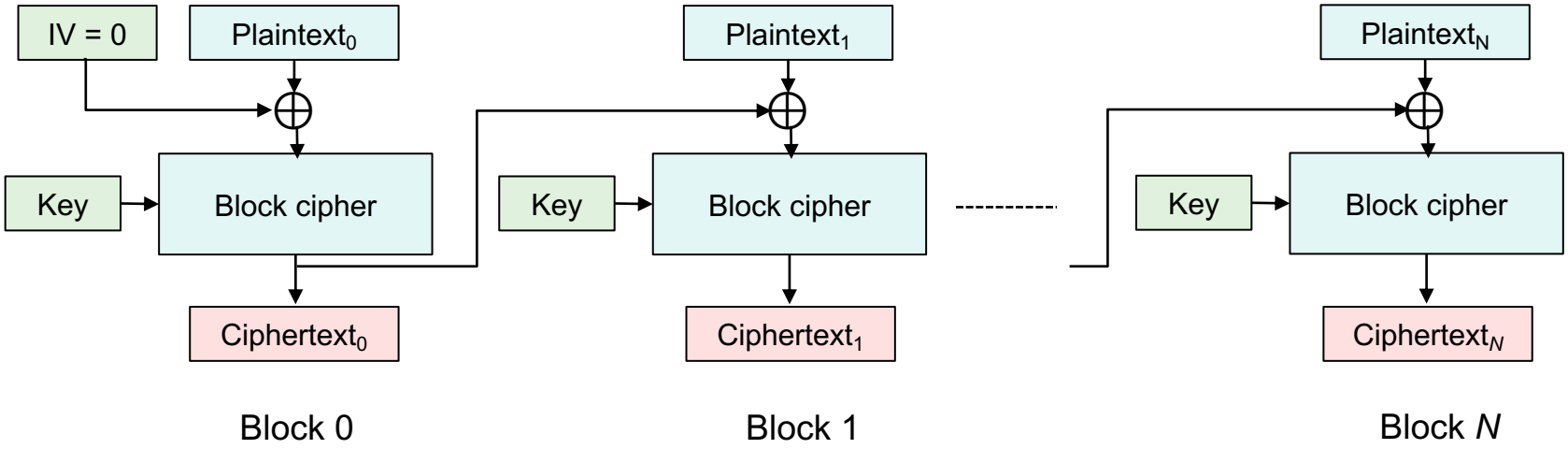
$\oplus$ = XOR, || = concatenation

Basically, incorporate a key into the message before hashing it

See RFC 2104

Block cipher based MAC: CBC-MAC

- Cipher Block Chaining (CBC) ensures that every encrypted block is a function of all previous blocks
- CBC-MAC uses a zero initialization vector



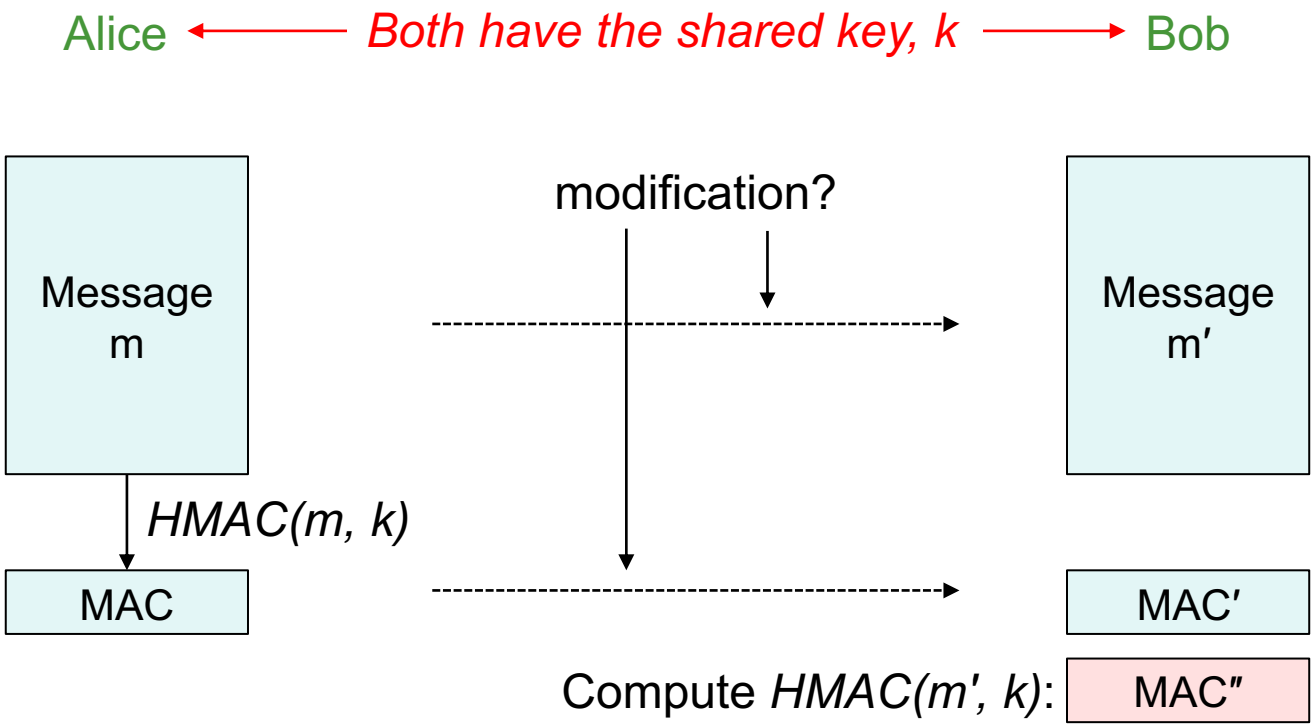
MAC = final ciphertext block – others are discarded

Examples: AES-CBC-MAC, DES-MAC

Don't use the same key for the MAC as for encrypting the message

If an adversary gets one of the keys, she will be unable to create either a valid message or a valid hash

Using a MAC



1. Bob receives the Message m' and a MAC.
2. Knowing the key, k, he generates a MAC for the message:
   MAC″ = HMAC(m′, k)
3. If MAC′ = MAC″, he's convinced that the message has not been modified

Digital Signatures

- MACs rely on a shared key
  - Anyone with the key can modify and re-sign a message

- Digital signature properties
  - Only you can sign a message, but anyone can validate it
  - You cannot cut and paste the signature from one message to another
  - If the message is modified, the signature will be invalid
  - An adversary cannot forge a signature
    - Even after inspecting an arbitrary number of signed messages

Digital Signature Primitives

1. Key generation

   { secret_key, verification_key } := **gen_keys**(key_size)

2. Signing

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

3. Validation

   Isvalid := **verify**(verification_key, message, signature)

We sign *hash(message)* instead of the *message*

- We'd like the signature to be a small, fixed size
- We may not need to hide the contents of the message
- We trust hashes to be collision-free
- We can use a signature in a hash pointer instead of just a hash
  - This will protect the integrity of the entire data structure to which its pointing

Digital Signatures & Public Key Cryptography

- Public key cryptography enables digital signatures
  - *secret_key = private key*
  - *verification_key = public key*

- Alice encrypts a message with her private key

$$S = E_a(M)$$

- Anyone can decrypt it using her private key

$$D_A(S) = D_A(E_a(M)) = M$$

- But nobody but alice can create S

Popular Digital Signature Algorithms

- DSA: Digital Signature Algorithm
  - NIST standard
  - Uses SHA-1 or SHA-2 hash
  - Key pair based on difficulty of computing discrete logarithms

- EdDSA: Edwards-curve Digital Signature Algorithm
- ECDSA: Elliptic Curve Digital Signature Algorithm
  - Variants of DSA that uses elliptic curve cryptography
  - Used in bitcoin

DSA systems combine hashing + encryption into one step

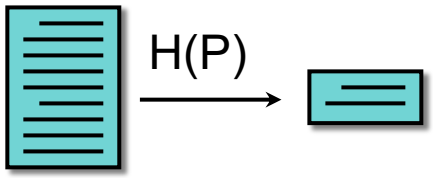
- RSA cryptography
  - $E_{pri\_key}(H(\mathrm{M}))$, $D_{pub\_key}(H(\mathrm{M}))$

Note: not Diffie-Hellman, which is only a key exchange algorithm

Digital signatures

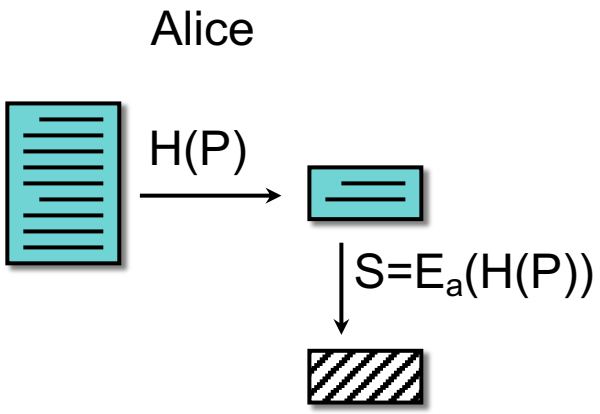
Alice

Bob



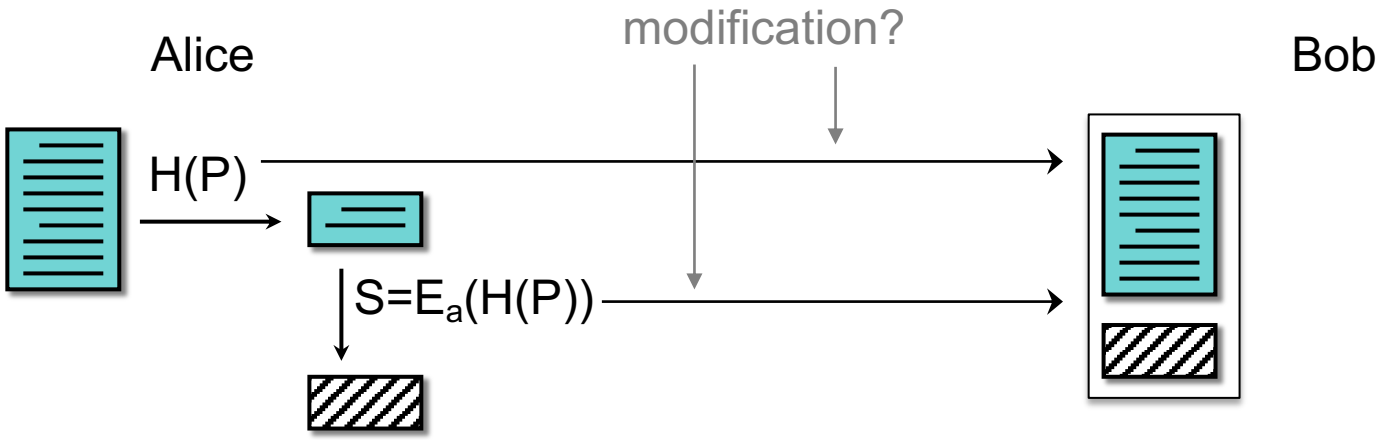
Alice generates a hash of the message

Digital signatures: public key cryptography



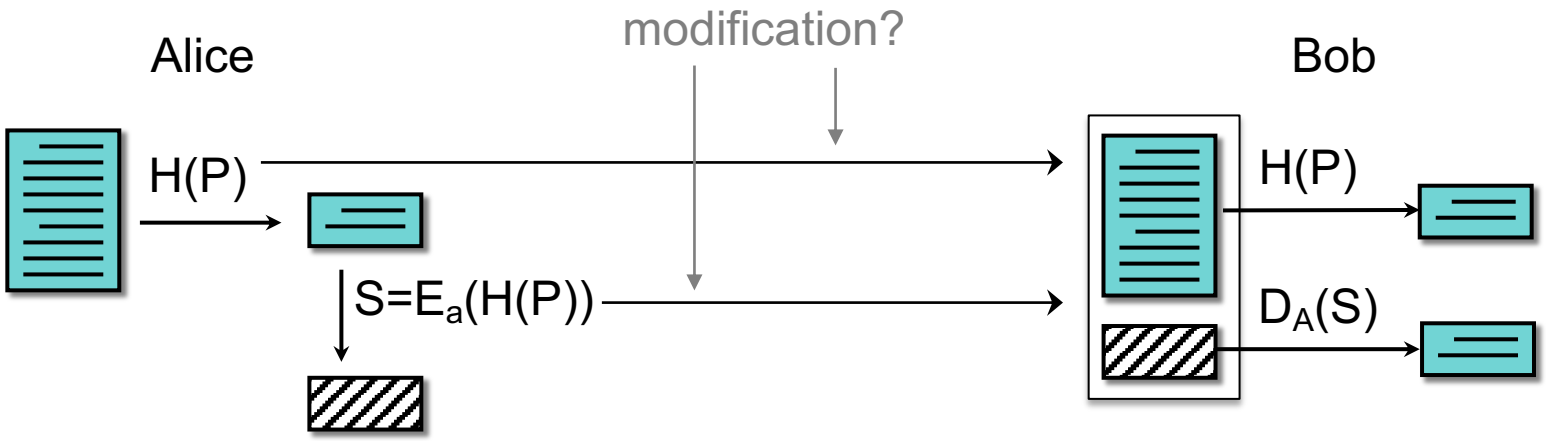
Alice encrypts the hash with her private key
This is her **signature**.

Digital signatures: public key cryptography



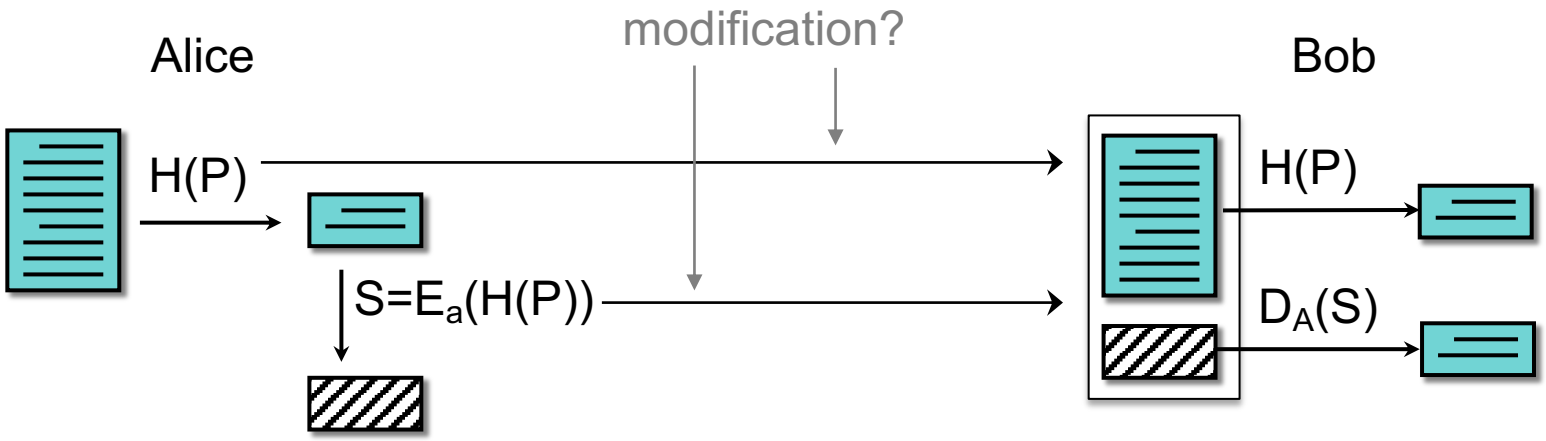
Alice sends Bob the message & the encrypted hash

Digital signatures: public key cryptography



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

Digital signatures: public key cryptography



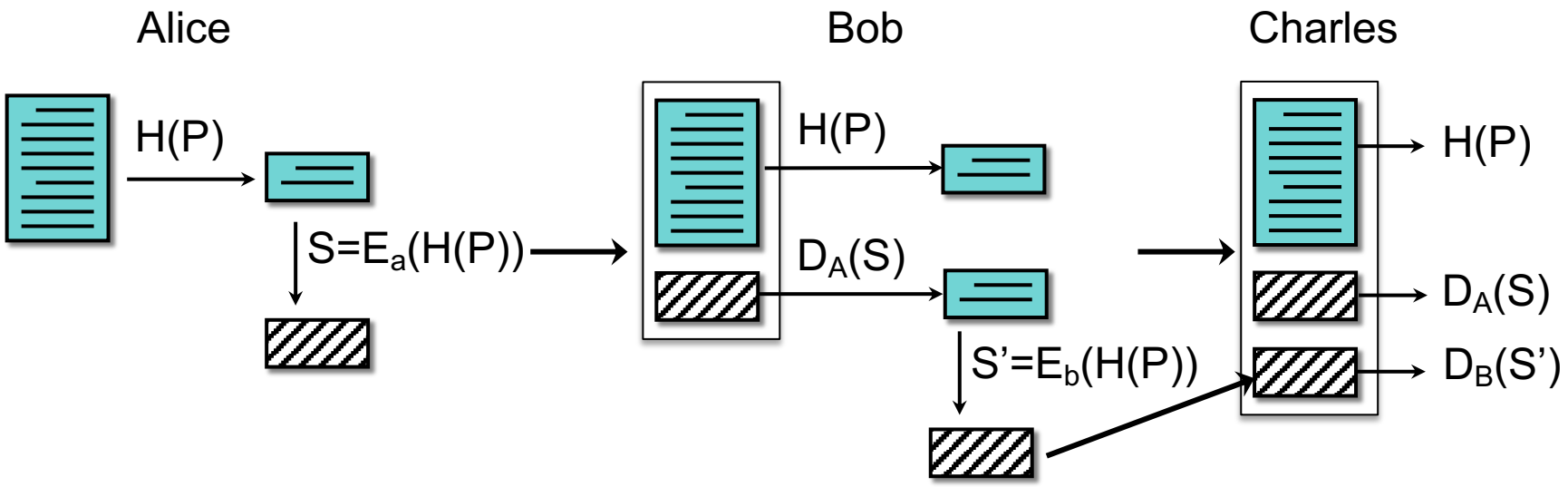
If the hashes match, the signature is valid
– the encrypted hash *must* have been generated by Alice

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: $D_B(S) \stackrel{?}{=} H(P)$

Covert AND authenticated messaging

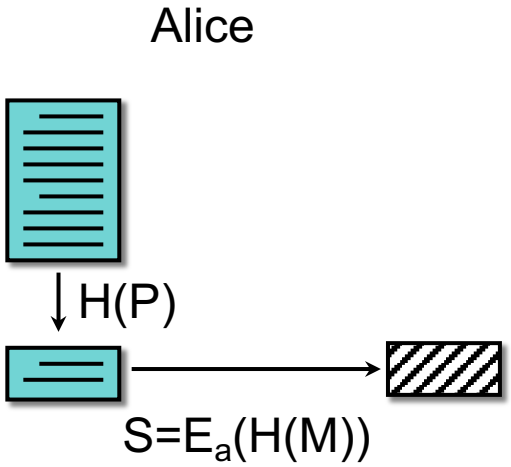
If we want to keep the message secret

- combine encryption with a digital signature

Use a session key:

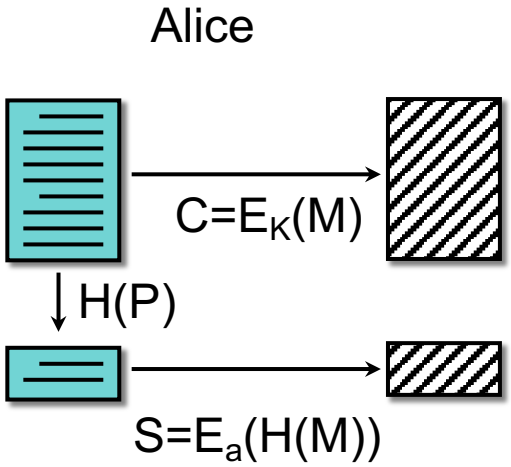
- Pick a random key, *K*, to encrypt the message with a symmetric algorithm
- encrypt *K* with the public key of each recipient
- for signing, encrypt the hash of the message with sender's private key

Covert and authenticated messaging



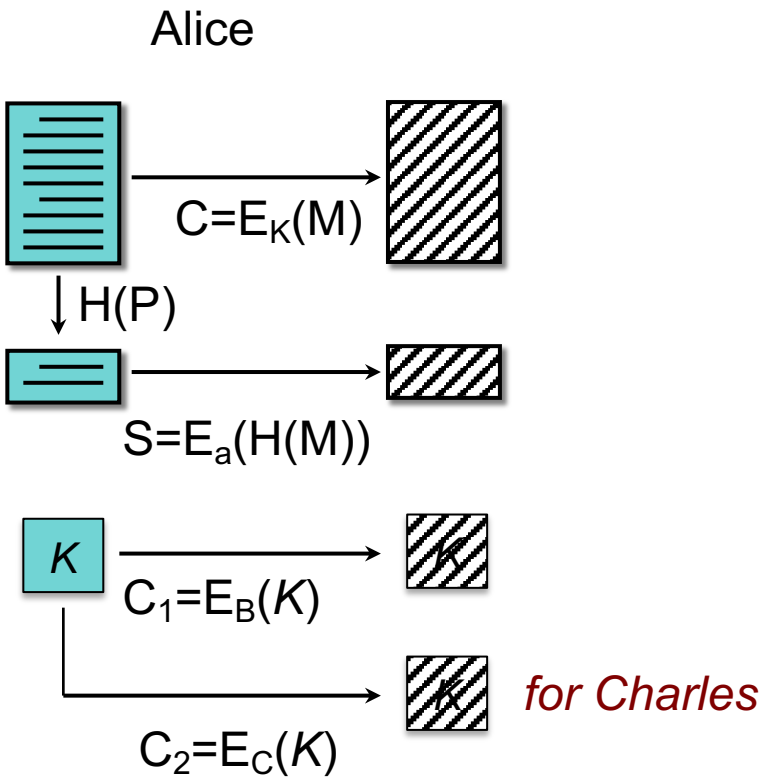
Alice generates a digital signature by encrypting the message with her private key

Covert and authenticated messaging



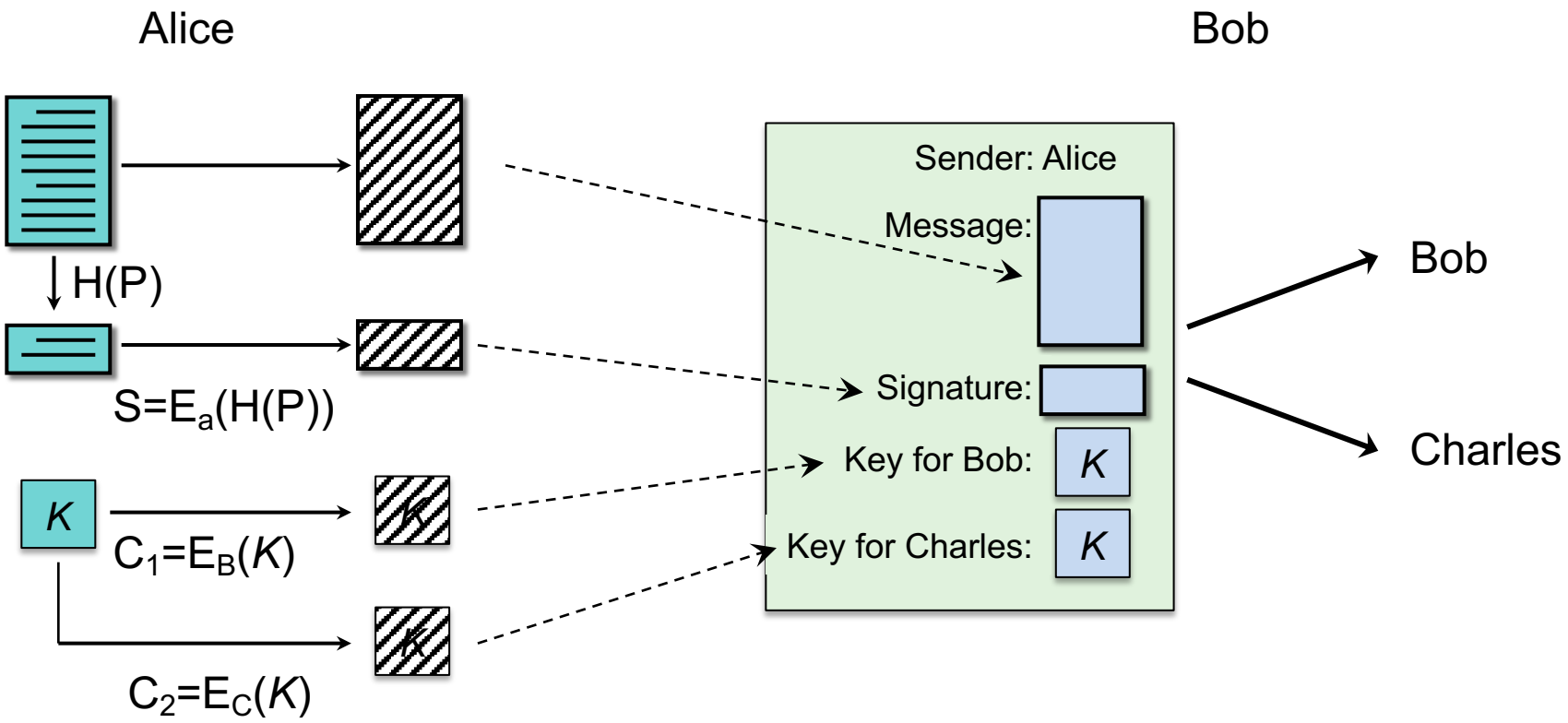
Alice picks a random key, *K*, and encrypts the message *P* with it using a symmetric cipher

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

Note: we do not have forward secrecy by doing this

Public Keys as Identities

A public signature verification key can be treated as an identity

- Only the owner of the corresponding private key will be able to create the signature

- New identities can be created by generating new random {*private, public*} key pairs

Anonymous identity – no identity management

- A user is known by a random-looking public key
- Anybody can create a new identity at any time
- Anybody can create as many identities as they want
- A user can throw away an identity when it is no longer needed
- Example: Bitcoin identity = hash(public key)

Certificates: Identity Binding

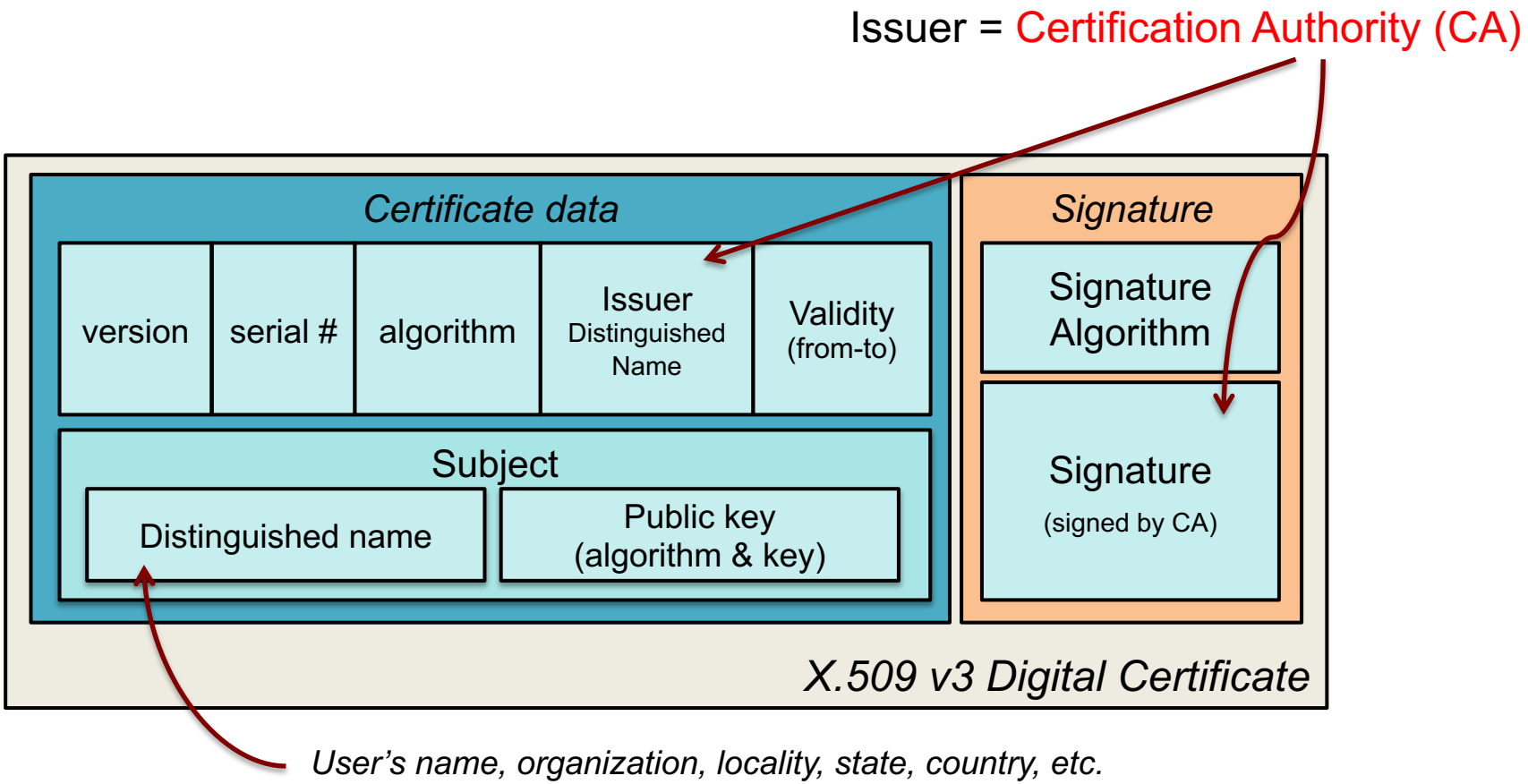
Identity Binding

- How does Alice know Bob's public key is really his?
- Get it from a trusted server?
  - What if the enemy tampers with the server?
  - Or intercepts Alice's query to the server (or the reply)?
  - What set of public keys does the server manage?
  - How do you find it in a trustworthy manner?
- Another option
  - Have a **trusted party** sign Bob's public key
  - Once signed, it is tamper-proof
  - But we need to know it's Bob's public key and who signed it
    - So … sign a data structure that
      1. Identifies Bob
      2. Contains his public key
      3. Identifies who is doing the signing

X.509 Certificates

ISO introduced a set of authentication protocols

X.509: Structure for public key certificates:



X.509 certificates

To validate a certificate

Verify its signature:

1. Hash contents of certificate data
2. Decrypt CA's signature with CA's public key

Obtain CA's public key (certificate) from trusted source

Certificates prevent someone from using a phony public key to masquerade as another person

…if you trust the CA

Certification Authorities (CAs)

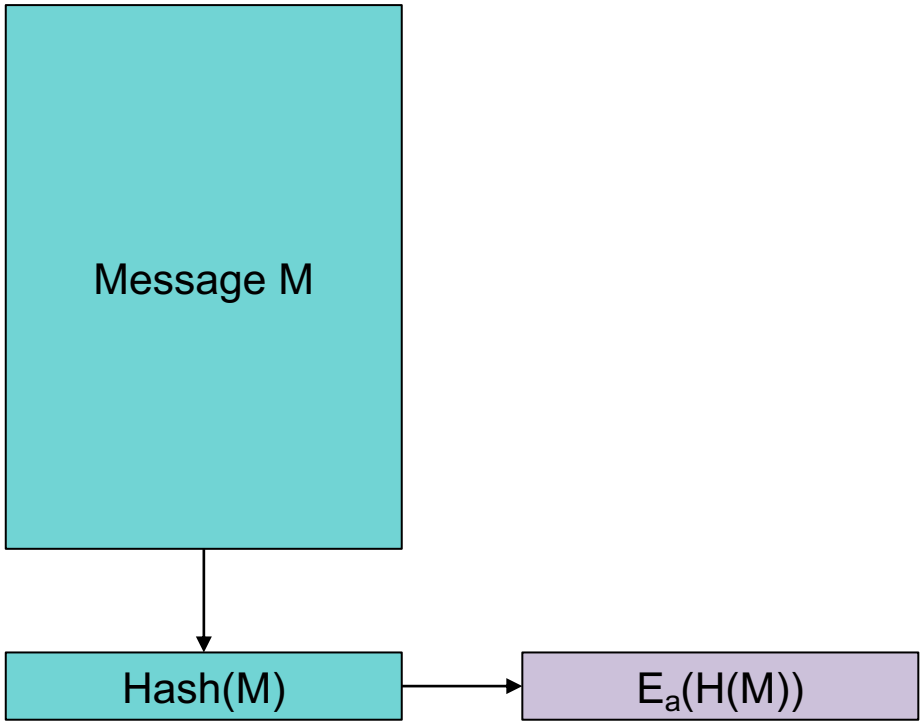
- How do you know the public key of the CA?
  - You can get it from another certificate!
  - This is called **certificate chaining**
  - But trust has to start somewhere: you need a public key you can trust (probably sitting inside a certificate) – this is the **root CA**
    - Apple's keychain is pre-loaded with hundreds of CA certificates
    - Windows stores them in the Certificate Store and makes them accessible via the Microsoft Management Console (mmc)
    - Android stores them in Credential Storage
- Can you trust a CA?
  - Maybe…
    check their reputation and read their *Certification Practice Statement*
  - Even trustworthy ones might get hacked (e.g., VeriSign in 2010)

Key revocation

- Used to invalidate certificates before expiration time
  - Usually because of a compromised key
  - Or policy changes (e.g., someone leaves a company)
- Certificate revocation list (CRL)
  - Lists certificates that are revoked
  - Only certificate issuer can revoke a certificate
- Problems
  - Need to make sure that the entity issuing the revocation is authorized to do this
  - Revocation information may not circulate quickly enough
    - Dependent on dissemination mechanisms, network delays & infrastructure
    - Some systems may not have been coded to process revocations

Code Integrity

Review: signed messages



Encrypt with Alice's private key
= digital signature

We can sign code as well

- Validate integrity of the code
  - If the signature matches, then the code has not been modified
- Enables
  - Distribution from untrusted sources
  - Distribution over untrusted channels
  - Detection of modifications by malware
- Signature = encrypted hash signed by trusted source
  - Does *not* validate the code is good … just where it comes from

Code Integrity: signed software

- Windows 7-10: Microsoft Authenticode
  - SignTool command
  - Hashes stored in system catalog or signed & embedded in the file
  - Microsoft-tested drivers are signed
- macOS
  - codesign command
  - Hashes & certificate chain stored in file
- Also Android & iOS

Code signing: Microsoft Authenticode

A format for signing executable code (dll, exe, cab, ocx, class files)

- Software publisher:
  - Generate a public/private key pair
  - Get a digital certificate: VeriSign class 3 Commercial Software Publisher's certificate
  - Generate a hash of the code to create a fixed-length digest
  - Encrypt the hash with your private key
  - Combine digest & certificate into a Signature Block
  - Embed Signature Block in executable
- Microsoft SmartScreen:
  - Manages reputation based on download history, popularity, anti-virus results
- Recipient:
  - Call *WinVerifyTrust* function to validate:
    - Validate certificate, decrypt digest, compare with hash of downloaded code

Per-page hashing

- Integrity check when program is first loaded
- Per-page signatures – improved performance
  - Check hashes for every page upon loading (demand paging)
- Per-page hashes can be disabled optionally on both Windows and macOS

Windows code integrity checks

- Implemented as a file system driver
  - Works with demand paging from executable
  - Check hashes for every page as the page is loaded
- Hashes stored in system catalog or embedded in file along with X.509 certificate.
- Check integrity of boot process
  - Kernel code must be signed or it won't load
  - Drivers shipped with Windows must be certified or contain a certificate from Microsoft

The End