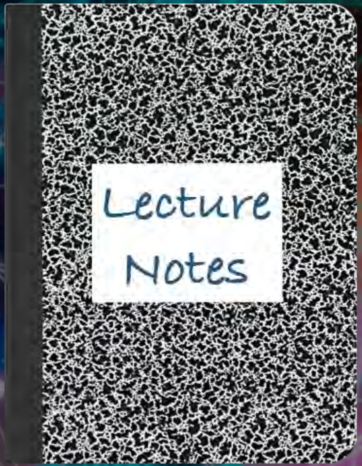
CS 419: Computer Security

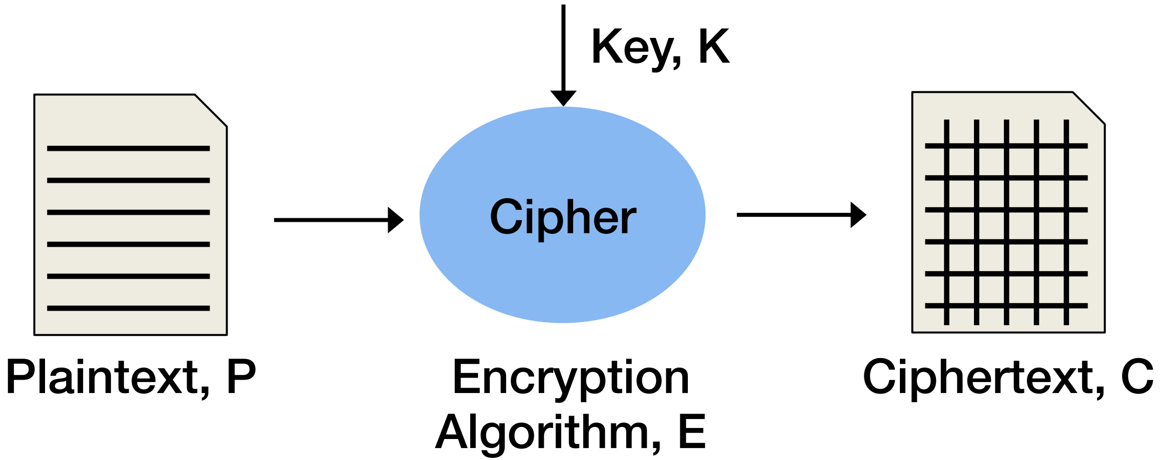
Week 3: Asymmetric Cryptography & Integrity



Paul Krzyzanowski

Key Distribution

Symmetric Cryptography: Terms



$$C = E_K(P)$$

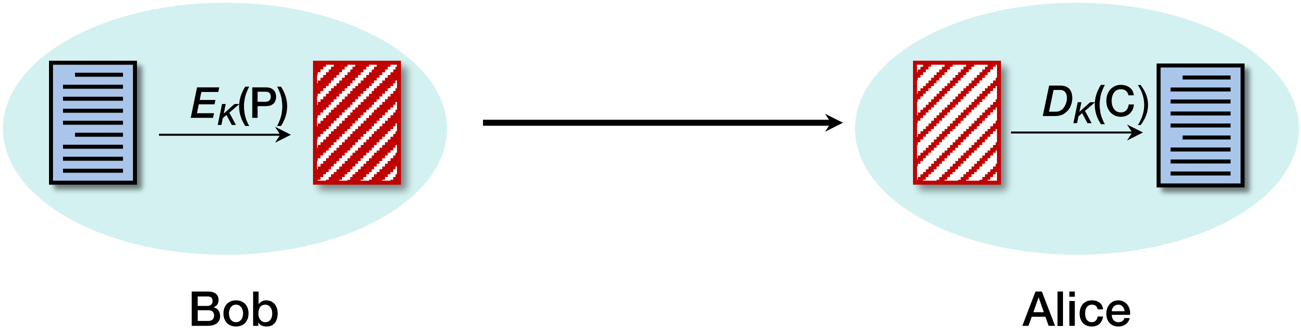
Original content

Encrypted content

The same key is used for encryption and decryption

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 by the adversary
– Users can be impersonated

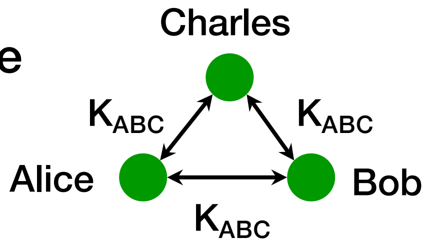
Problems With Keys In Symmetric Cryptography

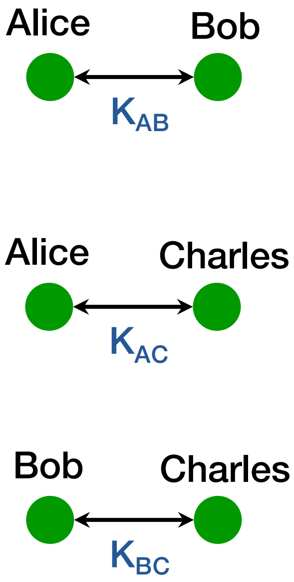
Key Management

- Potentially a lot of keys to track
- Every communicating group of users needs a key

Key Distribution

- How do you communicate with someone you've never met?
- You cannot send them the secret key if the communication line is not secure





Key Distribution

Secure key distribution is the biggest problem with symmetric cryptography

Key Exchange

Trapdoor functions

Trapdoor function

- Easy to compute in one direction
- The 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

Example trapdoor functions

Examples:

Factoring:

$pq = N$	EASY	Basis for RSA
find p,q given N	DIFFICULT	

Discrete Log:

$a^b \bmod c = N$	EASY	Basis for Diffie-Hellman & Elliptic Curve
find b given a, c, N	DIFFICULT	

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

Diffie-Hellman Key Exchange (DHKE)

Key distribution algorithm

- Allows two parties to share a secret key over a non-secure channel
- *Not* public key encryption
- Based on difficulty of computing discrete logarithms in a finite field compared with ease of calculating exponentiation

Allows two parties to negotiate a secret common key over an insecure communication channel without fear of eavesdroppers

Diffie-Hellman Key Exchange (DHKE)

- **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 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) $^{\textit{(Alice's private key)}}$ ***mod p***

Diffie-Hellman 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) $^{\textbf{(Bob's private key)}}$ ***mod p***

Diffie-Hellman 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:

$$K = Y_B^{X_A} \bmod p$$
$$= (\alpha^{X_B} \bmod p)^{X_A} \bmod p$$
$$= \alpha^{X_B X_A} \bmod p$$

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

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

- expanding:

$$K = Y_B^{X_B} \bmod p$$
$$= (\alpha^{X_A} \bmod p)^{X_B} \bmod p$$
$$= \alpha^{X_A X_B} \bmod p$$

K = K′

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

Diffie-Hellman simple example

Assume p=1151, α=57

- Alice's secret key X_A = 300
- Alice's public key Y_A = 57^{300} mod p = 282
- Alice computes

$$K = Y_B^{X_A} \bmod p = 1046^{300} \bmod p$$

$$K = 105$$

- Bob's secret key X_B = 25
- Bob's public key Y_B = 57^{25} mod p = 1046
- Bob computes

$$K = Y_A^{X_B} \bmod p = 282^{25} \bmod p$$

$$K = 105$$

Given p=1151, α=57, Y_A=282, Y_B=1046, you cannot get 105

Public Key Cryptography

Public-key cryptography

Two related keys:

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

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

K_1 is a **public** key

K_2 is a **private** key

Examples:

RSA,
Elliptic curve algorithms (ECC),
DSS (digital signature standard)

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 (~600 digits) prime numbers

RSA algorithm: key generation

You don't have to know this!

1. Choose two random large prime numbers p, q

2. Compute the product $n = pq$ and $\phi(n) = (p - 1)(q - 1)$

   n will be presented with the public & private keys.
   Length(n) is the **key length**

3. Choose the **public exponent**, e, such that:
   $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$
   Choose a value where e and $(p - 1)(q - 1)$ are relatively prime

4. Compute the **secret exponent**, d such that:

   $ed = 1 \bmod \phi(n)$

   $d = e^{-1} \bmod ((p - 1)(q - 1))$

5. **Public key** = (e, n)
   Private key = (d, n)
   Discard p, q, $\phi(n)$

An example with small numbers

3, 11

(3-1) x (11-1) = 20

Choose e=7

Find d: $7d$ = 1mod 20
7 x 3 = 21 ≡ 1 mod 20
d = 3

Pub key = (3, 33)
Pri key = (7, 33)

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

RSA Security

The security rests on the difficulty of factoring a large integer

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

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

If you know the public key (3, 33), can you derive the private key?

If you can factor 33 = 3 x 11 ⇒ ϕ(n) = 2 x 10 = 20
Find *d*, such that *3d* = 1 mod 20

RSA Security

Since the security of RSA rests on not knowing an efficient way to factor $n = pq$, we have to make it difficult to find factors via an exhaustive search

Common key lengths are 1024, 2048, 3072, 4096, 8129, 16384 bits

Example: a 2048-bit modulus (*n*) and secret exponent (*d*):

n =
0xa709e2f84ac0e21eb0caa018cf7f697f774e96f8115fc2359e9cf60b1dd8d4048d974cdf8422bef6be3c162b04b916f7ea2133f0
e3e4e0eee164859bd9c1e0ef0357c142f4f633b4add4aab86c8f8895cd33fbf4e024d9a3ad6be6267570b4a72d2c34354e0139e
74ada665a16a2611490debb8e131a6cffc7ef25e74240803dd71a4fcd953c988111b0aa9bbc4c57024fc5e8c4462ad9049c7f1a
bed859c63455fa6d58b5cc34a3d3206ff74b9e96c336dbacf0cdd18ed0c66796ce00ab07f36b24cbe3342523fd8215a8e77f89e
86a08db911f237459388dee642dae7cb2644a03e71ed5c6fa5077cf4090fafa556048b536b879a88f628698f0c7b420c4b7

d =
0x10f22727e552e2c86ba06d7ed6de28326eef76d0128327cd64c5566368fdc1a9f740ad8dd221419a5550fc8c14b33fa9f058b
9fa4044775aaf5c66a999a7da4d4fdb8141c25ee5294ea6a54331d045f25c9a5f7f47960acbae20fa27ab5669c80eaf235a1d0b1
c22b8d750a191c0f0c9b3561aaa4934847101343920d84f24334d3af05fede0e355911c7db8b8de3bf435907c855c3d7eeede4f
148df830b43dd360b43692239ac10e566f138fb4b30fb1af0603cfcf0cd8adf4349a0d0b93bf89804e7c2e24ca7615e51af66dccf
db71a1204e2107abbee4259f2cac917fafe3b029baf13c4dde7923c47ee3fec248390203a384b9eb773c154540c5196bce1

The size of each value is 10^{617}

Weak RSA Public Keys

March 14, 2022

- **Older software generated RSA keys that can be broken instantly with commodity hardware**
- **SafeZone library doesn't randomize the prime numbers well**
  - Used to generate RSA keys
  - After selecting one prime #, the second one is in close proximity to the first
- **Keys generated with primes that are too close together can be broken with Fermat's factorization method, described in 1643**

ars TECHNICA SUBSCRIBE SIGN IN

BREAKING KEYS —

Researcher uses 379-year-old algorithm to crack crypto keys found in the wild

It takes only a second to crack the handful of weak keys. Are there more out there?

DAN GOODIN - 3/14/2022, 5:31 PM

Enlarge

Cryptographic keys generated with older software now owned by technology company Rambus are weak enough to be broken instantly using commodity hardware, a researcher reported on Monday. This revelation is part of an investigation that also uncovered a handful of weak keys in the wild.

The software comes from a basic version of the SafeZone Crypto Libraries, which were developed by a company called Inside Secure and acquired by Rambus as part of its 2019 acquisition of Verimatrix, a Rambus representative said. That version was deprecated prior to the acquisition and is distinct from a FIPS-certified version that the company now sells under the Rambus FIPS Security Toolkit brand.

https://arstechnica.com/information-technology/2022/03/researcher-uses-600-year-old-algorithm-to-crack-crypto-keys-found-in-the-wild/

Weak RSA Public Keys

You don't have to know this!

- **The product of two large primes can be written as**
  $N = (a - b)(a + b)$
  – where a is the middle between the two primes
  – b is the distance from the middle to each of the primes
- **If the primes are close, then a is close to $\sqrt{N}$**
- **Attack: guess a by starting from $\sqrt{N}$ and then incrementing the guess**
  – Calculate $b^2 = a^2 - N$
  – If the result is a square, then we guessed correctly
  – Calculate the factors p, q as $p=a+b$, $q=a-b$

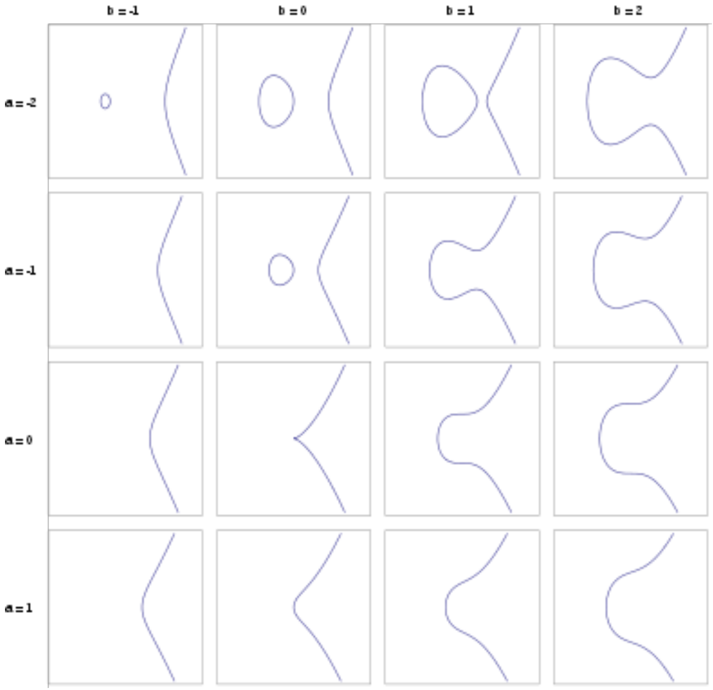
Elliptic Curve Cryptography

You don't have to know this!

Key Generation

Using discrete numbers, pick

- A prime number as a maximum (modulus)
- A curve equation in the family $y^2 = x^3 + ax + b$
- A public base point on the curve, *G*
- A random private key, *d*
- Public key is computed from the private key, the base point, and the curve: d x G



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

Elliptic Curve Cryptography

You don't have to know this!

Encryption

Using a public key Q_A and a message *m*

– G is the base point on the elliptic curve

– Map the message *m* to a point on the curve using a specified mapping *function*

- *M* is the resulting point

– Choose a random integer *k*, where $1 \leq k \leq n-1$
(where *n* is the order of the elliptic curve = # of discrete points)

– Compute two points

- $C_1 = k \times G$ – this will be one part of the ciphertext
- $C_2 = M + k \times Q_A$ – this is the encrypted message

– Ciphertext = $(C_1, C_2) = (k \times G, M + k \times Q_A)$

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 a widely used public key cryptosystem (but fading)**
  - Mostly due to inertia & widespread implementations – it had a 27-year head start
  - Trusted, well-tested deployments
  - Trust in the algorithm
    (there was initial skepticism over the choice of curves and trust in the NIST;
    also, the NSA tried to push an insecure random number generator)
  - Simpler implementation

- **ECC offers higher security with fewer bits than RSA**
  - ECC is faster for key generation & encryption
    - The private key is any random number within a certain range (e.g., a 512-bit integer)
    - Encryption is about 10x faster than RSA
  - 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

Comparable complexity:

- 3072-bit RSA = 256-bit elliptic curve = 128-bit symmetric cipher
- 15360-bit RSA = 512-bit elliptic curve = 256-bit symmetric cipher

Recommendations for long-term security

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

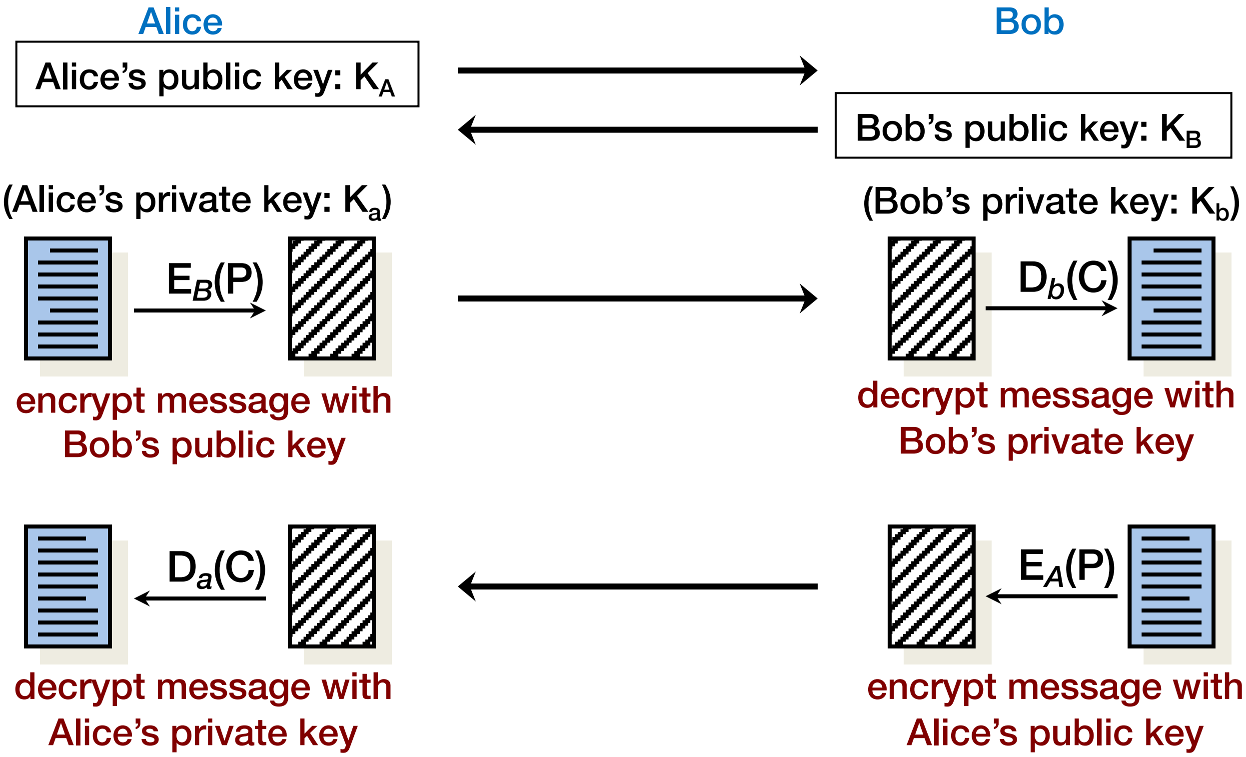
AES: 256-bit keys **RSA: 15,360-bit keys** **ECC: 512 bit-keys**

Communication with public key algorithms

Different keys for encrypting and decrypting

- No need to worry about key distribution

Communication with public key algorithms



Public key algorithms are not used for communication

Calculations are very expensive relative to symmetric algorithms

Speeds on an M1 Mac Mini*:

Algorithm	bytes/sec
AES-128-CBC	1,538,484,000
AES-256-CBC	1,097,739,000
RSA-2048 encrypt	56,225,434
RSA-2048 decrypt	2,695,014
RSA-4096 encrypt	9,805,875
RSA-4096 decrypt	143,053

AES: ~7600x faster to decrypt; 100x faster to encrypt than RSA

*openssl command benchmarks

Public key algorithms are not used for communication

- **Vulnerability to known plaintext attacks (or guessing)**
  - Content must be broken into smaller blocks since each block is treated like a number. An attacker can encrypt a wide set of predicted content with the recipient's public key and then look for matches in the ciphertext.
  - If I send "Yes" to you, I need to encrypt it with your public key. The attacker can encrypt "Yes", "No", and any other expected content with that same public key and see what matches the content I send. If there's a predicted chunk of a message, the attacker can spot it.
- **Some algebraic relationships may be preserved**
  - Some algebraic relationships that exist between plaintext content may exist with public key algorithms. This can provide an attacker with insights on the relationship between content.
  - Example: RSA is $c = m^e \bmod n$. If the plaintext block is 1 then the ciphertext will be 1; if plaintext=0 then ciphertext=0.
- **ECC ciphertext can be 2x as long as the plaintext**
- **ECC is faster than RSA and uses shorter keys but is still much slower than symmetric**
  - ECC public key generation is efficient compared with RSA but is still much slower than symmetric algorithms and not considered secure for bulk data
    - Requires math (point multiplication): see https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication

See: https://andrea.corbellini.name/2023/01/02/ec-encryption/

Hybrid Cryptosystems

Hybrid Cryptosystems

- **Session key**: randomly-generated symmetric key for one communication session

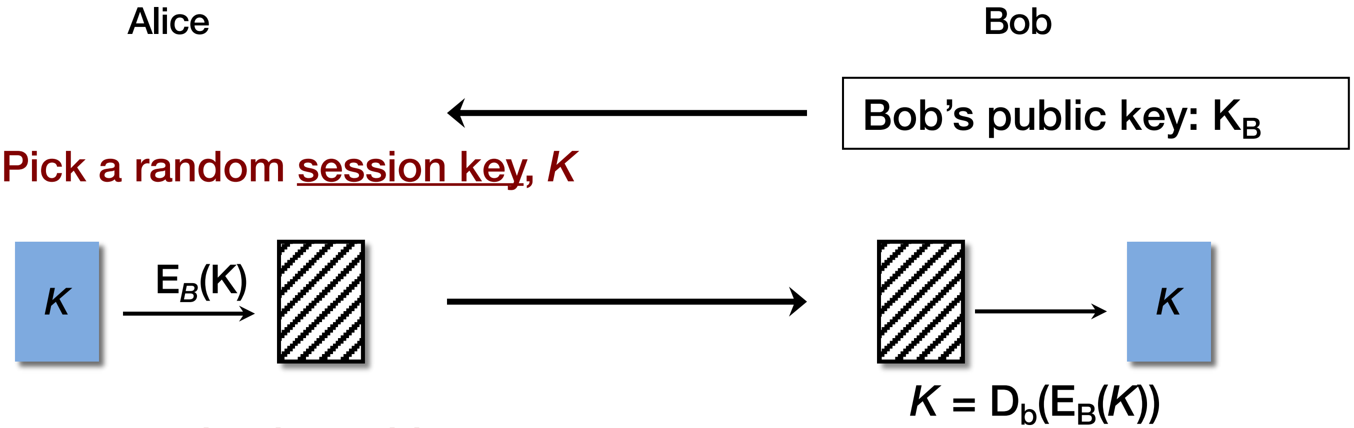
- Use a **public key algorithm** to send the session key

- Use a **symmetric algorithm** to encrypt data with the session key

Public key algorithms are never used to encrypt messages

- MUCH slower; vulnerable to *chosen-plaintext* and *algebraic attacks*

Communication with a hybrid cryptosystem



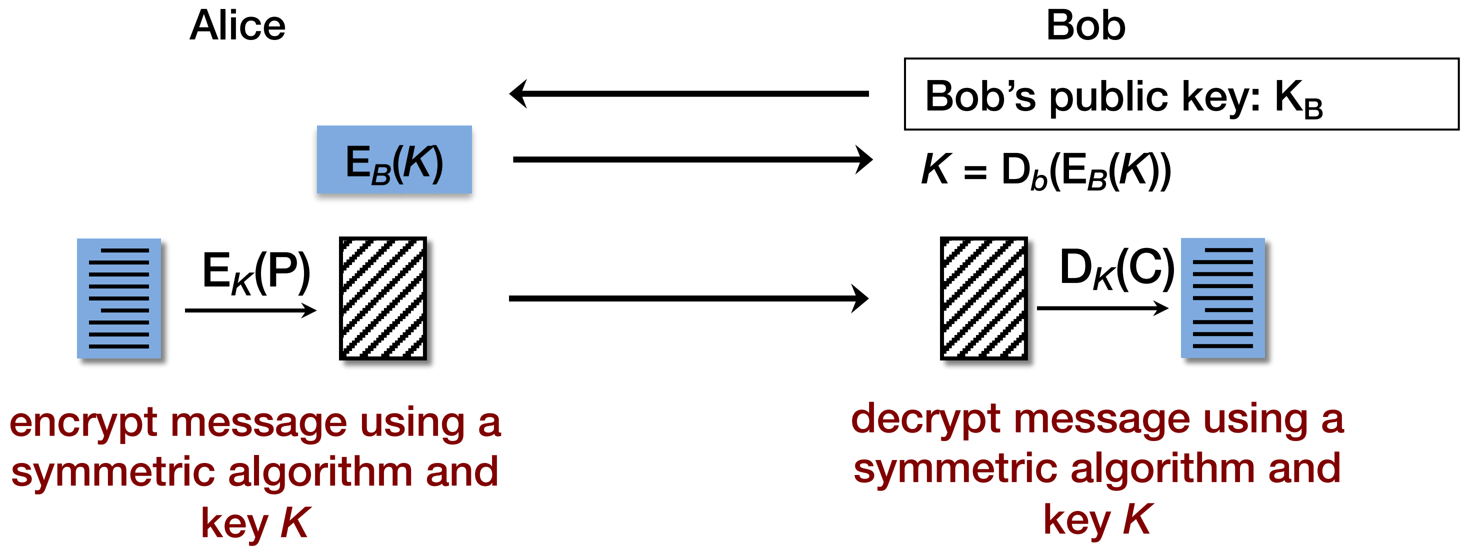
Pick a random session key, *K*

encrypt session key with Bob's public key

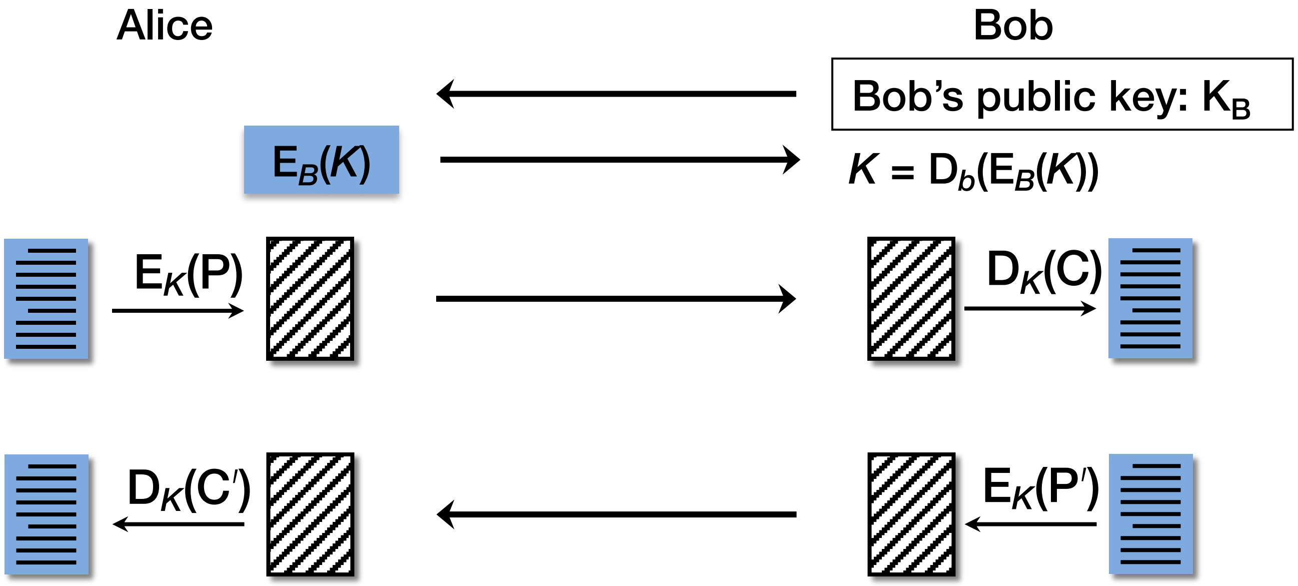
Bob decrypts *K* with his private key

Now Bob knows the secret session key, K

Communication with a hybrid cryptosystem



Communication with a hybrid cryptosystem

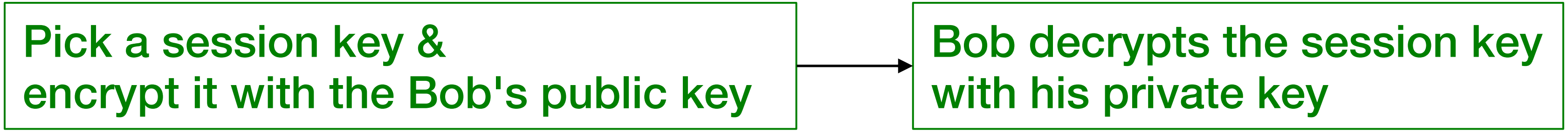


decrypt message using a symmetric algorithm and key *K*

encrypt message using a symmetric algorithm and key *K*

Forward Secrecy

Private keys need to be protected



Suppose an attacker steals Bob’s private key

- Future messages can be compromised
- The attacker can also 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

Long-term keys: cryptographic keys that remain unchanged for an extended period, typically used for identity verification, authentication, or key exchange.

Unlike **session keys**, which are temporary, long-term keys persist across multiple sessions and interactions

Achieving Forward Secrecy

Use ephemeral keys for key exchange + session keys for communication

Diffie-Hellman key exchange is commonly used for key exchange

- Generate a set of keys (Diffie-Hellman parameters) at the start of the 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, key generation in Diffie-Hellman is *extremely* efficient

Ephemeral = very-short term, single-use keys that are thrown away immediately

Client & server will generate new Diffie-Hellman parameters at the start of each session

Session = keys for encrypting data during a session; 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.

Communication with a hybrid cryptosystem (DHKE)

Alice

Create a random Diffie-Hellman key pair: X_A, Y_A

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

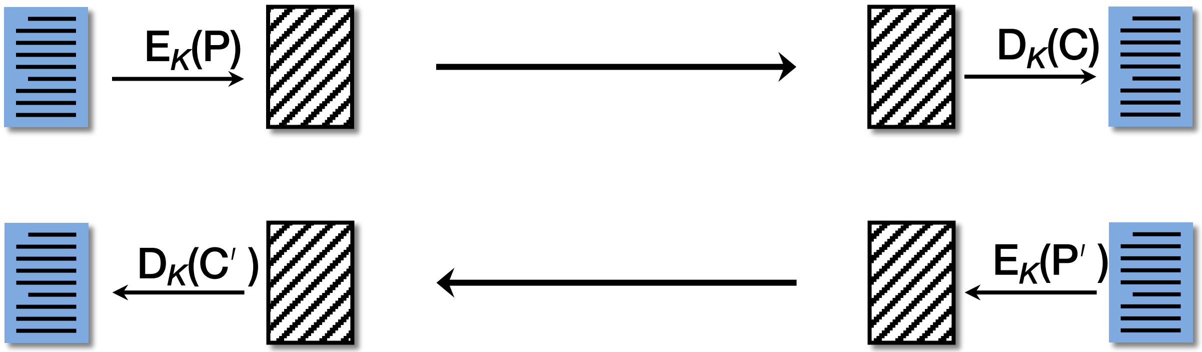
Alice's D-H public key: Y_A

Bob

Create a random Diffie-Hellman key pair: X_B, Y_B

Bob's D-H public key: Y_B

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



decrypt message using a symmetric algorithm and key K

encrypt message using a symmetric algorithm and key K

Cryptographic systems: summary

- **Symmetric ciphers**
  - Based on SP-networks (usually) = 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 + public key algorithm for key exchange
  - Use a symmetric key algorithm to encrypt traffic back & forth
  - **Forward secrecy**: establish session key via ephemeral keys
- **Key exchange algorithms (more to come later)**
  - Diffie-Hellman
  - Public key

  } ***Enables secure communication without knowledge of a shared secret***
- **Perfect secrecy**
  - Ephemeral keys + Session key

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 & Cryptography

Once (if) useful quantum computers can be built, they will be able to:

- 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

Not all is bad

Symmetric cryptography is largely immune to attacks

Some optimizations are predicted (Grover's algorithm):

Crack a symmetric cipher in time proportional to the square root of the key space size: $2^{n/2}$ vs. 2^n

The recommendation is to use 256-bit AES to be safe

Quantum-proofing cryptography

Quantum computing is not faster at everything

Only four types of problems are currently identified where quantum computing offers an advantage

Researchers have been developing algorithms that are be made more efficient with quantum computing

31108953	1190018662
104910828	2598220447
3027417464	3006531459
2376520867	804531264
2430217482	1122428373

Example: Add 3 out of a set of 10 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**

Which 3 numbers add up to 5656746864?

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

NSA Releases First 3 Post-Quantum Encryption Standards

2016: NSA called for a migration to "post-quantum cryptographic algorithms"

2022: Announced four algorithms slated for standardization:

CRYSTALS-Kyber, CRYSTALS-Dilithium, Sphincs+ and FALCON

August 13, 2024: Releases first three final standards

FIPS 203 - ML-KEM
(Module-Lattice-Based Key-Encapsulation Mechanism, based on CRYSTALS-Kyber algorithm)

FIPS 204 – ML-DSA
(Module-Lattice-Based Digital Signature Algorithm, based on the CRYSTALS-Dilithium algorithm)

FIPS 205 – SLH-DSA
(Stateless Hash-Based Digital Signature Algorithm) – intended as a backup to ML-DSA

(draft) FIPS 206 – FN-DSA
(Fast-Fourier Transform over NTRU-Lattice-Based Digital Signature Algorithm)

Message Integrity

John 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

Example of a one-way function: middle squares

Example with a 20-digit number

A = 18932442986094014771

A^2 = 358437397421700454779607531189166182441

Middle square, B = 42170045477960753118

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

Properties

> **Also called *digests* or *fingerprints***

- 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***
  - This creates an ***avalanche*** effect: the smallest change in input should affect roughly 50% of output bits
- **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

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
    - Used in the NIST Digital Signature Standard (DSS)
    - Produces 160-bit hash values
    - Chosen prefix collision attacks were demonstrated in May 2019
- **Successors**
  - **SHA-2** (2001) – *SHA-224, SHA-256, SHA-384, SHA-512*
    - 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

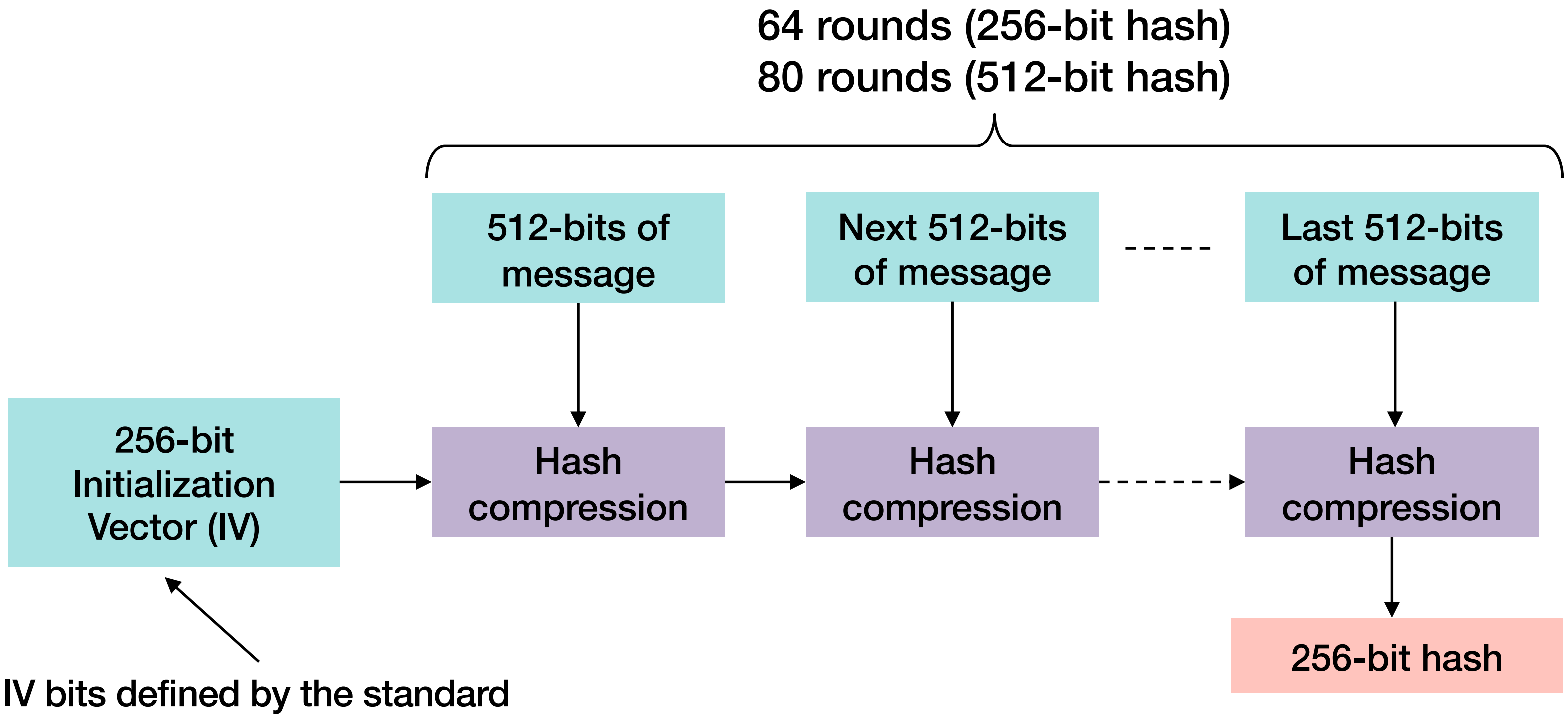
- **Prepare the message**
  - Append the bit 1 to the message
  - Pad message with 0 bits so its length = 448 mod 512
  - Append length of message as a 64-bit big endian integer
- **Use an Initialization Vector (IV) = 5-word (160-bit) buffer:**

```
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 ORs, XORs, ANDs, shifts, and adds
  - 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 (& formerly 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: still used as a 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
- Variations: SHA-224, SHA-256, SHA-384, SHA-512
- Linux passwords use SHA-512
- Bitcoin uses SHA-256

Believed to be secure

SHA-3

Believed to be secure
- 256 & 512 bit

Believed to be secure

bcrypt
- Blowfish cipher used for *bcrypt* password hashing in OpenBSD since 1997
- Phased out in 2023: *scrypt* and *Argon2* are replacements

Designed to be slow!

3DES
- Linux passwords used to use this

Creating hashes via the openssl command

MD5 hash

```
echo 'hello, world!'| openssl dgst -md5
MD5(stdin)= 910c8bc73110b0cd1bc5d2bcae782511
```

SHA-1 hash

```
echo 'hello, world!'| openssl dgst -sha1
SHA1(stdin)= e91ba0972b9055187fa2efa8b5c156f487a8293a
```

256-bit SHA-2 hash

```
echo "hello, world!" | openssl dgst -sha2-256
SHA2-256(stdin)= 4dca0fd5f424a31b03ab807cbae77eb32bf2d089eed1cee154b3afed458de0dc
```

256-bit SHA-3 hash

```
echo "hello, world!" | openssl dgst -sha3-256
SHA3-256(stdin)= 5208fd28810f11b7781a86289fb9121ccc754a5bd8260bcfa539163890092c7e
```

512-bit SHA-3 hash

```
echo "hello, world!" | openssl dgst -sha3-512
SHA3-512(stdin)=
8fc33b84ff22559082893fdc73f6877e590eb67533441fe5e48cd6d8a11aaf8d6270f82ef437c2c758000d65b09b4511
6b9c0eb3f3162149b13ca98c8cc8c90f
```

Hash Collisions

Hashes are *collision resistant*, but collisions can occur

Pigeonhole principle


wikipedia

- 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**

  E.g.: the probability that that someone has the same birthday as Alice
  vs. the probability of two people in a room having the same birthday

- **The 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 $1{:}2.9 \times 10^{8}$

Data Integrity:
Message Authentication Codes and Digital Signatures

Data 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 are convinced that the messages are different**

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

MACs (also called a Keyed Hash)

We rely on hashes to assert the integrity of messages

But 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)

Two forms: hash-based & block cipher-based

HMAC: Hash-based MAC (RFC 2104, FPIS 198-1)

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 – the double hashing is in place to make any collisions more difficult to exploit
- $\oplus$ = XOR, $\|$ = concatenation

Note the extra hash.
The simple form of an HMAC would simply be hash(m, k)
The HMAC standard devised this to strengthen the HMAC against weaker hash functions.

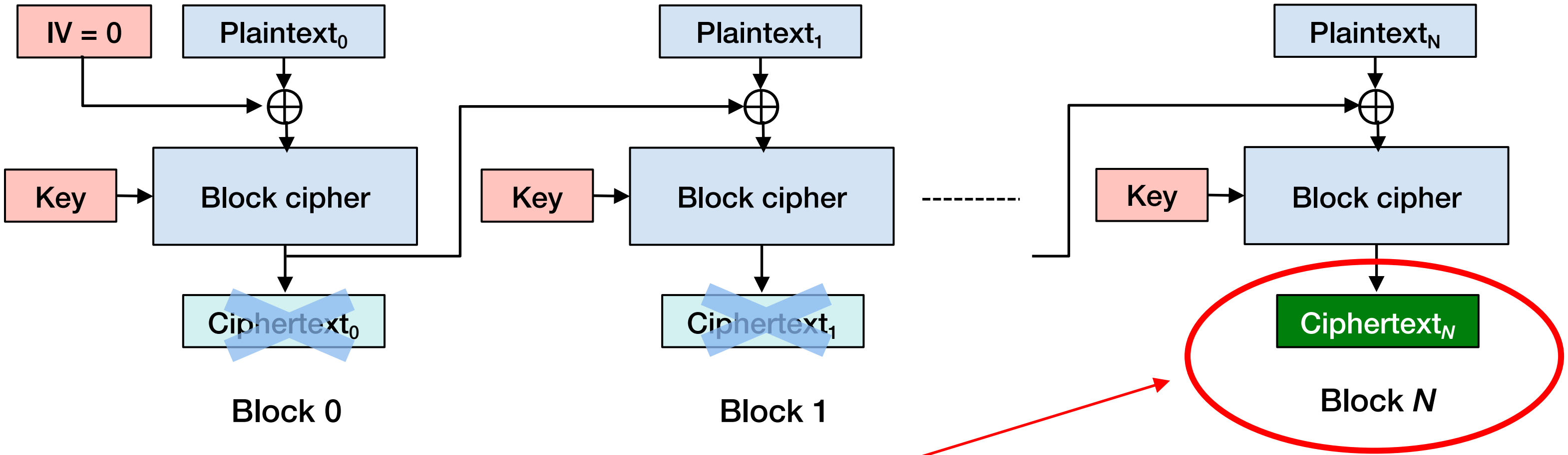
Basically, incorporate a key into the message before hashing it

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 an initialization vector = 0



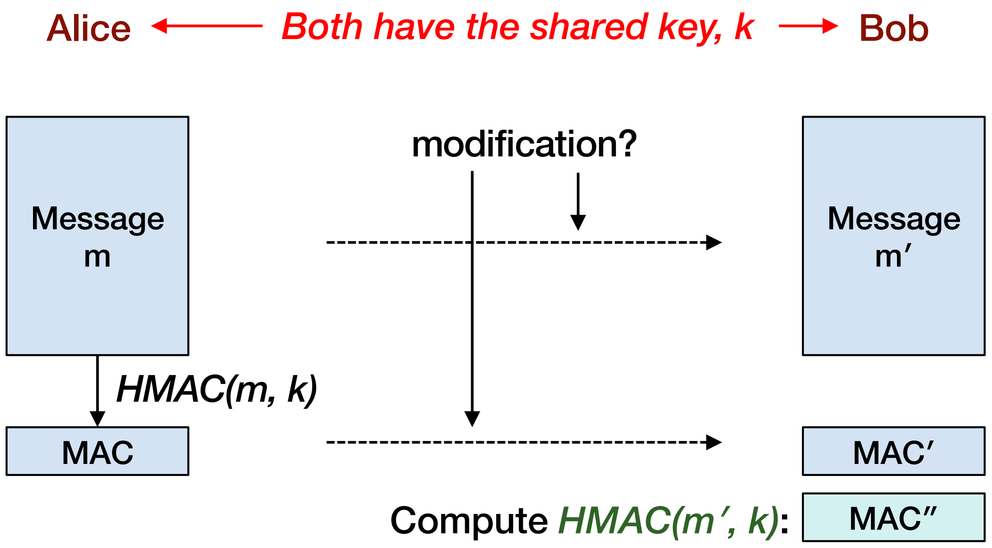
MAC = final ciphertext block – others are discarded

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**

   is_valid := **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

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 public key**

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

- **Nobody but Alice can create S**

Popular Digital Signature Algorithms

Digital Signature Algorithms combine hashing + encryption into one step

signature: $S := E_{pri\_key}(H(M))$

verification = $H(M) \stackrel{?}{=} D_{pub\_key}(S)$

- **DSA: Digital Signature Algorithm**
  - NIST standard – Uses SHA-1 or SHA-2 hash
  - Key pair based on difficulty of computing discrete logarithms
- **ECDSA: Elliptic Curve Digital Signature Algorithm**
  - Variants of DSA that uses elliptic curve cryptography
  - Used in bitcoin
- **EdDSA: Edwards-curve Digital Signature Algorithm**
  - Slightly faster than ECDSA

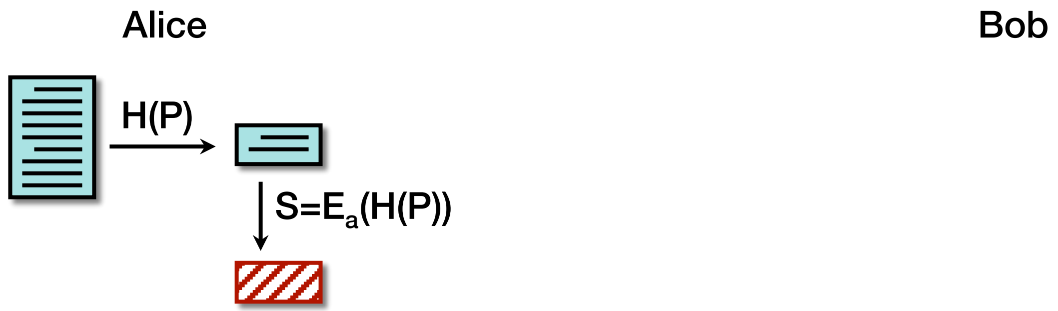
Some signature algorithms implement public-private keys and trapdoor functions but do not perform encryption – the verification function can tell if the hash was modified or signed incorrectly but cannot decrypt the hash.

Conceptually, you can still think of a signature as a hash encrypted with a private key.

Digital signatures

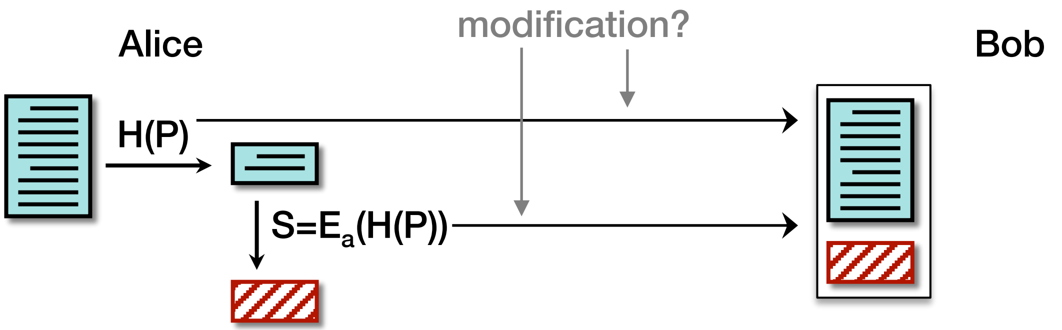
Alice generates a hash of the message, H(P)

Digital signatures: public key cryptography



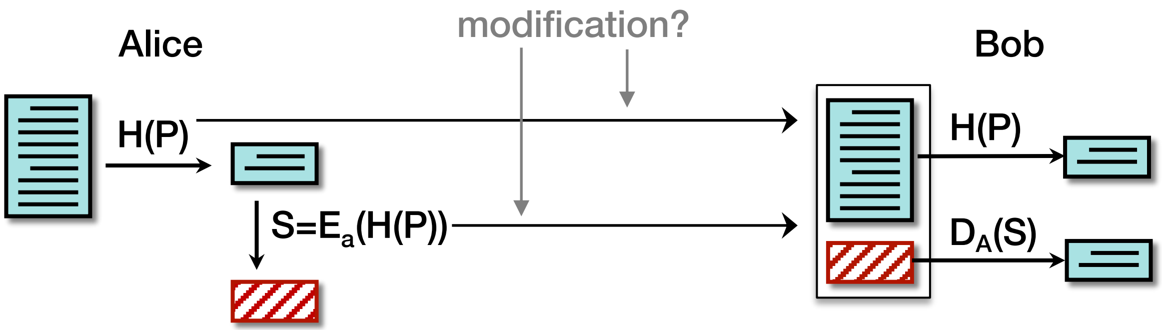
Alice encrypts the hash with her private key
This is her signature.

Using Digital Signatures



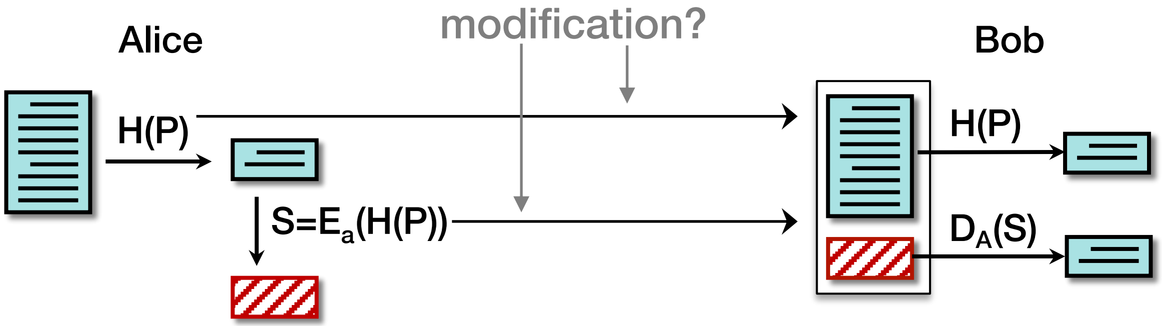
Alice sends Bob the message & the encrypted hash

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

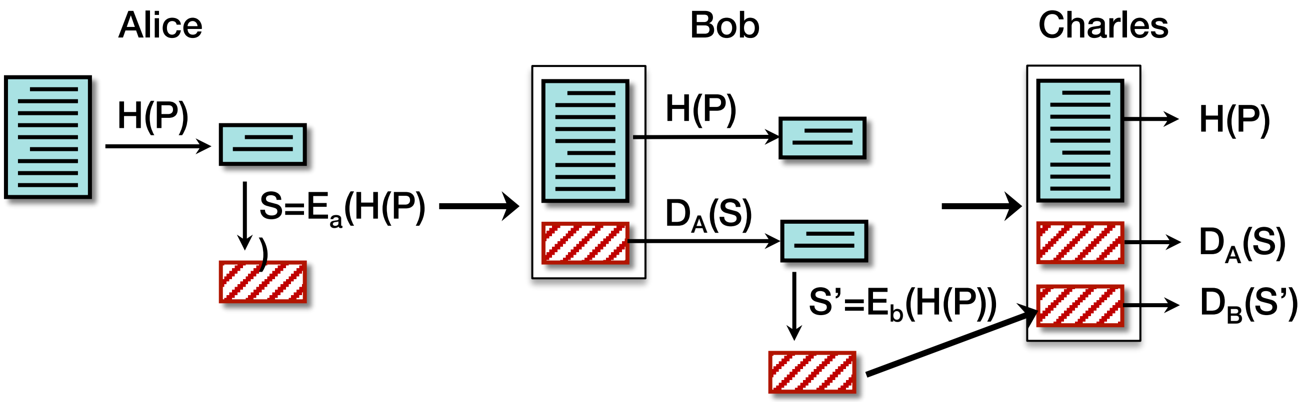


If the hashes match, the signature is valid
⇒ the encrypted hash *must* have been generated by Alice

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

Covert AND authenticated messaging

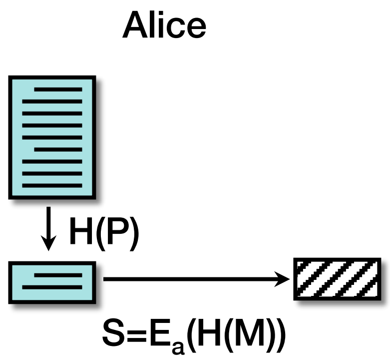
If we want to keep the message secret

- combine encryption with a digital signature

Use a session key:

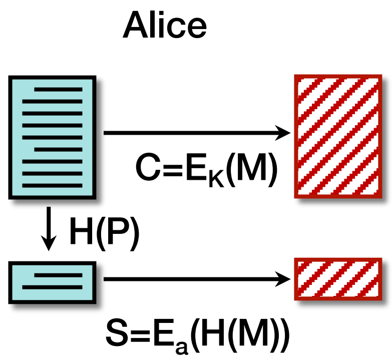
- Pick a random key, *K*, to encrypt the message with a symmetric algorithm
- Encrypt *K* with the public key of each recipient
- For signing, encrypt the hash of the message with sender's private key

Covert and authenticated messaging



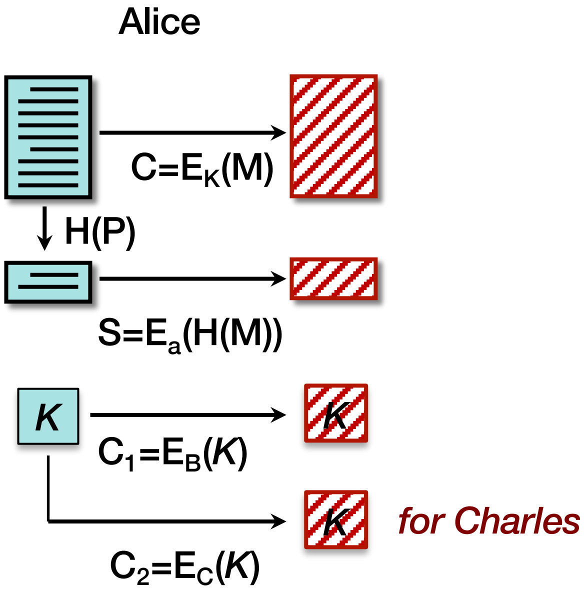
Alice generates a digital signature by encrypting the message with her private key

Covert and authenticated messaging



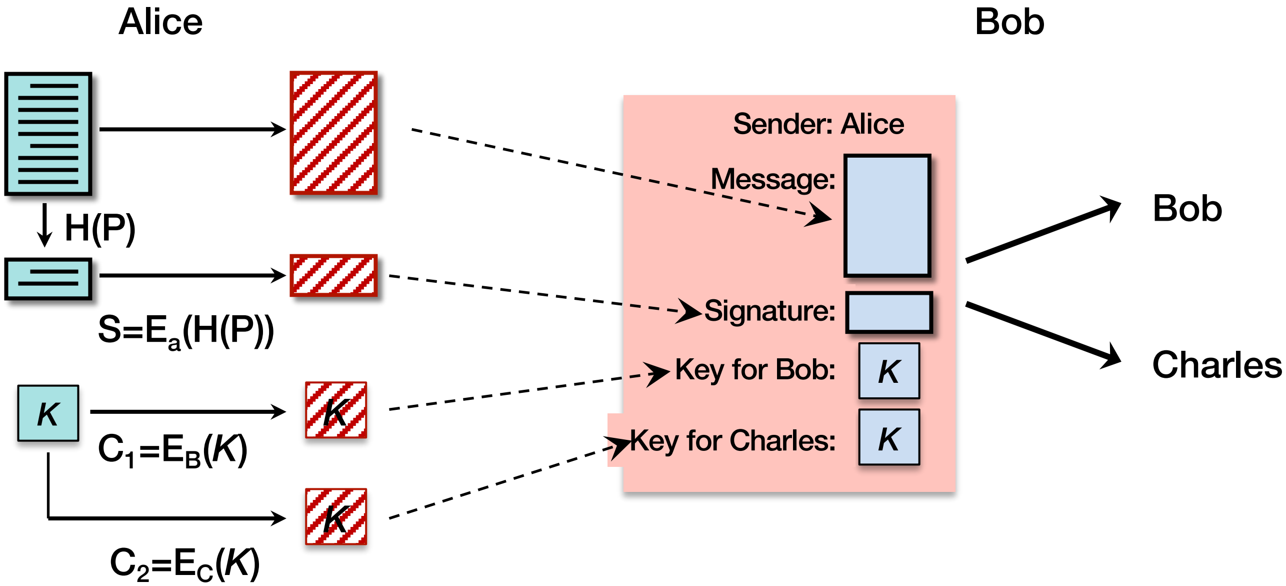
Alice picks a random key, *K*, and encrypts the message *P* with it using a symmetric cipher

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

Certificates: Identity Binding

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: your Bitcoin identity = hash(public key)

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?

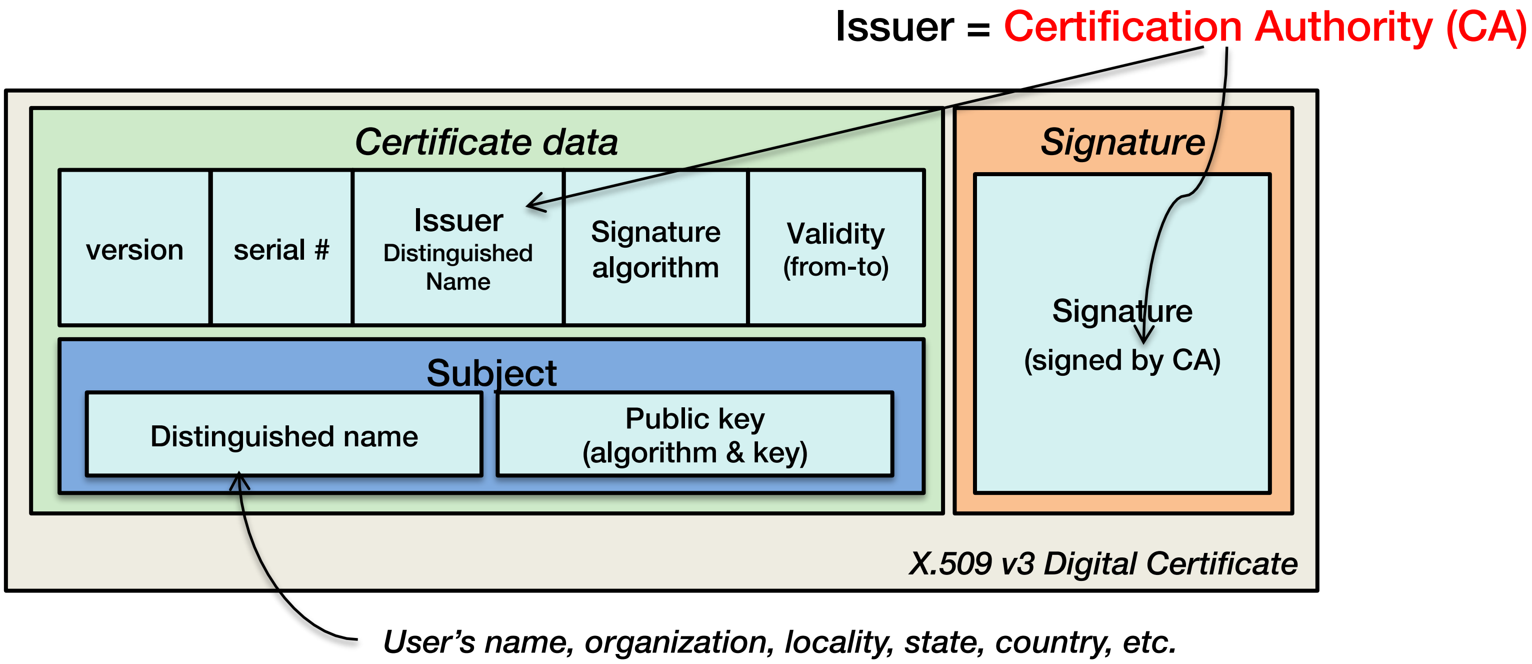
Identity Binding – Another Option

- **Have a trusted party sign Bob's public key**
- **Once signed, it is tamper-proof**
  - An attacker cannot generate the signature after modifying the key
- **But we need to know it's Bob's public key and who signed it**
  - Create & sign a data structure that
    - Identifies Bob
    - Contains his public key
    - Identifies who is doing the signing

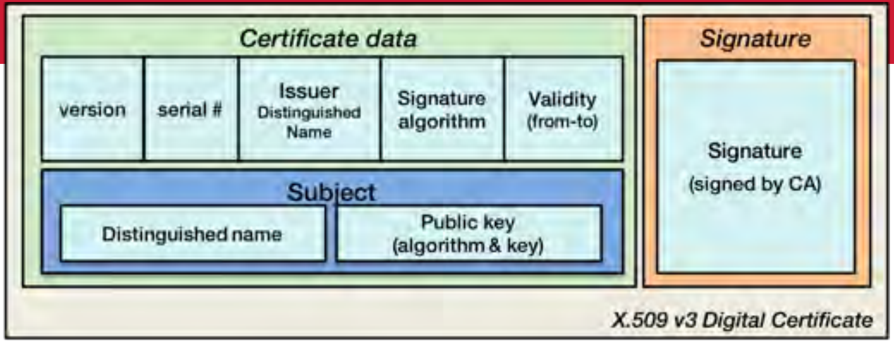
X.509 Certificates

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. Get the issuer (CA) from the certificate
2. Validate the certificate's signature against the issuer's public key
   - Hash contents of certificate data
   - Decrypt CA's signature with <u>CA's public key</u>

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

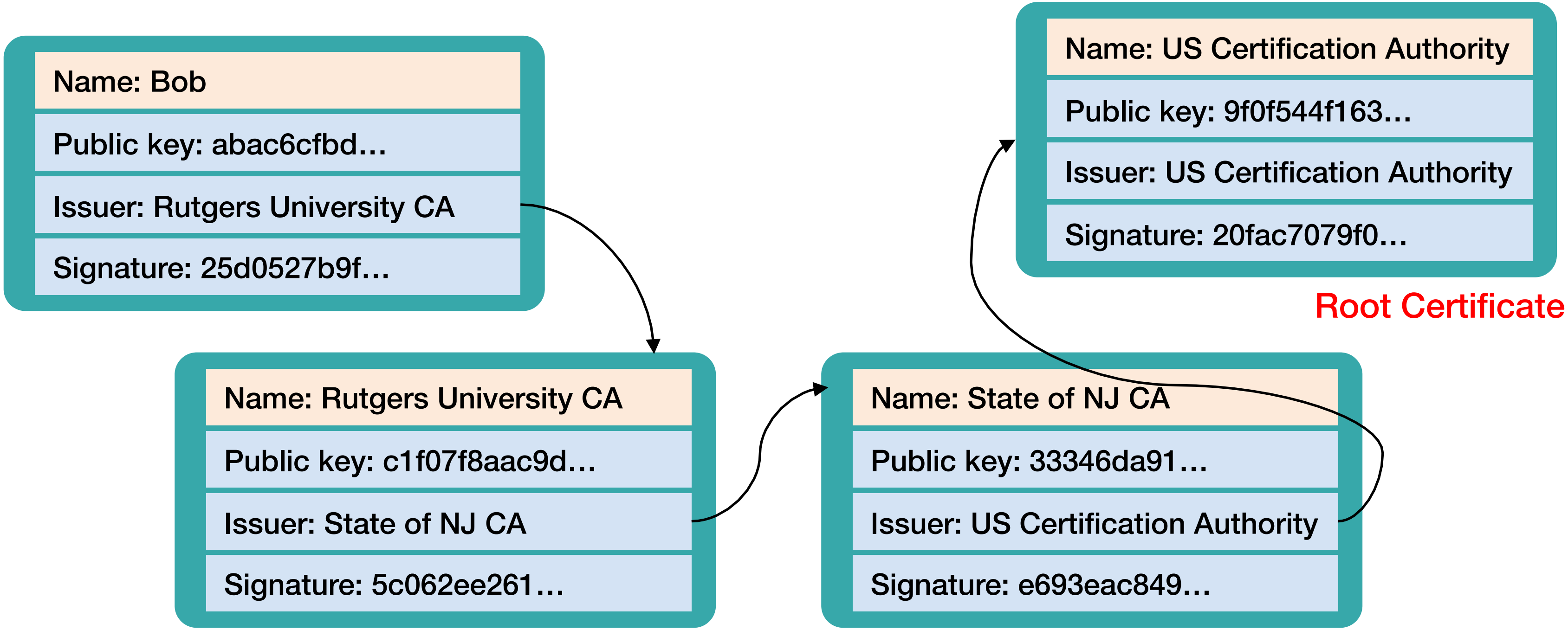
Certificates prevent someone from using a phony public key to masquerade as another person

…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



Certification Authorities (CAs)

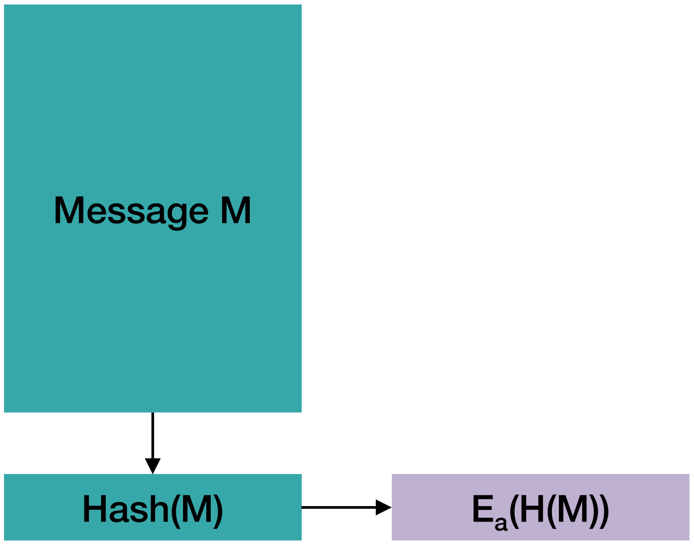
- **But trust must start somewhere**
  You need a public key you can trust – this is the root certificate
  - Apple's Trust Store is pre-loaded with over 160 CA certificates
    - Stores non-personal security info; accessed via Keychain
  - Windows stores them in the Certificate Store and makes them 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 (CPS)
  - 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 since XP*: 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**

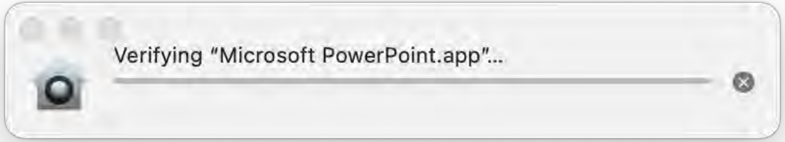
*Windows XP had partial support for Authenticode; it did not support signed drivers.

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 from a certification authority (CA) that is enrolled in the *Microsoft Trusted Root Certificate Program*
  - 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 package
- **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 hashses

- **Integrity check when program is first loaded (this takes time)**



- **Check a hash for a page when it is needed (demand paging)**
  - This is efficient (pages are small; checking a hash is quick)

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