CS 1699
Privacy in the Electronic Society

William Garrison
bill@cs.pitt.edu
6311 Sennott Square
http://cs.pitt.edu/~bill/445

04: Hashing and public-key cryptography

Today's topics: Continuing crypto basics

Encryption does not automatically provide integrity

- But, block ciphers can be used as MACs

Hash functions are important across cryptography

- Including integrity via HMAC

Symmetric crypto has a key distribution problem

- Public-key crypto can help
- Hybrid crypto combines both

Encryption does not provide integrity/authenticity!

Just because a message decrypts does not mean it’s what was sent!

You may have experience with checksums to detect errors

- e.g., MD5 checksum on file

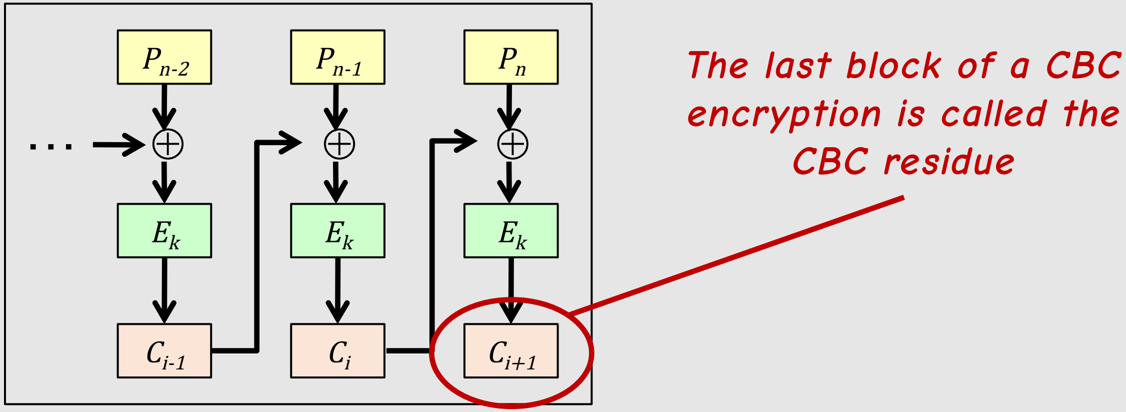
However, if an attacker can change the file, they can change the checksum too

- Need a keyed primitive!

Message authentication codes (MACs) can solve this problem

- MAC(k, m) represents the MAC of message m using key k

The CBC residue of an encrypted message can be used as a cryptographic MAC



How does this work?

- Use a block cipher in CBC mode to encrypt m using the shared key k
- Save the CBC residue r, transmit m and r to the remote party
- The remote party recomputes and verifies the CBC residue of m

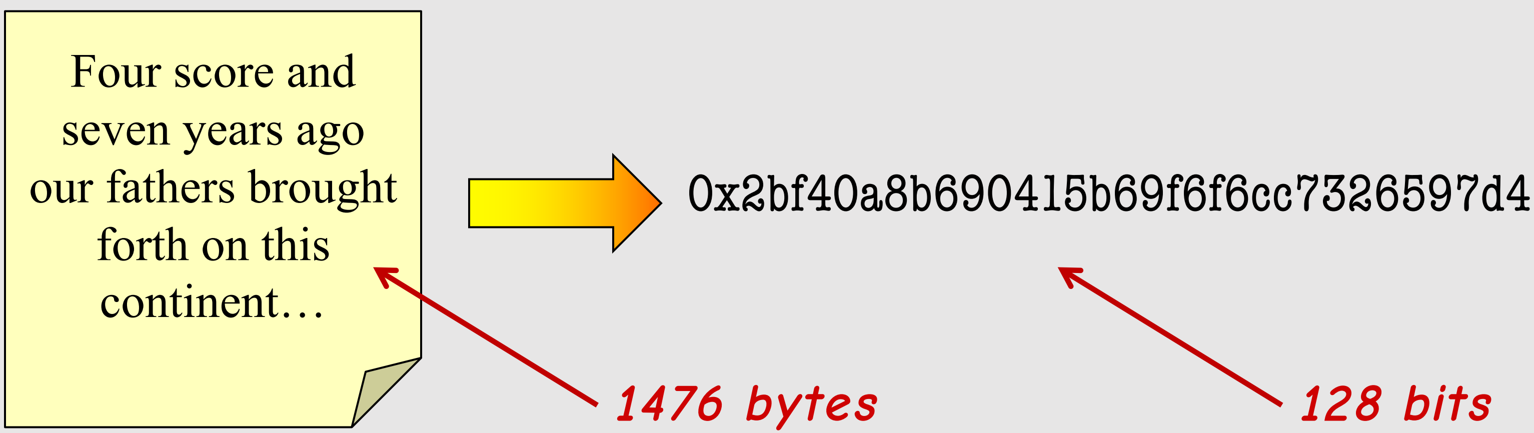
Why does this work?

- Malicious parties can still manipulate m in transit
- However, without k, they cannot compute the corresponding CBC residue!

The bad news: Encrypts the whole message, need 2 keys for confidentiality and integrity

What is a hash function?

A hash function is a function that maps variable-length input to fixed-length output



Intuitively, a cryptographically strong hash function needs to appear random in output

More formally, cryptographic hash functions should satisfy three properties

Assume that we have a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^m$

Preimage resistance: Given a hash output value z, it should be infeasible to calculate a message x such that H(x) = z

- i.e., H is a one way function
- Ideally, computing x from z should take $O(2^m)$ time

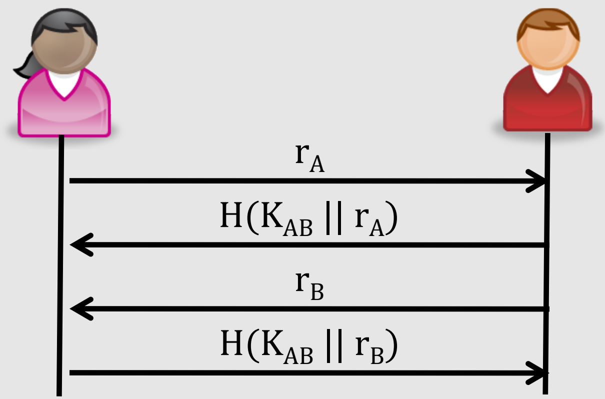
Second preimage resistance: Given a message x, it is infeasible to calculate a second message y such that H(x) = H(y)

- Note that this attack is always possible given infinite time (Why?)
- Ideally, this attack should take $O(2^m)$ time

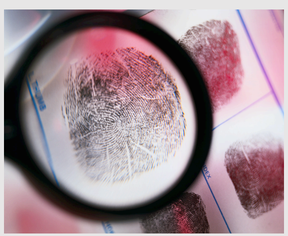
Collision resistance: It is infeasible to find two messages x and y such that H(x) = H(y)

- Ideally, this attack should take $O(2^m/2)$ time

What can we do with strong hash functions?



Mutual Authentication



Document Fingerprinting

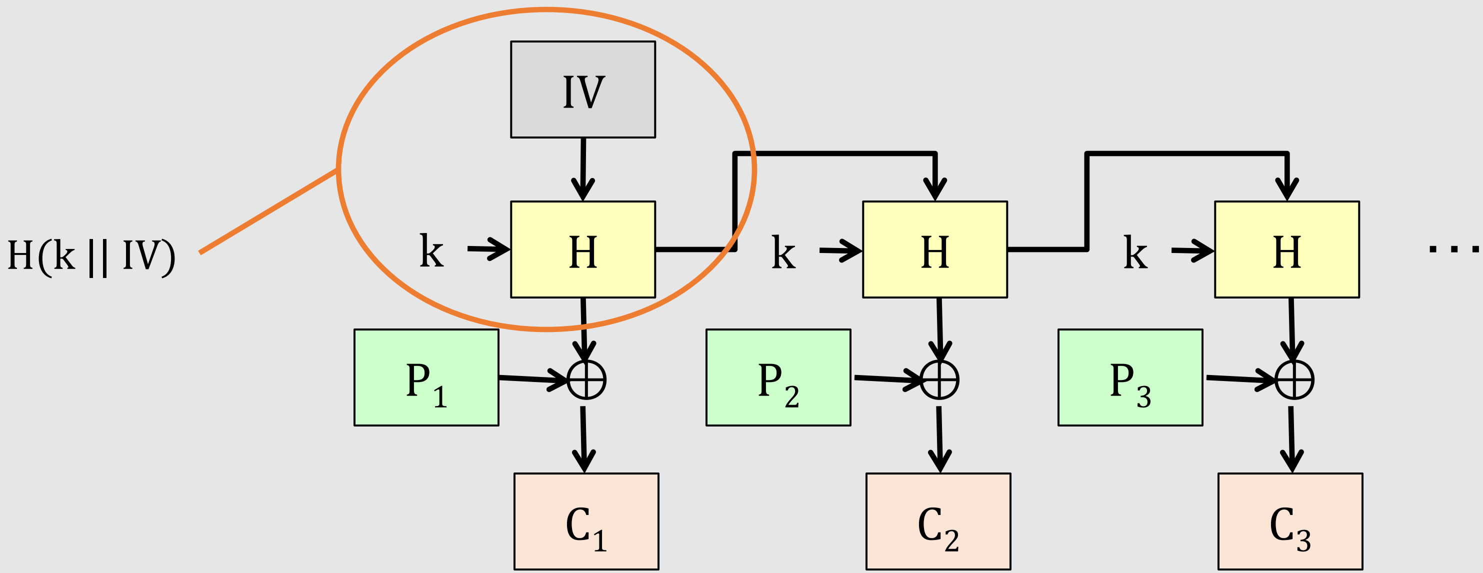
- Use H(D) to see if D has been modified

MAC Functions

- Assume a shared key K
- Sender:
  - Compute c = $E_K(H(m))$
  - Transmit m and c
- Receiver:
  - Compute d = $E_K(H(m))$
  - Compare c and d

Hash functions can even be used to generate cipher keystreams!

Consider a block mode similar to OFB (output feedback)



SHA-1 is built using the Merkle-Damgård construction

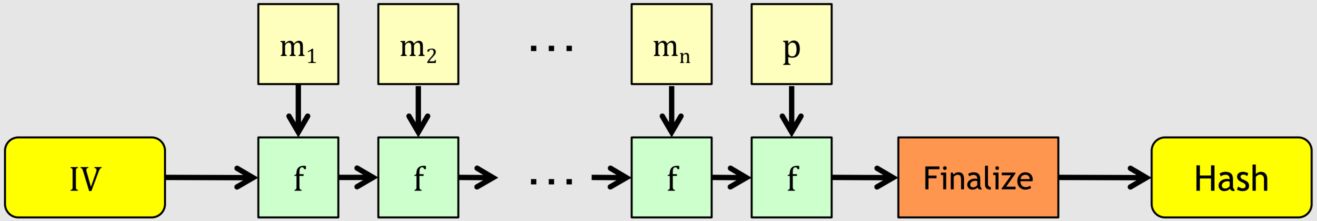
The Merkle-Damgård construction is a "template" for constructing cryptographic hash functions

- Proposed in the late '70s
- Named after Ralph Merkle and Ivan Damgård

Essentially, a Merkle-Damgård hash function does the following:

- Pad the input message if necessary
- Initialize the function with a (static) IV
- Iterate over the message blocks, applying a compression function f
- Finalize the hash block and output

Why is a static IV needed?



Merke and Damgård independently showed that the resulting hash function is secure if the compression function is collision resistant

A thousand-mile view…

Input: A message of bit length $\leq 2^{64} - 1$

Output: A 160-bit digest

Steps:

- Pad message to a multiple of 512 bits
- Process one 512 bit chunk at a time
- Expand the sixteen 32-bit words into eighty 32-bit words
- Initialize five 32-bit words of state
- For each block of five 32-bit words
  - Apply function at right
  - Add result to output
- Concatenate five 32-bit words of output state

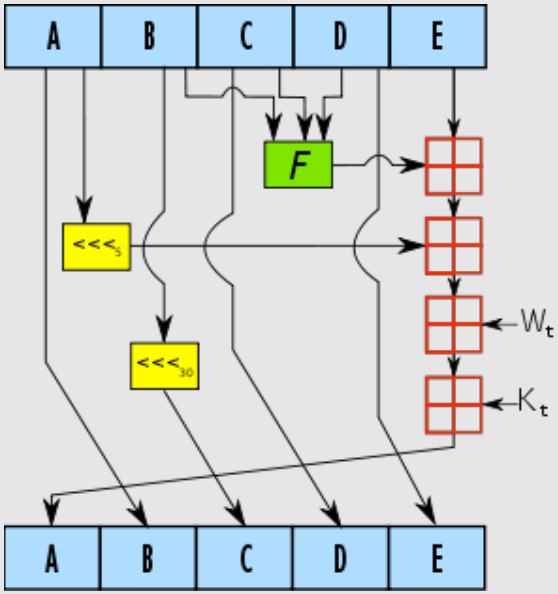


Image from Wikipedia

Initialization and Padding

Note: These variables comprise the internal state of SHA-1. They are continuously updated by the compression function, and are used to construct the final 160-bit hash value.

Initialize variables:

h0 = 0x67452301

h1 = 0xEFCDAB89

h2 = 0x98BADCFE

h3 = 0x10325476

h4 = 0xC3D2E1F0

Pre-processing:

append the bit ‘1’ to the message

append $0 \leq k < 512$ bits ‘0’, so that the resulting message length (in bits)

is congruent to $448 \equiv -64 \pmod{512}$

append length of message (before pre-processing), in bits, as 64-bit big-endian integer

Example:

0xDEADBEEF → 0xDEADBEEF8000 … 0020

32 bits

$32_{10} = 0x20$

Initializing the compression function

```
Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    Extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
            w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16])  <<< 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4
```

Note: <<< denotes a left rotate.

Example: 00011000 <<< 4

⇓

10000001

Main body of the compression function

Main loop:

```
for i from 0 to 79
    if 0 ≤ i ≤ 19 then
            f = (b and c) or ((not b) and d); k = 0x5A827999
    else if 20 ≤ i ≤ 39
            f = b xor c xor d; k = 0x6ED9EBA1
    else if 40 ≤ i ≤ 59
            f = (b and c) or (b and d) or (c and d); k = 0x8F1BBCDC
    else if 60 ≤ i ≤ 79
            f = b xor c xor d; k = 0xCA62C1D6

    temp = (a <<< 5) + f + e + k + w[i]
    e = d; d = c; c = b <<< 30; b = a; a = temp
```

Add this chunk's hash to result so far:

```
h0 = h0 + a; h1 = h1 + b; h2 = h2 + c; h3 = h3 + d; h4 = h4 + e
```

Note: Sometimes, we treat state as a bit vector…

… but other times, it is treated as an unsigned integer

Finalizing the result

Produce the final hash value (big-endian):

output = h0 || h1 || h2 || h3 || h4

"||" denotes concatenation

Interesting note:

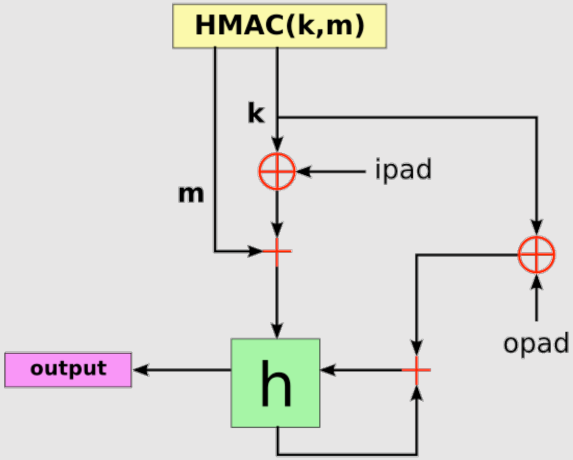
- k_1 = 0x5A827999 = $2^{30} \times \sqrt{2}$
- k_2 = 0x6ED9EBA1 = $2^{30} \times \sqrt{3}$
- k_3 = 0x8F1BBCDC = $2^{30} \times \sqrt{5}$
- k_4 = 0xCA62C1D6 = $2^{30} \times \sqrt{10}$

Question: Why might it make sense to choose the k values for SHA-1 in this manner?

HMAC is a construction that generates a strong MAC from a hash function

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

- opad = 01011010
- ipad = 00110110



The opad and ipad constants were carefully chosen to ensure that the internal keys have a large Hamming distance between them

Note that H can be any hash function. For example, HMAC-SHA-1 is the name of the HMAC function built using the SHA-1 hash function.

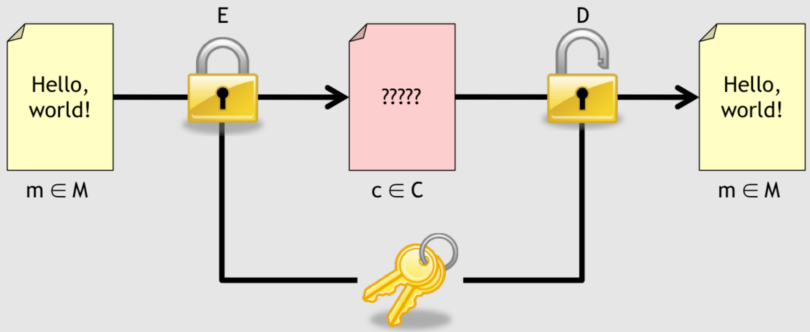
Benefits of HMAC:

- Hash functions are faster than block ciphers
- Good security properties
- Since HMAC is based on an unkeyed primitive, it is not controlled by export restrictions!

Public-Key Cryptography

Motivation

Recall: In a symmetric key cryptosystem, the same key is used for both encryption and decryption



Note: The sender and recipient need a shared secret key

The good news is that symmetric key algorithms…

- Have been well-studied by the cryptography community
- Are extremely fast, and thus good for encrypting bulk data
- Provide good security guarantees based on very small secrets

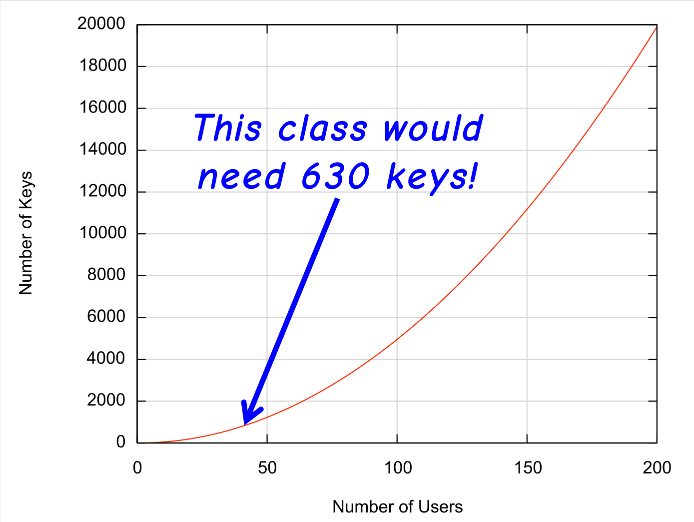
Unfortunately…

Symmetric key cryptography is not a panacea

Question: What are some ways in which the need for a shared secret key might cause a problem?

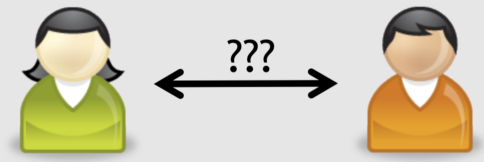
Problem 1: Key management

- In a network with n participants, C(n,2) = n(n-1)/2 keys are needed!
- This number grows very rapidly!



Problem 2: Key distribution

- How do Alice and Bob share keys in the first place?



- What if Alice and Bob have never met in person?
- What happens if they suspect that their shared key K_{AB} has been compromised?

Wouldn't it be great if we could securely communicate without needing pre-shared secrets?

Thought Experiment

Forget about bits, bytes, ciphers, keys, and math…

The Scenario: Assume that Alice and Bob have never met in person. Alice has a top secret widget that she needs to send to Bob using an untrusted courier service. Alice and Bob can talk over the phone if needed, but are unable to meet in person. Due to the high-security nature of their work, the phones used by Alice and Bob may by wiretapped by other secret agents.

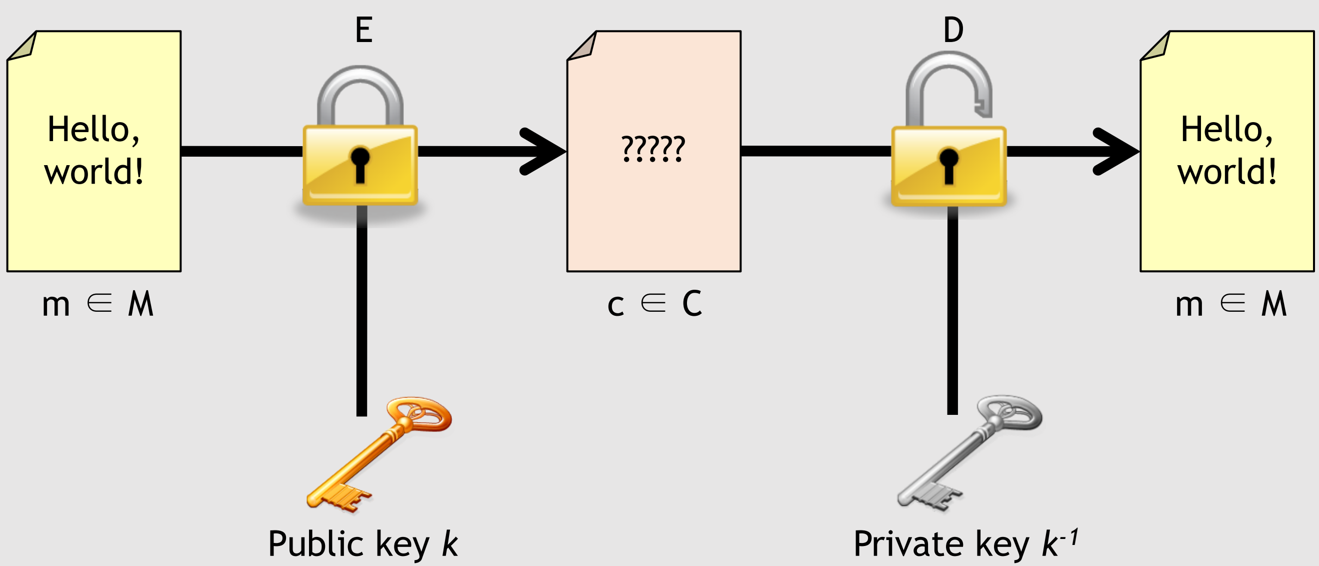
Problem: How can Alice send her widget to Bob while having very high assurance that Bob is the only person who will be able to access the widget if it is properly delivered?

Public key cryptosystems are a digital counterpart to the strongbox example

Formally, a cryptosystem can be represented as the 5-tuple (E, D, M, C, K)

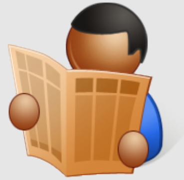
- M is a message space
- K is a key space
- E : M × K → C is an encryption function
- C is a ciphertext space
- D : C × K → M is a decryption function

Note: Each "key" in K is actually a pair of keys, (k, k^{-1})



What can we do with public key cryptography?

First, we need some way of finding a user's public key



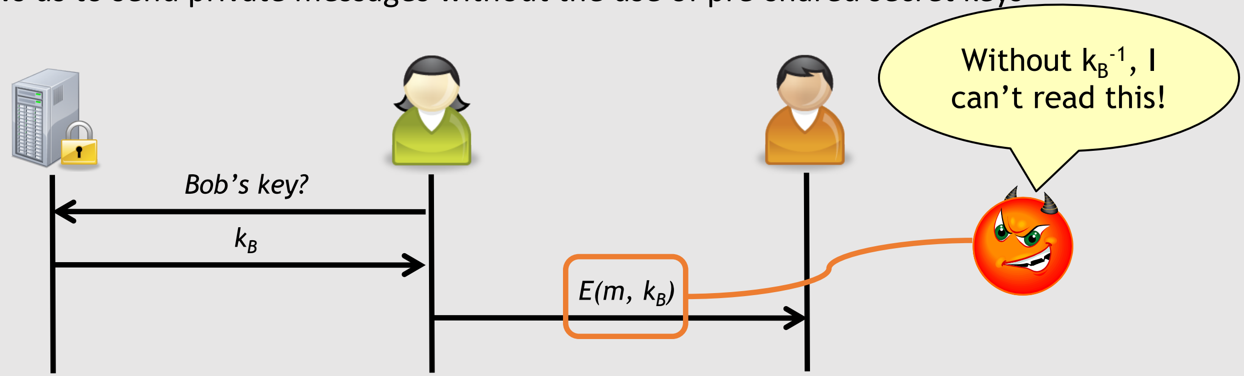
Print it in
the newspaper

Post it on your
webpage

A trusted
keyserver (PKI)

Important: It is critical to verify the authenticity of any public key! (How?)

Public key cryptography allows us to send private messages without the use of pre-shared secret keys



Diffie-Hellman key exchange: Public-key for deriving a symmetric key

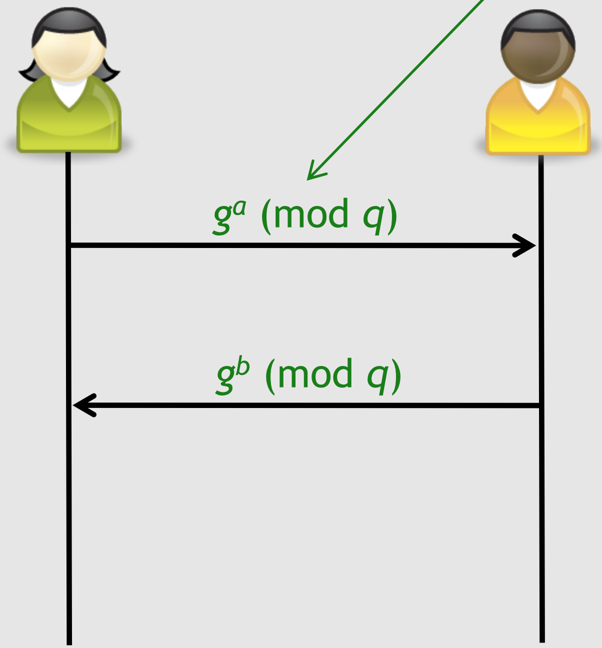
Step 0: Alice and Bob agree on a finite cyclic group G of (large) prime order q, and a generator g for this group. This information is all public.

a is Alice's private key

$g^a \pmod{q}$ is Alice's public key

Step 1:

- Randomly choose $a \in \{1, 2, \ldots, q\text{-}1\}$
- Compute $g^a \pmod{q}$
- Send $g^a \pmod{q}$



Step 2:

- Randomly choose $b \in \{1, 2, \ldots, q\text{-}1\}$
- Compute $g^b \pmod{q}$
- Send $g^b \pmod{q}$

Step 3:

- Compute $(g^b \pmod{q})^a \pmod{q}) = g^{ba} \pmod{q} = K_{ab}$

Step 3':

- Compute $(g^a \pmod{q})^b \pmod{q}) = g^{ab} \pmod{q} = K_{ab}$

Why is the Diffie-Hellman key exchange protocol safe?

Recall: We need to show that it is hard for an attacker to learn any of the secret information generated by this protocol, assuming that they know all public information

Public information: $G, g, q, g^a \pmod q, g^b \pmod q$
Private information: $a, b, K_{ab} = g^{ab} \pmod q$

Tactic 1: Can we get $g^{ab} \pmod q$ from $g^a \pmod q$ and $g^b \pmod q$?

- We can get $g^{am+bn} \pmod q$ for arbitrary m and n, but this is no help…

Tactic 2: Can we get a from $g^a \pmod q$?

- This called taking the discrete logarithm of $g^a \pmod q$
- The discrete logarithm problem is widely believed to be very hard to solve in certain types of cyclic groups

Conclusion: If solving the discrete logarithm problem is hard, then the Diffie-Hellman key exchange is secure!

The RSA cryptosystem picks up where Diffie and Hellman left off

RSA was proposed by Ron Rivest, Adi Shamir, and Leonard Adelman in 1978. It can be used to encrypt/decrypt and digitally sign arbitrary data!

Key generation:

- Choose two large prime numbers p and q, compute $n = pq$
- Compute $\phi(n) = (p\text{-}1)(q\text{-}1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n) + 1} \pmod{n} = M^1 \pmod{n} = M$

Unfortunately, RSA is slow when compared to symmetric key algorithms like AES or HMAC-*X*

Using RSA as part of a hybrid cryptosystem can speed up encryption

- Generate a symmetric key k_s
- Encrypt m with k_s
- Use RSA to encrypt k_s using public key k
- Transmit $E(k_s, m)$, $E(k, k_s)$

Using hash functions can help speed up signing operations

- Intuition: H(m) << m, so signing H(m) takes far less time than signing m
- Why is this safe? H's preimage resistance property!

Conclusions

Integrity can be provided by symmetric crypto via residues

Hash functions can provide faster MACs

Symmetric encryption is fast, but has key management issues

Public-key crypto improves key management, but is much slower

Hybrid cryptography combines public-key distribution with symmetric (or hash) speed for bulk of the work