A Brief Introduction to Practical Cryptography

Part 1:
Message Integrity

Goal: Message Integrity

Alice wants to send message ***m*** to Bob

- don’t fully trust the messenger or network carrying the message
- want to be sure what Bob receives is actually what Alice sent

Threat model:

- Mallory can see, modify, forge messages
- Mallory wants to trick Bob into accepting a message Alice didn’t send

One approach:

1. Alice computes **v** := ***f***(**m**)

2. 

e.g. "Attack at dawn", 628369867…

3. Bob verifies that **v'** = ***f***(**m'**), accepts message iff this is true

Function ***f*** ?

Easily computable by Alice and Bob; <u>not</u> computable by Mallory

(Idea: Secret only Alice & Bob know)

We're sunk if Mallory can learn ***f***(**x**) for any **x** ≠ **m**!

Candidate $\boldsymbol{f}$:

Random function

Input: Any size up to huge maximum

Output: Fixed size (e.g. 256 bits)

Defined by a giant lookup take that's filled in by flipping coins

```
0 → 0x0b6589fc6ab0…
1 → 0x1356a192b791…
2 → 0x2da4b9237bac…
⋮          ⋮
```

Completely <u>impractical</u>

Provably <u>secure</u>

(Mallory can't do better than random guessing)

Want a function that's practical
but "looks random"…

Pseudorandom function (PRF)

Let's build one:

Start with a big *family of functions*

$\boldsymbol{f}_0, \boldsymbol{f}_1, \boldsymbol{f}_2, \ldots$ all known to Mallory

Use $\boldsymbol{f}_\mathbf{k}$, where **k** is a secret value (or "key") known only to Alice/Bob

k is (say) 256 bits, chosen randomly

Kerckhoffs's Principle

Don't rely on secret functions

Use a secret key, to choose from a function family [Why?]

Auguste Kerckhoffs (1853)

Why Kerchoffs's Principle?

1. can quantify probability that adversary will guess key because chosen randomly from known space
2. different people can use same system, different keys:
   Alice and Bob use one key, Charlie and Diane use another
3. can change key if something goes wrong

Formal definition of a secure **PRF**:

Game against Mallory

1. We flip a coin secretly to get bit **b**
2. If **b**=0, let ***g*** be a random function
   If **b**=1, let ***g*** = $\boldsymbol{f}_k$, where **k** is a randomly chosen secret
3. Repeat until Mallory says "stop":
   Mallory chooses **x**; we announce ***g***(**x**)
4. Mallory guesses **b**

We say ***f*** is a *secure PRF* if Mallory can't do better than random guessing*

i.e., $\boldsymbol{f}_k$ is indistinguishable in practice from a random function, unless you know **k**

Important fact: There's an algorithm that always wins for Mallory

[What is it?] [How to fix it?]

*actually, it's OK if Mallory has a negligible advantage over guessing, so long as it's vanishingly small

---

Important fact:

There is an algorithm that always wins for Mallory

1. get a long list of (x, g(x)) pairs
2. try every value of k to see whether f_k matches that list
3. guess b=1 iff some f_k matches

To fix this, need to limit Mallory to "practical" or "efficient" algorithms

Won't define this precisely in this course
(but precise definitions do exist)

A solution for Alice and Bob:

1. Let ***f*** by a secure PRF
2. In advance, choose a random **k** known only to Alice and Bob
3. Alice computes **v** := f_k(**m**)
4. 

5. Bob verifies that **v'** = f_k(**m'**), accepts message iff this is true

[Important assumptions?]

What if Alice and Bob want to send more than one message?

[Attacks?] [Solutions?]

Important Assumptions:
- Physical security
- Key management
- Note key exchange in advance

--

What if Alice and Bob want to send more than one message?
- can't just redo same plan
- replay attacks / reordering attacks

Solutions:
- add sequence number to message
- or, use different key each time

Annoying question:
Do PRFs actually exist?

Annoying answer:
We don't know.

Best we can do…

Well-studied functions where we haven't spotted a problem yet (e.g. HMAC-SHA256)

Terminology

Message Authentication Code (MAC)

(essentially the same as a PRF)

Currently popular "PRF" (we hope!)

HMAC-SHA256

see RFC 2104

$\text{HMAC}_k(\mathbf{m})$ =

$$\text{SHA256}\left(k \oplus c_1 \parallel \text{SHA256}\left(k \oplus c_2 \parallel m\right)\right)$$

XOR   `0x3636…`   `0x5c5c…`

Concatenation

SHA256 function

takes arbitrary length input,
returns 256-bit output

What is SHA256?

"Cryptographic hash function"

Input: arbitrary length data (No key)
Output: 256 bits

Built with "compression function" ***h***

(256 bits, 512 bits) in → 256 bits out

Designed to be really hairy
We won't go into details

Entire algorithm:

1. Pad input to multiple of 512 bits (using a fixed algorithm [Why?])
2. Break into 512-bit blocks $\mathbf{b}_0$, $\mathbf{b}_1$, … $\mathbf{b}_\mathbf{n}$-1
3. $\mathbf{y}_0$ =const, $\mathbf{y}_1 = \boldsymbol{h}(\mathbf{y}_0,\mathbf{b}_0)$, …, $\mathbf{y}_\mathbf{i} = \boldsymbol{h}(\mathbf{y}_\mathbf{i}\text{-}1,\mathbf{b}_\mathbf{i}\text{-}1)$
4. Return $\mathbf{y}_\mathbf{n}$

Pad using a fixed algorithm – why?
- Part of the definition of the function
- Otherwise different implementations would calculate different outputs for the same inputs

Compression function h:
- designed to be really hairy
- lots of bit twiddling
- look up in book if you want; not very enlightening

Message Authentication Code (MAC)

e.g. HMAC-SHA256

think of as synonymous with PRF

vs.

Cryptographic hash function

e.g. SHA256

not a strong PRF

Used to think the distinction didn't matter, now we think it does

Better to use a MAC/PRF (not a hash)

```
$ openssl dgst -sha256 -hmac <key>
```

[What if you don't need a key?]

If you don't need a secret key, pick a key and announce it, use different keys for different purposes to be safe.

--

Alternate ending if out of time here:

Also useful to "stretch" a shared secret: given one shared secret key, generate two

Do this trick over and over to generate N shared secret keys

"pseudorandom generator"

topic of the next lecture

Part 2:
Randomness
and
Pseudorandomness

Intro:

Randomness has a central role in cryptography,
but randomness is hard to get, hard to share,
so want to use pseudorandomness instead.

Review

Problem:

Integrity of message from Alice to Bob

Alice must append bits to message that only Alice (or Bob) can make

Solution:

Random function

Practical solution:

Pseudorandom function (PRF) –
$\boldsymbol{f}_{\mathbf{k}}$ is indistinguishable in practice from a random function, unless you know **k**

Where do these random keys **k** come from … ?
Careful: We're often sloppy about what is "random"

we're often sloppy about what is "random"
e.g., rand() function is C library is not at all random
"some random kids were there"

need to be precise in crypto --- or we'll be sorry

True Randomness

- Output of a physical process that is inherently random
- Scarce and hard to get

Pseudorandom generator (PRG)

- Takes small seed that is really random
- Generates long sequence of numbers that are "as good as random"

This should sound familiar…

Definition: **PRG** is secure if it's indistinguishable from random

Similar game to PRF definition:

1. We flip a coin secretly to get a bit **b**
2. If **b**=0, let ***s*** be a truly random stream
   If **b**=1, let ***s*** be $\boldsymbol{g}_{\mathbf{k}}$ for random secret **k**
3. Mallory can see as much of the output of ***s*** as he/she wants
4. Mallory guesses **b**,
   wins if guesses correctly

Say **g** is a secure PRG if there is no winning strategy for Mallory*

*usual caveats:
- OK if negligible advantage,
- only practical strategies allowed

Here's a *simple PRG that works:*

For some random **k** and PRF ***f***,
output: $f_k(0) \,||\, f_k(1) \,||\, f_k(2) \,||\, \ldots$

Theorem: If ***f*** is a secure PRF, and ***g*** is built from ***f*** by this construction, then ***g*** is a secure PRG.

Proof: Assume ***f*** is a secure PRF, we need to show that ***g*** is a secure PRG.

Proof by contradiction:

1. Assume ***g*** is *not* secure;
   therefore Mallory can win the PRG game
2. This gives Mallory a winning strategy for the PRF game:
   a. query the PRF with inputs 0, 1, 2, …
   b. apply the PRG-distinguishing algorithm
3. Therefore, Mallory can win the PRF game, which is a contradiction
4. Therefore, g is secure

…it seems like we're in good shape, but we still need a truly random value k.

We could measure physical randomness…
   but: often, bits are biased, not independent
   … and how do we know if we have enough "randomness"?

And: Does physical randomness actually exist?
   i.e., Is the universe inherently random?
   while philosophers are debating that question, let's try another approach

Where do we get true randomness?

Want “indistinguishable from random” which means: adversary can’t guess it

Gather lots of details about the computer that the adversary will have trouble guessing [Examples?]

- Problem: Adversary can predict some of this
- Problem: How do you know when you have enough randomness?

Modern OSes typically collect randomness, give you API calls to get it

- e.g., Linux:
- `/dev/random` is a device that gives random bits, blocks until available
- `/dev/urandom` gives output of a PRG, nonblocking, seeded from `/dev/random`

Examples of random sources?
exact history of keypresses, including micro-time
exact path of mouse
exact history of network packet arrival
internal temperature of computer
ambient noise picked up by the microphone
maybe even add hardware that will behave unpredictably
   example: camera pointed at lava lamps

--

Problem: Adversary can predict some of this

- Not a problem, as long as there is “enough randomness” in the data
- Gather data for “a long time” then run it through a PRF
- Intuition: “distill out” the randomness, reduce size but keep randomness

Problem: How do you know when you have enough randomness?
- If you use PRF output before you have enough, you’ll be sorry
- Usual solution: Collect way too much, just to be sure

Part 3: Confidentiality

Review

Problem:

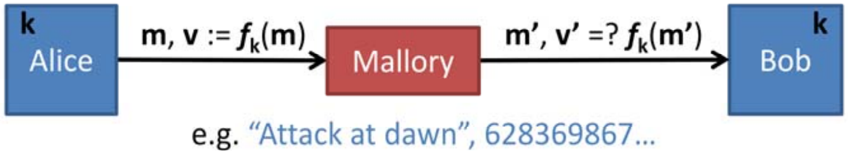
Integrity of message from Alice to Bob over an untrusted channel

Alice must append bits to message that only Alice (or Bob) can make

Solution:

Random function

Practical solution:

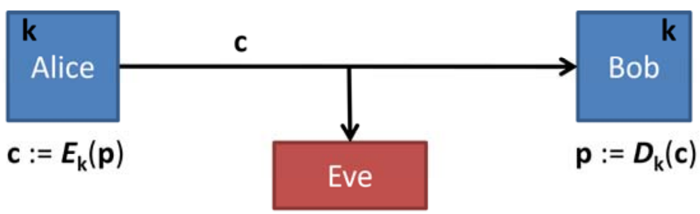


Pseudorandom function (PRF)

$\boldsymbol{f}_k$ is indistinguishable in practice from a random function, unless you know **k**

Confidentiality

Goal: Keep contents of message **p** secret from an eavesdropper



Terminology

p	plaintext
c	ciphertext
k	secret key
E	encryption function
D	decryption function

Digression: **Classical Cryptography**

Caesar Cipher

First recorded use: Julius Caesar (100-44 BC)

Replaces each plaintext letter with one a fixed number of places down the alphabet

Encryption: $\mathbf{c_i} := (\mathbf{p_i} + \mathbf{k}) \bmod 26$

Decryption: $\mathbf{p_i} := (\mathbf{c_i} - \mathbf{k}) \bmod 26$

e.g. (**k**=3):

```
   Plain:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  +Shift:  33333333333333333333333333
 =Cipher:  DEFGHIJKLMNOPQRSTUVWXYZABC
```

```
   Plain:  fox   go wolverines
    +Key:  333   33 3333333333
 =Cipher:  ira   jr zroyhulqhv
```

[Break the Caesar cipher?]

k=3 – This happens to be the key Caesar always used.

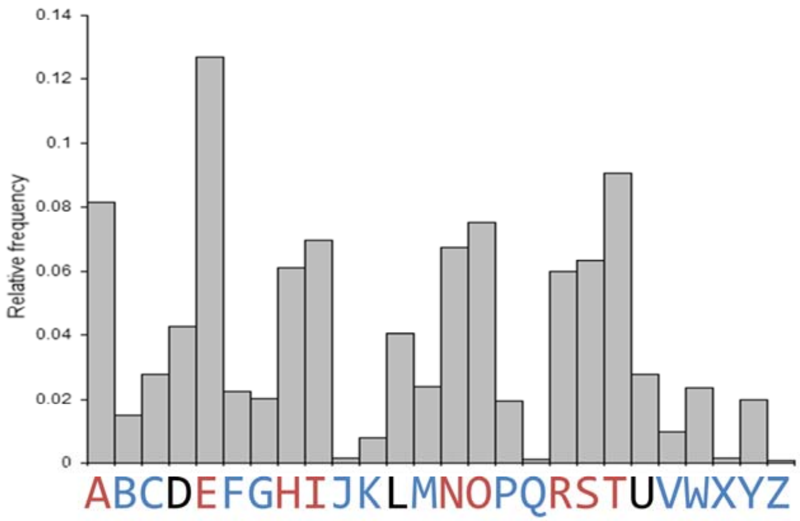
Cryptanalysis of the Caesar Cipher

Only 26 possible keys:

Try every possible **k** by *"brute force"*

Can a computer recognize the right one?

Use *frequency analysis*: English text has distinctive letter frequency distribution



Recognize with (e.g.) chi-square test

Later advance: **Vigènere Cipher**

First described by Bellaso in 1553,
later misattributed to Vigenère

Called « le chiffre indéchiffrable »
("the indecipherable cipher")

Encrypts successive letters using a sequence of Caesar ciphers determined by the letters of a keyword

For an **n**-letter keyword **k**,

Encryption: $\mathbf{c_i} := (\mathbf{p_i} + \mathbf{k}_{\mathbf{i} \text{ mod } \mathbf{n}}) \text{ mod } 26$
Decryption: $\mathbf{p_i} := (\mathbf{c_i} - \mathbf{k}_{\mathbf{i} \text{ mod } \mathbf{n}}) \text{ mod } 26$

Example: **k**=ABC (i.e. $\mathbf{k}_0=0$, $\mathbf{k}_1=1$, $\mathbf{k}_2=2$)

```
  Plain:  bbbbbb  amazon
   +Key:  012012  012012
=Cipher:  bcdbcd  anczpp
```

[Break *le chiffre indéchiffrable*?]

Cryptanalysis of the Vigènere Cipher

Simple, if we know the keyword length, **n**:

1. Break ciphertext into **n** slices
2. Solve each slice as a Caesar cipher

How to find **n**? One way: **Kasiski method**

Published 1863 by Kasiski (earlier known to Babbage?)

Repeated strings in long plaintext
will sometimes, by coincidence,
be encrypted with same key letters

```
  Plain: CRYPTOISSHORTFORCRYPTOGRAPHY
   +Key: ABCDABCDABCDABCDABCDABCDABCD
=Cipher: CSASTPKVSIQUTGQUCSASTPIUAQJB
```

Distance: 16

Distance between repeated strings in the ciphertext is likely a multiple of key length

e.g., distance 16 implies **n** is 16, 8, 4, 2, or 1

Find multiple repeats to narrow down

[What if key is as long as the plaintext?]

Charles Babbage

Back to the present:

One-time Pad (OTP)

Alice and Bob jointly generate a secret, very long, string of random bits (the one-time pad, **k**)

To encrypt: $\mathbf{c}_i = \mathbf{p}_i \text{ xor } \mathbf{k}_i$

To decrypt: $\mathbf{p}_i = \mathbf{c}_i \text{ xor } \mathbf{k}_i$

a	**b**	**a** xor **b**
0	0	0
0	1	1
1	0	1
1	1	0

a xor **b** xor **b** = **a**

a xor **b** xor **a** = **b**

"one-time" means you should never reuse any part of the pad. If you do:

Let $\mathbf{k}_i$ be pad bit
Adversary learns (**a** xor $\mathbf{k}_i$) and (**b** xor $\mathbf{k}_i$)
Adversary xors those to get (**a** xor **b**), which is useful to him [How?]

Provably secure [Why?]

Usually impractical [Why? Exceptions?]

Obvious idea: Use a **pseudorandom generator** instead of a truly random pad

(Recall: Secure **PRG** inputs a seed **k**, outputs a stream that is practically indistinguishable from true randomness unless you know **k**)

Called a **stream cipher**:

1. Start with shared secret key **k**
2. Alice & Bob each use **k** to seed the PRG
3. To encrypt, Alice XORs next bit of her generator's output with next bit of plaintext
4. To decrypt, Bob XORs next bit of his generator's output with next bit of ciphertext

Works nicely, but: don't <u>ever</u> re-use the key, or the generator output bits

Another approach: **Block Ciphers**

Functions that encrypt fixed-size blocks with a reusable key

The most commonly used approach to encrypting for confidentiality

A block cipher is <u>not</u> a pseudorandom function [Why?]

With a PRF, diff. inputs can generate same output
Random functions will have *collisions*, so will PRF

What we want instead:
pseudorandom permutation (PRP)

function from n-bit input to n-bit output

distinct inputs yield distinct outputs

Defined similarly to PRF:

practically indistinguishable from a random permutation without secret **k**

Basic challenge: Design a hairy function that is invertible, but only if know the key

Minimal properties of a good block cipher:

- Highly nonlinear ("confusion")
- Mixes input bits together ("diffusion")
- Depends on the key

Design challenges:
When designing a PRF: pile on the hairy nonlinearity; more is better
However, can't use that here, since need invertibility-with-key

Ideally, changing one bit of plaintext or key should change about half the bits of the ciphertext

Simple definition of nonlinear:
having no simple proportional relation between cause and effect

Today's most common block cipher:

AES (**Advanced Encryption Standard**)

- Designed by NIST competition, long public comment/discussion period
- Widely believed to be secure, but we don't know how to prove it
- Variable **key size** and **block size**
- We'll use 128-bit key, 128-bit block (are also 192-bit and 256-bit versions)
- Ten **rounds**: Split **k** into ten **subkeys**, performs set of operations ten times, each with diff. subkey

AES round

128-bits in, 128-bit sub-key,
128-bits out

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

Four steps:
(picture as operations on a 4x4 grid of 8-bit values)

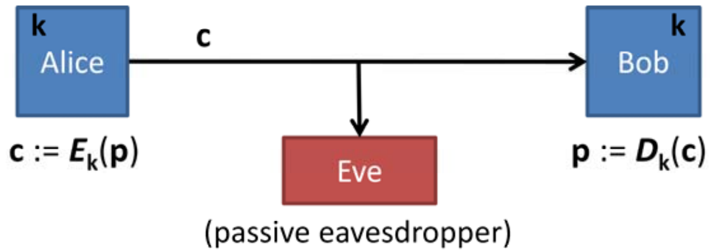
1. Non-linear step
   Run each byte thru a non-linear function (lookup table)
2. Shift step
   Circular-shift each row: **i**th row shifted by **i** (0-3)
3. Linear-mix step
   Treat each column as a 4-vector; multiply by a constant invertible matrix
4. Key-addition step
   XOR each byte with corresponding byte of round subkey

To decrypt, just undo the steps, in reverse order

Part 4:
Cipher Modes
&
Key Exchange

Review: Confidentiality

Goal: Keep contents of message **p** secret from an eavesdropper

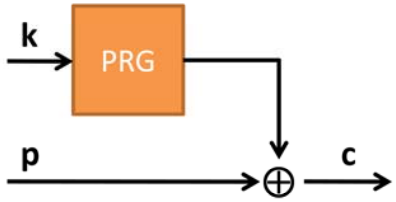


Terminology

p	plaintext
c	ciphertext
k	secret key
E	encryption function
D	decryption function

Review

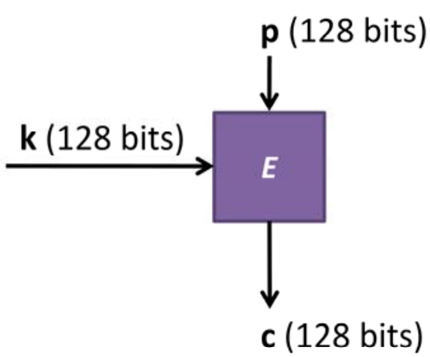
Stream Ciphers



Recall: *Secure PRG*'s output is practically indistinguishable from a random stream unless you know **k**.

<u>But:</u> Don't reuse **k**!

Block Ciphers (e.g. AES-128)



Recall: Operates in fixed-size blocks; separate decryption function; OK to reuse **k**

Block-cipher modes

We know how to encrypt one block,
but what about multi-block messages?

Different methods, called “cipher modes”

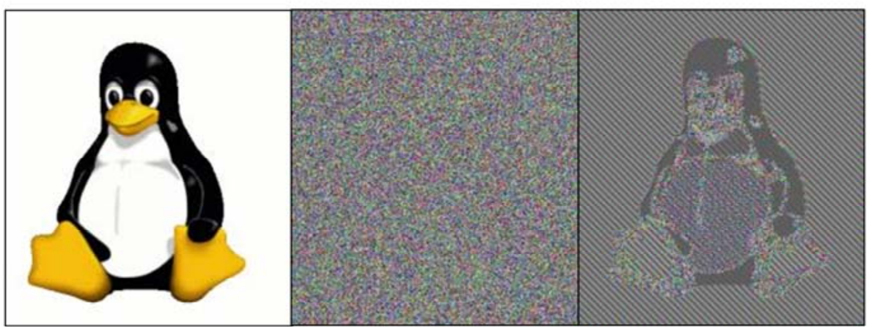
Straightforward (but bad) approach:

ECB mode (encrypted codebook)

Just encrypt each block independently

$\mathbf{C_i} := \boldsymbol{E}_{\mathbf{k}}(\mathbf{P_i})$

[Disadvantages?]



Plaintext Pseudorandom ECB mode

Disadvantages of ECB mode:
- Same plaintext yields same ciphertext (at block and message level)
- Rearrangeable

Better (and common):

CBC mode (cipher-block chaining)

Lame-CBC (for illustration only)

For each block $\mathbf{P_i}$:

1. Generate random block $\mathbf{R_i}$
2. $\mathbf{C_i} := (\mathbf{R_i}, \boldsymbol{E}_k(\mathbf{P_i} \text{ xor } \mathbf{R_i}))$

[Pros and cons?]

Lame-CBC
pro: same plaintext yields different ciphertext
con: ciphertext is 2x size of plaintext

Real CBC

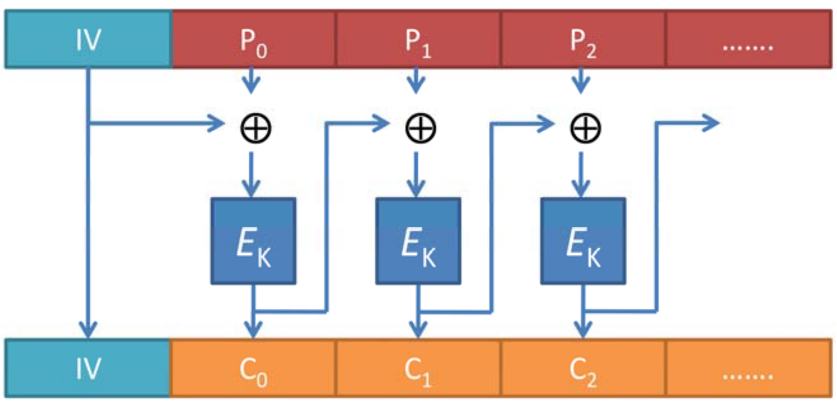
Replace $\mathbf{R}_i$ with $\mathbf{C}_{i-1}$

No need to send separately

Must still add one random $\mathbf{R}_{-1}$ to start, called "**initialization vector**" ("**IV**")

[Is CBC space-efficient?]

Illustration: CBC Encryption



[Decryption?]

space-efficient?
Clearly we need to add something to make same **p** yield diff. **c**
Here we only add one block

Remaining block-cipher problem:
How to encrypt odd-sized messages?

Padding

Can only encrypt in units of cipher **blocksize**, but message might not be multiples of blocksize

Solution: Add **padding** to end of message

Must be able to recognize and remove padding afterward

Common approach:
Add **n** bytes that have value **n**

[Caution: What if message ends at a block boundary?]

Message ending at block boundary
if message ends at block boundary, have to add whole block of padding.
Otherwise might be tricked into interpreting end of message as padding – caution!

Other modes

OFB, CFB, etc. – used less often

Counter mode

Essentially uses cipher as a pseudorandom generator

xor **i**th block of message with $\boldsymbol{E}_{\mathbf{k}}$(message_id || **i**)

CAUTION!

None of the modes we've discussed provides <u>integrity</u> protection (only confidentiality)

May not even give confidentiality unless you protect integrity some other way

Building a secure channel

What if you want confidentiality and integrity at the same time?

Encrypt, then add integrity,
not the other way around
(some reasons are subtle)

Use separate keys for
confidentiality and integrity

- Need two shared keys,
  but only have one?
  That's what PRGs are for!

- If there's a reverse (Bob to Alice)
  channel, use separate keys for that too

Amazing fact:
Alice and Bob can have a public conversation to derive a shared key!

Diffie-Hellman (D-H) key exchange

1976: Whit Diffie, Marty Hellman
with ideas from Ralph Merkle
(earlier, in secret, by Malcolm Williamson of British intelligence agency)

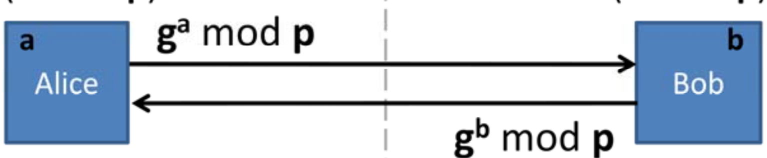
Relies on a mathematical hardness assumption called *discrete log problem* (a problem believed to be hard)

D-H protocol

1. Alice and Bob agree on public parameters (maybe in standards doc)
   p: a large "safe prime" s.t. (**p**-1)/2 is also prime
   g: a square mod **p** (but not 1)

2. **Alice**
   Generates random secret value **a**
   ($0 < \mathbf{a} < \mathbf{p}$)

   Bob
   Generates random secret value **b**
   ($0 < \mathbf{b} < \mathbf{p}$)

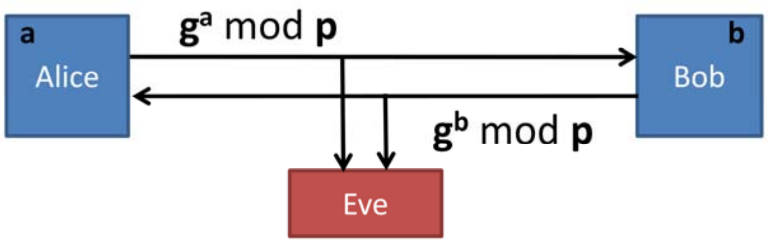


3. Computes **x**
   $= (g^b \bmod p)^a \bmod p$
   $= g^{ba} \bmod p$

   Computes **x'**
   $= (g^a \bmod p)^b \bmod p$
   $= g^{ab} \bmod p$

(Notice that **x** == **x'**)
Can use **k** := $\mathrm{HMAC}_0(\mathbf{x})$ as a shared key

Passive eavesdropping attack



Eve knows: **p**, **g**, g^a mod **p**, g^b mod **p**

Eve wants to compute **x** = g^{ab} mod **p**

Best known approach:
Find **a** or **b**, then compute **x**

Finding **y** given g^y mod **p** is an instance of the **discrete log problem**:
No known efficient algorithm

[What's D-H's big weakness?]

Notice what we've done:
Alice and Bob had public conversation to derive shared secret!
it's a very short conversation, too

Man-in-the-middle (MITM) attack

Alice does D-H exchange, *really with Mallory*, ends up with $\mathbf{g^{au}}$ mod **p**

Bob does D-H exchange, *really with Mallory*, ends up with $\mathbf{g^{bv}}$ mod **p**

Alice and Bob each think they are talking with the other, but really Mallory is between them and knows both secrets

Bottom line:
D-H gives you secure connection, but you don't know who's on the other end!

We've been talking about confidentiality in terms of a passive eavesdropper, Eve
But some attacks can do more than listen in—remember Mallory, from our discussion of integrity?
D-H can only give us confidentiality if we **already** have some assurance of integrity

chess story

- I know how to play chess against the two best players in the world, and beat one
- play black vs. Anand on Board 1, white vs. Topalov on Board 2
- algorithm
  - wait for Anand to move on Board 1
  - copy Anand's move on Board 2
  - wait for Topalov to reply on Board 2
  - copy Topalov's reply on Board 1
  - etc.
- Anand and Topalov are really playing each other
  - but they both think they're playing me
- **moral of the story:**
  - "in the middle" is a powerful place to be
    - shape the parties' views of the world

Defending D-H against MITM attacks

- Cross your fingers and hope there isn't an active adversary
- Rely on out-of-band communication between users [Examples?]
- Rely on physical contact to make sure there's no MITM [Examples?]
- Integrate D-H with user authentication

  If Alice is using a password to log in to Bob, leverage the password:

    Instead of a fixed g, derive g from the password – Mallory can't participate w/o knowing password
- Use digital signatures
  (Thursday's lecture…)

Out-of-band communication (e.g., SSH)
Example: I call you on the phone, confirm my key – works if we recognize each others' voices

Physical contact (e.g., smartcards)

Part 5:
Key Management
&
Public-Key Crypto

Review: Integrity

Problem: Sending a message over an **untrusted channel** without being changed

Provably-secure solution: **Random function**

Practical solution:

e.g. "Attack at dawn", 628369867…

Pseudorandom function (**PRF**)

Input: arbitrary-length **k**
Output: fixed-length value

Secure if practically indistinguishable from a random function, unless know **k**

Real-world use:

Message authentication codes (**MACs**) built on cryptographic hash functions

Popular example: **HMAC-SHA256$_k$(m)**

[Cautions?!]

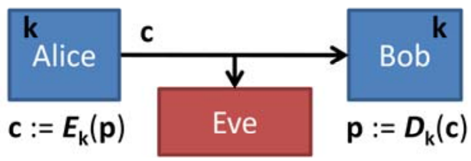
Review: Confidentiality

Problem: Sending message in the presence of an **eavesdropper** without revealing it

Provably-secure solution: **One-time pad**

Practical solution:

Pseudorandom generator (**PRG**)



Input: fixed-length **k**
Output: arbitrary-length stream

Secure if practically indistinguishable from a random stream, unless know **k**

Real-world use:

Stream ciphers (can't reuse **k**)
  Popular example: **AES-128** + **CTR mode**

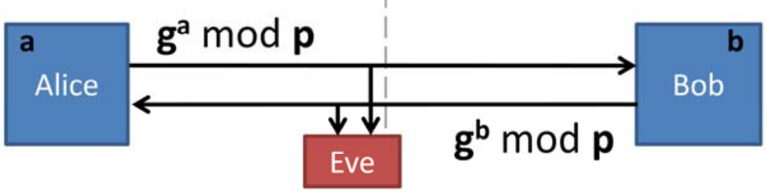
Block ciphers (need **padding/IV**)
  Popular example: **AES-128** + **CBC mode**

[Cautions?!]

Review: Diffie-Hellman Key Exchange

Lets Alice and Bob **agree on a shared secret** value by having a public conversation

Alice	**Bob**
Generates random secret **a** ($0 < a < p$)	Generates random secret **b** ($0 < b < p$)



Computes **x**	Computes **x'**
$= (g^b \bmod p)^a \bmod p$	$= (g^a \bmod p)^b \bmod p$
$= g^{ba} \bmod p$	$= g^{ab} \bmod p$

x == x'

Problem: **Man-in-the-middle attacks**

<u>Caution</u>: D-H gives you a shared secret, but don't know who's on the other end!

Issue: How big should keys be?

Want prob. of guessing to be infinitesimal… but watch out for Moore's law – safe size gets 1 bit larger every 18 months

128 bits usually safe for ciphers/PRGs

Need larger values for MACs/PRFs due to **birthday attack**

Often trouble if adversary can find <u>any two messages</u> with same MAC

Attack: Generate random values, look for coincidence

Requires $O(2^{|k|/2})$ time, $O(2^{|k|/2})$ space

For 128-bit output, takes 2^{64} steps: doable!

[Puzzle: Do it in constant space?]

Upshot: Want output of MACs/PRFs to be twice as big as cipher keys

e.g. use HMAC-SHA256 along side AES-128

2^{128} is approx. 10^{39}
at 1 trillion guesses/sec., takes 10 quadrillion times lifetime of universe

Birthday paradox: By the pigeonhole principle, 100% probability of two same birthdays with 367 people; but counter-intuitively, reach 99% probability with only 57 people, and 50% probability with just 23 people
Intuition? (23 choose 2) = 253 pairs – chances for a collision

The hard part of crypto

Key-management

Principles:

0. Always remember,
   key management is the hard part
1. Each key should have only one purpose
2. Vulnerability of a key increases:
   a. The more you use it
   b. The more places you store it
   c. The longer you have it
3. Keep your keys far from the attacker
4. Protect yourself against compromise of old keys

   Goal: **forward secrecy** — learning old key shouldn't help adversary learn new key

   [How can we get this?]

1. Each key should have only one purpose:
- diff keys for signing, encrypting
- diff keys for encrypting, MACing
- diff keys for Alice->Bob and Bob->Alice
- diff keys for diff protocols

reason: prevent attacker from "repurposing" content
example: reflection attack

2. Vulnerability of a key increases…
consequences:
- change your keys periodically, use session keys
- take care to erase keys from memory when you're done with them
- don't let your keys get swapped out to disk

3. Keep your keys far from the attacker
- memory of networked, unguarded PC: bad
- memory of non-networked, guarded PC: not as bad
- stored in tamper-resistant device: better
- stored in tamper resistant device, locked in safe: best

4. Protect yourself against compromise of old keys
- bad practice: Alice tells Bob, "Here's the new key: ..." encrypted under old key
  adversary can record this, then attack old key
  get old key, then he can get new key
  worse yet: if long chain of keys, he can attack any one, chain unravels
    chain as strong as its weakest link!

goal: "forward secrecy"
  learning old key doesn't help adversary learn new key
  how to do?
  – use Diffie-Hellman to negotiate a fresh secret
  - can use old key to provide integrity
    (as long as attacker doesn't know it today,
    learning old key tomorrow won't let attacker discover new key)

  also: actively destroy old key when you're done with it
  find all copies, write zeroes over them

Suppose Alice publishes data to lots of people, and they all want to verify integrity…

Can't share an integrity key with *everybody*, or else *anybody* could forge messages

Suppose Bob wants to receive data from lots of people, confidentially…

Schemes we've discussed would require a separate key shared with each person

[What to do?]

existing scheme is impractical

Alice would have to share an integrity key with everybody, but then anybody could put integrity mark on message

Recall that Alice and Bob know the same key, so Bob can make integ. marks
- Not a problem if only Alice and Bob
  (Bob tricking himself?)
- Trouble if many recipients

Solution:

Public-key Crypto

So far, encryption key == decryption key
  "**symmetric key crypto**"

New idea: Keys are distinct, and
  you can't find one from the other

Almost always used by splitting key-pair
  Alice keeps one key private ("**private key**")
  Publishes the other key ("**public key**")

Many applications

Invented in 1976 by Diffie and Hellman
  (earlier by Clifford Cocks of British intelligence, in secret)

Best known, most common public-key algorithm: **RSA**
  Rivest, Shamir, and Adelman 1978

How RSA works

Key generation:

1. Pick large (say, 2048 bits) random primes **p** and **q**
2. Compute **N** := **pq** (RSA uses multiplication mod **N**)
3. Pick **e** to be relatively prime to (**p**-1)(**q**-1)
4. Find **d** so that **ed** mod (**p**-1)(**q**-1) = 1
5. Finally: **Public key** is (e,N)
   Private key is (d,N)

To encrypt: $E(\mathbf{x}) = \mathbf{x}^{\mathbf{e}} \bmod \mathbf{N}$

To decrypt: $D(\mathbf{x}) = \mathbf{x}^{\mathbf{d}} \bmod \mathbf{N}$

Why RSA works

"It works" theorem:

For all $0 < \mathbf{x} < \mathbf{N}$,
can show that $\boldsymbol{E}(\boldsymbol{D}(\mathbf{x})) = \boldsymbol{D}(\boldsymbol{E}(\mathbf{x})) = \mathbf{x}$

Proof of $\boldsymbol{E}(\boldsymbol{D}(\mathbf{x}))$ side:

$\boldsymbol{E}(\boldsymbol{D}(\mathbf{x})) = (\mathbf{x}^{\mathbf{d}} \bmod \mathbf{pq})^{\mathbf{e}} \bmod \mathbf{pq}$

$= \mathbf{x}^{\mathbf{de}} \bmod \mathbf{pq}$

$= \mathbf{x}^{\mathbf{a}(\mathbf{p}-1)(\mathbf{q}-1)+1} \bmod \mathbf{pq}$ for some $\mathbf{a}$

(because $\mathbf{ed} \bmod (\mathbf{p}-1)(\mathbf{q}-1) = 1$)

$= (\mathbf{x}^{(\mathbf{p}-1)(\mathbf{q}-1)})^{\mathbf{a}}\mathbf{x} \bmod \mathbf{pq}$

$= (\mathbf{x}^{(\mathbf{p}-1)(\mathbf{q}-1)} \bmod \mathbf{pq})^{\mathbf{a}}\mathbf{x} \bmod \mathbf{pq}$

$= 1^{\mathbf{a}}\mathbf{x} \bmod \mathbf{pq}$

(because of the fact that if $\mathbf{p},\mathbf{q}$ are prime, then for all $0 < \mathbf{x} < \mathbf{N}$, $\mathbf{x}^{(\mathbf{p}-1)(\mathbf{q}-1)} \bmod \mathbf{pq} = 1$)

$= \mathbf{x}$

Is RSA secure?

Best known way to compute **d** from **e**
is factoring **N** into **p** and **q**

Best known factoring algorithm
(**general number field sieve**)
takes more than polynomial time
but less than exponential time
to factor **n**-bit number

(Still takes way too long if **p**,**q**
are large enough and random)

Fingers crossed…
but can't rule out a breakthrough

Subtle fact: RSA can be used for either confidentiality or integrity

RSA for confidentiality:

Encrypt with public key
Decrypt with private key
"your eyes only" message

RSA for integrity:

Encrypt ("sign") with private key
Decrypt ("verify") with public key
called a **digital signature**

[What if we want both confidentiality and integrity on the same message?]

For both, use RSA twice, once for each purpose (w/ separate key-pairs)

RSA drawback: Performance

Factor of 1000 or more slower than AES

Dominated by exponentiation – cost goes up (roughly) as cube of key size

Message must be shorter than **N**

[How big should the RSA keys be?]

Use in practice:

Encryption:

Use RSA to encrypt a random **x**, compute **k** := PRF(**x**), encrypt message using a symmetric cipher and key **k**

Signing:

Compute **v** := PRF(**m**), use RSA to sign a carefully padded version of **v** (many gotchas!)

Almost always should use crypto libraries to get the details right

Reasons for larger key size:
- Unlike w/ symmetric key, p and q can't be chosen uniformly at random (only primes will do)
- Also, can attack by factoring (known factoring algorithms are better than dumb brute force)
- Need some cushion for when factoring algorithms improve

Good advice today: 2048-bit p and q seem safe for foreseeable future
Use 1024-bit only if you need performance boost

Putting it all together:

A secure channel

(Assume Alice and Bob know each others' public keys)

1. They establish a shared secret **k** with D-H
2. Make sure they're really talking to each other by exchanging and verifying RSA signatures on **k**
3. Use a PRG to split **k** into four distinct keys, for integrity and confidentiality in each direction
4. To exchange messages over the channel: Encrypt them with a symmetric cipher, then add MACs for integrity

We're done… except for one nagging detail:
How do A. & B. learn each others' public keys?
Family of solutions called **public-key infrastructure (PKI)**