# applied crypto.rb
## theory & application.

martin boßlet

# Symmetric Cryptography

Encryption in the old days

Caesar's cipher aka
ROT13

As soon as you learn how it works
you can break it

# Kerckhoff's Principle

„A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.“

# Advantages of having a „key“

Much smaller than One Time Pad (more on that soon!)

„Simple“ form of a secret

Endorse public analysis

Kerckhoff's principle

vs.

Security by Obscurity

Principle extends to open source implementations:

A publicly accessible implementation
is potentially more trustworthy than
closed-source products.

# What is Encryption?

A „Cipher“ is a pair of algorithms **E** and **D**, s.t.

```
E: K x P -> C
D: K x C -> P
```

$$\forall\ m \in P,\ k \in K:\ D(k, E(k, m)) = m$$

P is the set of „plaintexts“, C is the set of „ciphertexts“ and K is the „key space“, the set of all keys. m is often denoted as the „message“, k as the „key“. E and D are denoted as „encryption“ and „decryption“ functions.

When dealing with computers,

K, P and C are typically elements from $\{0,1\}^n$.

This means we represent keys, plaintexts and ciphertexts as their „bit encodings“ consisting of just zeroes and ones.

Do such functions exist?

Certainly!

$$E(m) := m \oplus k$$
$$D(c) := c \oplus k$$

where m, k, and c are of equal length. Then:

$$\begin{aligned} D(k, E(k, m)) &= E(k, m) \oplus k \\ &= m \oplus k \oplus k \\ &= m \end{aligned}$$

```
E(m) := m ⊕ k
D(c) := c ⊕ k
```

is called the One Time Pad.

Is it secure?

What does „secure“ mean?

„Impossible to learn the key k“

$$E(k, m) = m$$

The message is important, not the key!

„Impossible to decrypt the ciphertext“

What about parts of the ciphertext?

„Impossible to learn any character of the plaintext“

Encrypted salary:

$$E(k, s) := \begin{cases} 1\|c & \text{if } s > 100000 \\ 0\|c & \text{otherwise} \end{cases}$$

=> Didn't learn a character, but still got useful information!

„Impossible to learn any meaningful information about the plaintext from the ciphertext“

Actually, not bad, but define „meaningful“ !

„Impossible to compute any function of the plaintext from the ciphertext“

Correct!

# Perfect Secrecy

A cipher (E, D) has perfect secrecy if

$\forall$ m0, m1$\in$P $\forall$c$\in$C and
m0, m1 of equal length

$$Pr(E(k, m0) = c) = Pr(E(k, m1) = c)$$

where k is a fixed random key.

## What does this tell us?

Given an arbitrary ciphertext, the probability that our plaintext is either m0 or m1 is always the same, for any m0 or m1 we choose.

Given the ciphertext of a certain plaintext m and its length, no algorithm exists that can determine any partial information on the message with higher probability than all other algorithms that only have access to the message length (and not the ciphertext).

Is there a cipher that fulfills „Perfect Secrecy?“

The One Time Pad has perfect secrecy.

(cf. http://www.ics.uci.edu/~stasio/fall04/lect1.pdf)

So why not use it all the time?

Key is always as long as the message

and has to be completely random!

Not practical!

Unfortunately, Perfect Secrecy implies:

(cf. http://www.ics.uci.edu/~stasio/fall04/lect1.pdf)

$|K| >= |P|$

=> length of key >= length of plaintext

(in that sense One Time Pad is optimal)

Perfect Secrecy is really hard to achieve, yet it doesn't cover all aspects of security:

$$c0 = m0 \oplus k$$
$$c1 = m1 \oplus k$$

$$\Rightarrow c0 \oplus c1$$
$$= (m0 \oplus k) \oplus (m1 \oplus k)$$
$$= m0 \oplus k \oplus m1 \oplus k$$
$$= m0 \oplus m1 \oplus k \oplus k$$
$$= m0 \oplus m1$$

But how does `m0 ⊕ m1` help?

0x20 (Space) in one message swaps the case of the letter in the other message:

0x20 ⊕ 0x41 (A)

$$
\begin{array}{r}
0010\ 0000 \\
\oplus\ 0100\ 0001 \\
=\ 0110\ 0001 \\
=\ \text{0x61(a)}
\end{array}
$$

=> One message has a space and the other 'A'

Use snippets like „the“ in different positions:

Ciphertext: ... 0x15 0x06 0x01 ...

XOR with 0x74 0x68 0x65 ('the')

= 0x61 0x6E 0x64 ('and')

=> One message has 'the', the other one 'and' at that position. Repeat in different positions with different snippets.

Never reuse a key with the One Time Pad!

# Ciphertext-only attack

We could retrieve the original message only by looking at the ciphertext!

Ciphertext-only is the weakest form of attack.

# Chosen plaintext attack (CPA)

The attacker can choose plaintexts and is given the resulting ciphertexts („encryption oracle“).

# Chosen ciphertext attack (CCA)

The attacker can choose ciphertexts and is given the resulting plaintexts („decryption oracle“).

# Stream Ciphers

Idea:

Reuse XOR construction from One Time Pad

But somehow generate a random „stream“ of bits

Pad generation needs to be:

deterministic

not predictable by uninvolved third parties deterministic

vs.

not predictable

CONFLICT !!!

True randomness is not predictable

but by definition it is also not deterministic!

We need something „seemingly random“.

More precise: We need a function G(k) that generates data given a key k whose output (without knowing k) is indistinguishable from true random data.

Pseudo-random (Number) Generator (PRNG):

A function $G: K \rightarrow \{0, 1\}^n$ , where
$K = \{0, 1\}^m$ and n typically much larger than m.

We can then define a Stream Cipher (E, D) by

$$E(k, m): m \oplus G(k)$$
$$D(k, c): c \oplus G(k)$$

Then:

$$\begin{aligned} D(k, E(k, m)) &= E(k, m) \oplus G(k) \\ &= m \oplus G(k) \oplus G(k) \\ &= m \end{aligned}$$

How „small“ can we afford our key k to be?

Example: $k \in \{0, 1\}^3$

Given a ciphertext c, just try $c \oplus G(k_i)$ for all $k_i \in \{000, 001, 010, 011, 100, 101, 110, 111\}$

The key must be large enough
to render brute force attempts
„computationally infeasible“

Typically, space/time in the range
of >= 2^60 bits/steps
is considered infeasible,
even for government-size adversaries

Can there be a Stream Cipher

with Perfect Secrecy?

No,

since the key k is in general much shorter than the messages in our space of plaintexts, which are possibly unbounded.

We need a different notion of „security“.

How about we use Math.random()

with the key being the seed ?

Not a good idea.

It is possible with „manageable“ effort to predict future outputs.

=> We can retrieve the ciphertext at some point

# Predictable PRNG:

Given n >= 0 output bits of a PRNG, an „efficient“ algorithm exists that allows to predict output bit n+1 with a probability > 1/2.

A PRNG where no such algorithm exists for any n >= 0 is called unpredictable.

Try to find a security definition for our PRNG.

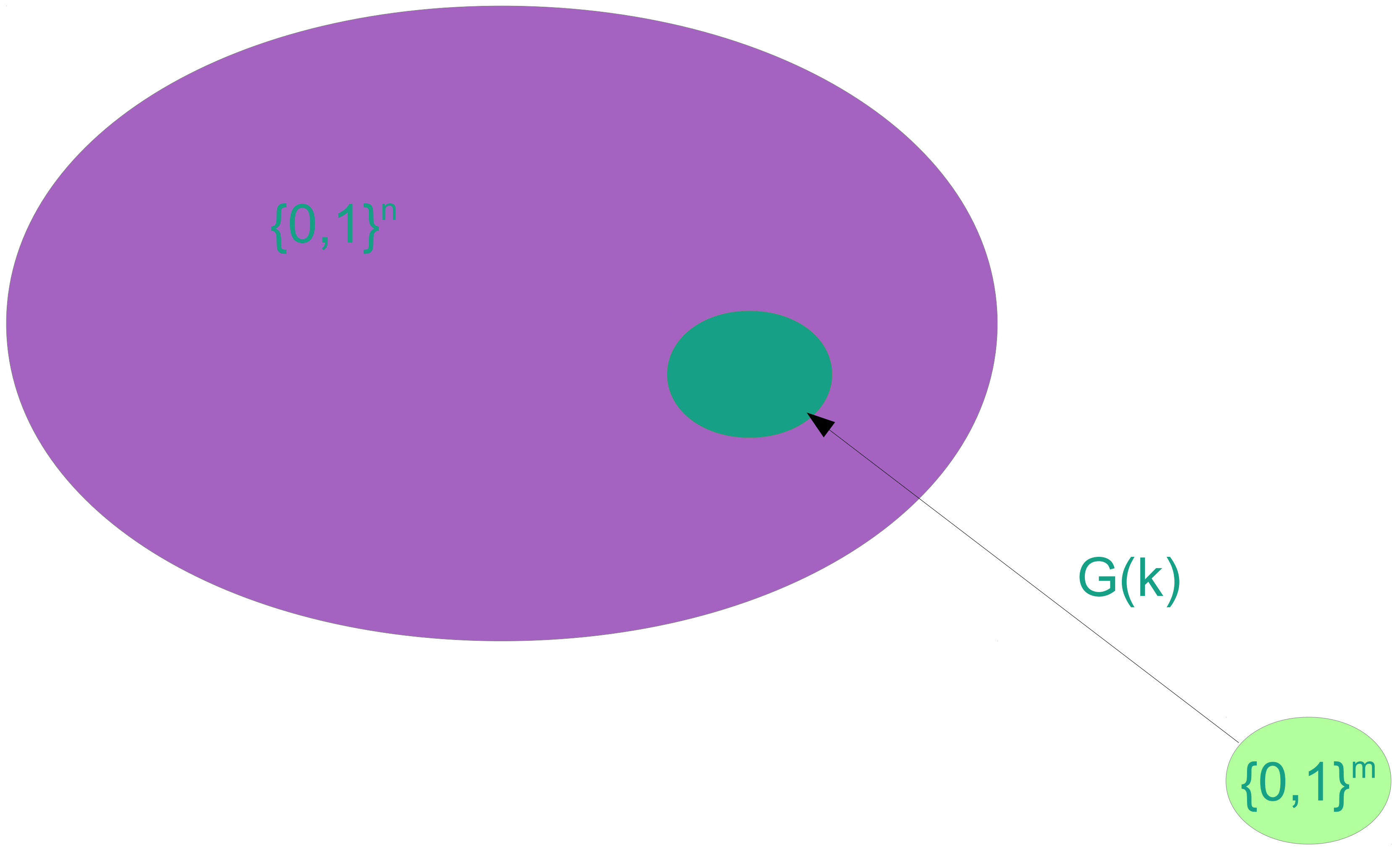
What does it mean for a PRNG to be secure?

„A PRNG is secure if its output
is indistinguishable from
a real random function“

With enough effort we can always do:

Assume k is 128 bits long (i.e. n = 128)
=> 2^128 choices for k
=> less than or equal to 2^128 distinct values for G(k)
=> but P=C=$\{0,1\}^n$ where m much larger than 128
=> a lot of elements from $\{0,1\}^n$ are never generated
=> use this knowledge to tell PRNG from real random

**stream.ciphers.**



„A PRNG is secure if its output
is computationally indistinguishable from
a real random function“

A PRNG is secure

iff

it is unpredictable

We'd like:

PRNG is secure

=> Stream Cipher constructed from PRNG is secure

But under which notion of security?

Idea:

Relax requirements for Perfect Secrecy

# Semantic Security

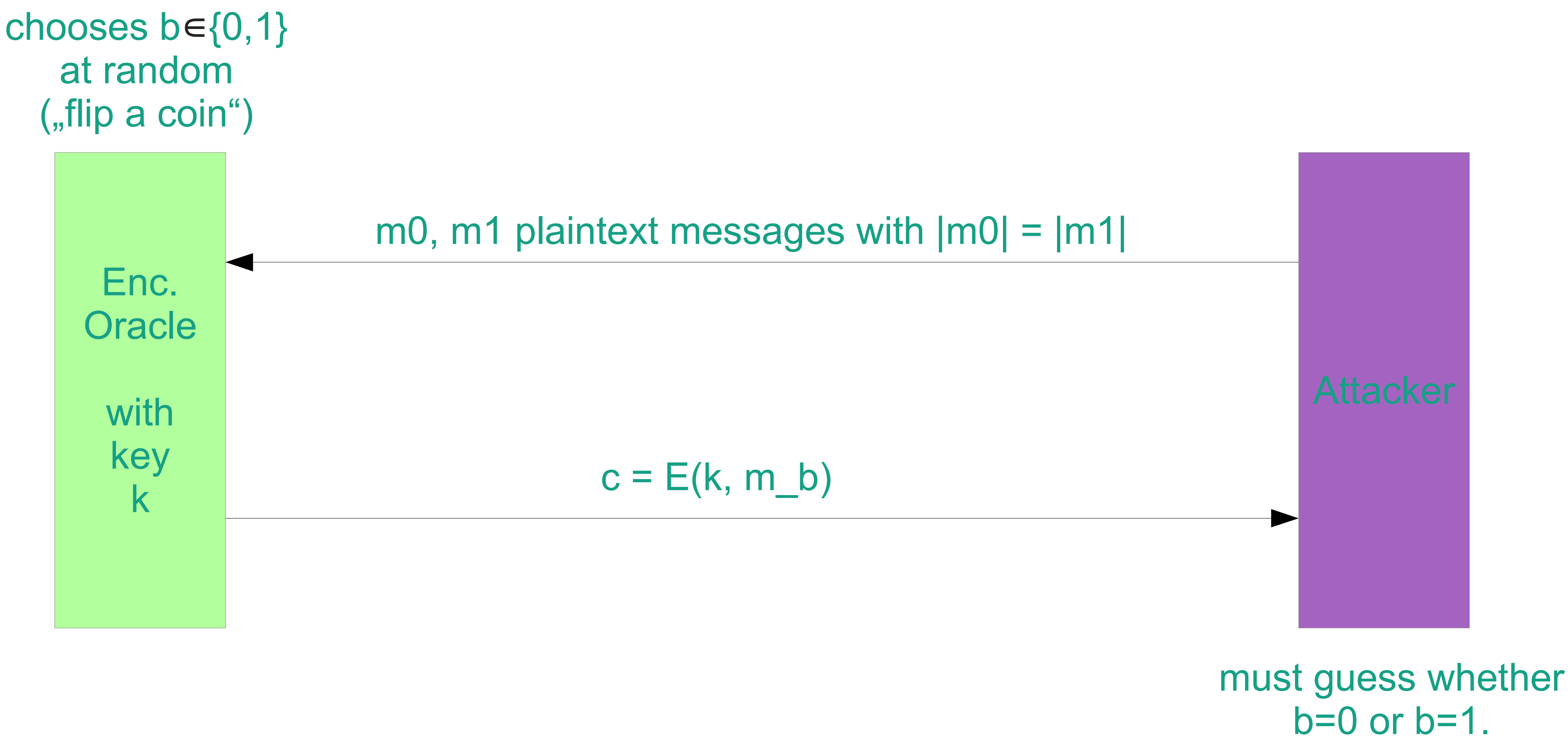
A Cipher is semantically secure if given the ciphertext of a certain plaintext m and its length, no computationally feasible algorithm can determine any partial information on the message with higher probability than all other algorithms that only have access to the message length (and not the ciphertext).

# Semantic Security vs. Perfect Secrecy

Perfect secrecy: Impossible to compute any function of the plaintext from the ciphertext.

Semantic security: Computationally infeasible to compute any function of the plaintext from the ciphertext.

# Security proofs can be nicely visualized using a „game“



Attacker wins the game if able to predict b „with a better success rate than guessing randomly“

If we denote
W_0 => { Attacker outputs 1 when b = 0 }
W_1 => { Attacker outputs 1 when b = 1 }

then
| Pr(W_0) – Pr(W_1)|  is not negligible

When guessing using „coin flipping“

$Pr(W_0) = 1/2$

$Pr(W_1) = 1/2$

so $| Pr(W_0) - Pr(W_1) | = 0$

(As low as it gets, „attacker has no clue“)

Note:

Chosen Plaintext Attack

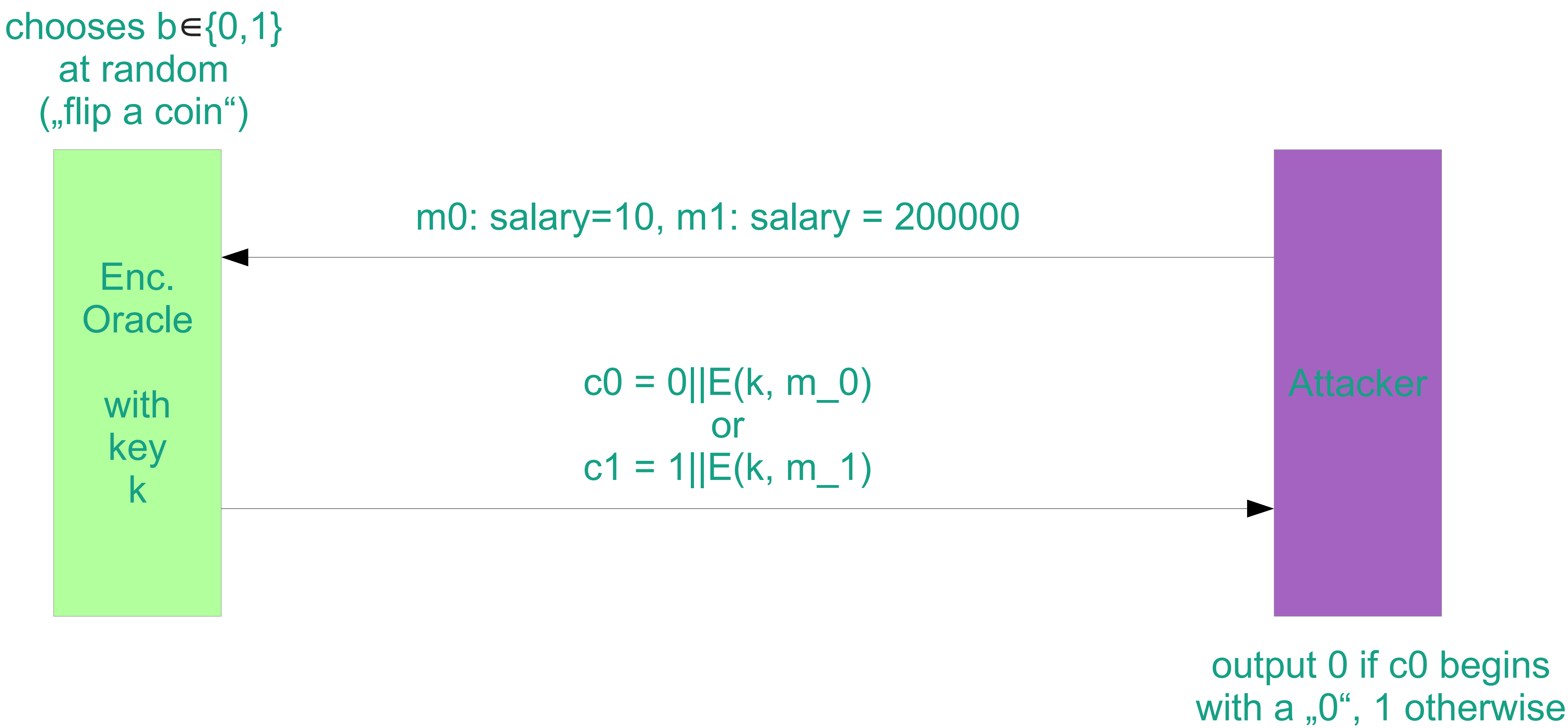
But model just allows to do one single query

(Will be extended to more powerful attackers)

Encrypted salary example:

$$E(k, s) := \begin{cases} 1||E(k, s) & \text{if } s > 100000 \\ 0||E(k, s) & \text{otherwise} \end{cases}$$

# Security „game“ for salary example



$$Pr(W\_0) = 0$$
$$Pr(W\_1) = 1$$

Attacker is just never wrong!

so $| Pr(W\_0) - Pr(W\_1) | = 1$

(As high as it gets, „completely broken“)

It can be proven:

If PRNG is secure

=> Stream cipher constructed from PRNG is semantically secure.

# Does Semantic Security cover all relevant aspects of „being secure“ ?

$$c0 = m0 \oplus G(k)$$
$$c1 = m1 \oplus G(k)$$

$$\begin{aligned}
&\Rightarrow c0 \oplus c1 \\
&= (m0 \oplus G(k)) \oplus (m1 \oplus G(k)) \\
&= m0 \oplus G(k) \oplus m1 \oplus G(k) \\
&= m0 \oplus m1 \oplus G(k) \oplus G(k) \\
&= m0 \oplus m1
\end{aligned}$$

(analogous to One Time Pad)

Never reuse a key with a Stream Cipher!

Is our security model broken?

No, we explicitly allowed just one query!

But as the threat is valid, we need to extend our model to include the capability to query more messages.

Long-term keys
can be used with a per-encryption nonce

# Encryption of a transaction

Amount:000323 €

bf99EIk44fmUy
000323

Flip some bits – get rich!

Would like to protect ourselves against these attacks

-> Semantic Security is not enough

# Real-world examples of Stream Ciphers

RC4

Very fast
Very old

A lot of attacks exist – while not completely broken it shouldn't be used in new designs.

Problem: RC4 is the only cipher not vulnerable to the BEAST attack in SSL/TLS versions <= 1.0

(cf. http://www.infoworld.com/sites/infoworld.com/files/pdfe/BEAST_Duong_Rizzo.pdf)

# RC4

Even bigger problem: A feasible attack for RC4 exists

(cf. http://www.isg.rhul.ac.uk/tls/RC4biases.pdf)

# RC4

Don't use RC4

Will hopefully vanish
with more wide-spread adoption of TLS 1.2

# RC4

Impractical since you can't reuse the key.

Every key is literally a one-time key.

# eSTREAM ciphers

## Software:

HC-128
Rabbit
Salsa20
SOSEMANUK

## Hardware:

Grain
MICKEY
Trivium

# eSTREAM ciphers

At least you may reuse the key now because of the addition of an IV/nonce.

But you **still** need to be careful not to reuse any IVs/nonces!

(More about IVs/nonces later)

# eSTREAM ciphers

Salsa20 could be added to future versions of TLS

# Key Generation

„I want to use a Cipher,
but how do I generate a key?“

# key.generation.

With modern ciphers, it's quite simple:

128 bit key => generate 16 bytes with secure PRNG

256 bit key => generate 32 bytes with secure PRNG

Done!

Older Ciphers (notably DES) had „weak“ keys.

=> Use library functionality to generate keys.

If the language/library offers it, it is never a bad idea to use specific key generation tools.

# Secure Pseudo-random Number Generators

They are arguably the most important part
of cryptographic algorithms

If the PRNG is broken, crypto becomes trivially predictable

# The Problem

Computers are inherently deterministic.

Hardware PRNGs – do we trust them?

A high-level description of common software PRNGs:

Take a small amount of „real“ random data

Apply a hash function to that block of data
-> return the result

Iteratively apply the hash to previous result
-> return the result

If the underlying hash function is secure,
it is infeasible to predict the output

as long as the initial value (the „seed“) is
kept secret.

# Problems

PRNGs have received much less cryptanalysis than other primitives

Different OS use different algorithms

Different libraries use different algorithms again

Most algorithms seem rather ad-hoc

# Problems

Linux PRNG is almost 20 years old!

Apple, Microsoft ?

Standards do exist

but are not widely adopted (Fortuna)
(cf . http://en.wikipedia.org/wiki/Fortuna_(PRNG))

or no longer trusted (NISTs SP 800-90 A, B, C)
(cf. http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf)

because of backdoors
(http://en.wikipedia.org/wiki/Dual_EC_DRBG)

(although the hash-based PRNGs are good!)

Where does the „real“ random data for the seed come from?

(http://www.pinkas.net/PAPERS/gpr06.pdf)

Mouse/keyboard activity
disk IO
network IO
specific interrupts.

/dev/random on *nix systems

Can't we use this all the time?

No, the pool is quickly drained, and then the device blocks until further data is available

Therefore, we only seed with /dev/random

and then „stretch" the data

typically by using hash functions

/dev/urandom (*nix)
CryptGenRandom (Windows)

OpenSSL
Java SecureRandom
...

While things seem relatively simple,
the devil is in the details

(cf. http://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html)
(cf. http://martinbosslet.de/blog/2013/08/21/openssl-prng-is-not-really-fork-safe/)

What does this mean for me?

Forget that Math.random ever existed

Use secure PRNGs exclusively

Either OS-based such as /dev/urandom
or software-based such as SecureRandom

Let library maintainers handle the details

What does this mean for me?

Most important of all:

Do not roll your own scheme,
highly unlikely that you
improve the situation

# RELAX.

Finally, some code.

# Hex encoding

Built-in, but clumsy to use.
Make your own module

… or use krypt!

# Base64 encoding

Built-in, but no support for streaming.

If you need streaming Base64, use krypt.

# Secure Random Numbers

Aside:

Generate an „unguessable“ token

How many bytes is „unguessable“ ?

# Secure Random Numbers

10 bytes => 2^80 possible values

=> Birthday Paradoxon says
collision after roughly 2^40 attempts

doable!

(more on the Birthday Paradoxon when we talk about Hash collisions)

# Secure Random Numbers

Security margin of 2^80:

We need 2^160 possibilities => 20 bytes!

# Secure Random Numbers

Careful:

A hex string of 20 bytes has only an entropy equivalent to 10 „real“ bytes

=> Collision after 2^40 instead of 2^80

# Secure Random Numbers

Careful:

A Base64 string of 20 bytes has only an entropy equivalent of (20 * 3/4) 15 „real“ bytes

# Secure Random Numbers

On *nix, you may also choose

/dev/urandom

Otherwise, use SecureRandom
(http://www.ruby-doc.org/stdlib-2.1.0/libdoc/securerandom/rdoc/SecureRandom.html)

Why not OpenSSL?
(cf. http://martinbosslet.de/blog/2013/08/21/openssl-prng-is-not-really-fork-safe/)

# Symmetric Key Generation

Create them yourself with SecureRandom

Or better:

Use Cipher#random_key and Cipher#random_iv

http://www.ruby-doc.org/stdlib-2.1.0/libdoc/openssl/rdoc/OpenSSL/Cipher.html

# Stream Ciphers

RC4 is all we got with OpenSSL :/

No eStream ciphers...

# Stream Ciphers

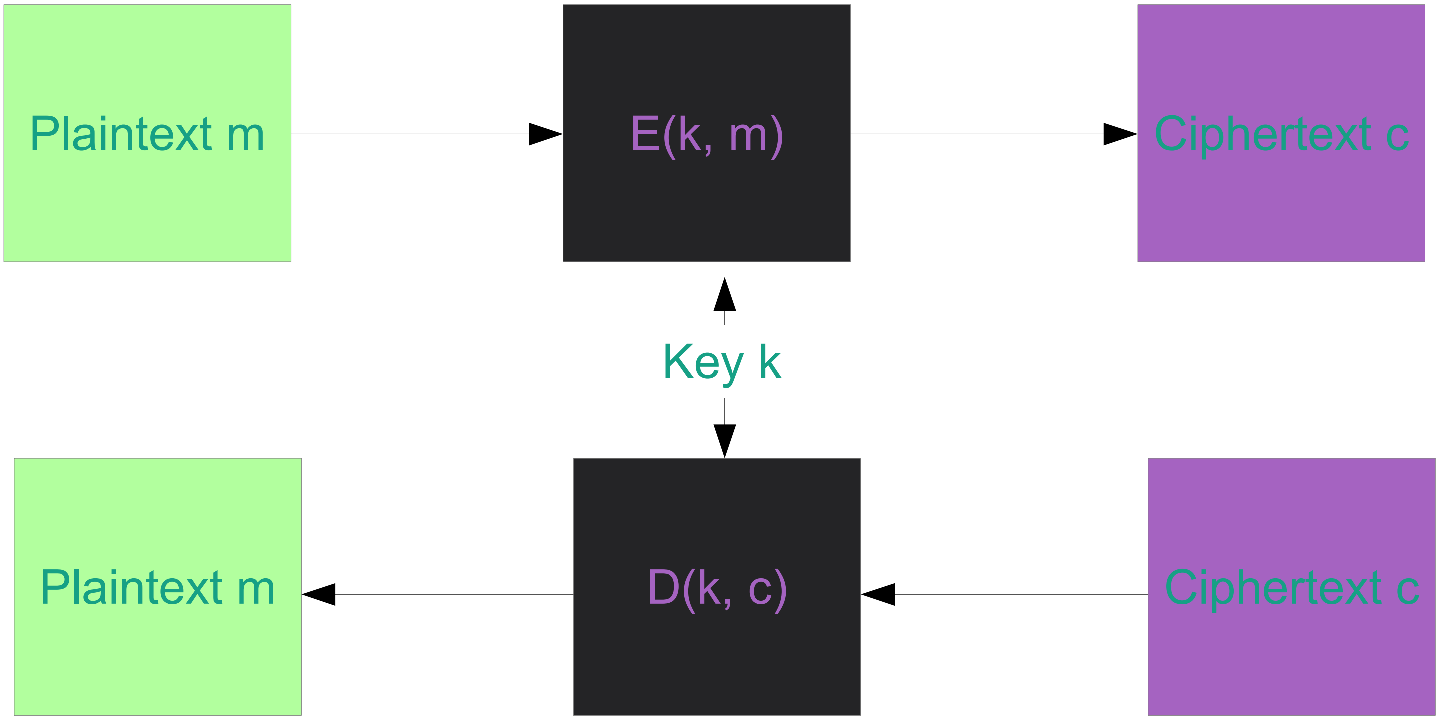
Worth repeating:

Don't ever reuse the key with RC4

Don't ever reuse an IV with nonce-/IV-based eSTREAM stream ciphers like Salsa20.

(You won't, since right now you can't ;)

# Block Ciphers



Instead of XORing a „key stream“ to the plaintext,

the message is split into n equal-length blocks $m\_0, m\_1, ..., m\_{n-1}$.

Each block is then encrypted using $E(k, m_i) = c_i$ yielding the ciphertext $c = c_0 c_1 .... c_{n-1}$

# Pseudo-random Permutation

`E: K x X -> X`

where E is „efficiently“ computable
E(k, x) is bijective (and thus invertible)

Inversion algorithm `D: K x X -> X`
is again efficiently computable and

`∀ x∈X, k∈K: D(k, E(k, x)) = x`

We call such a pseudo-random permutation

a Block Cipher.

# Can there be a Block Cipher

# with Perfect Secrecy?

No,
again the key is generally much smaller than the plaintext space.

But we can reuse our relaxed notion of Semantic Security.

Typically, we want to reuse a key k when using block ciphers.

Semantic Security for key used more than once

=

CPA security

=

IND-CPA

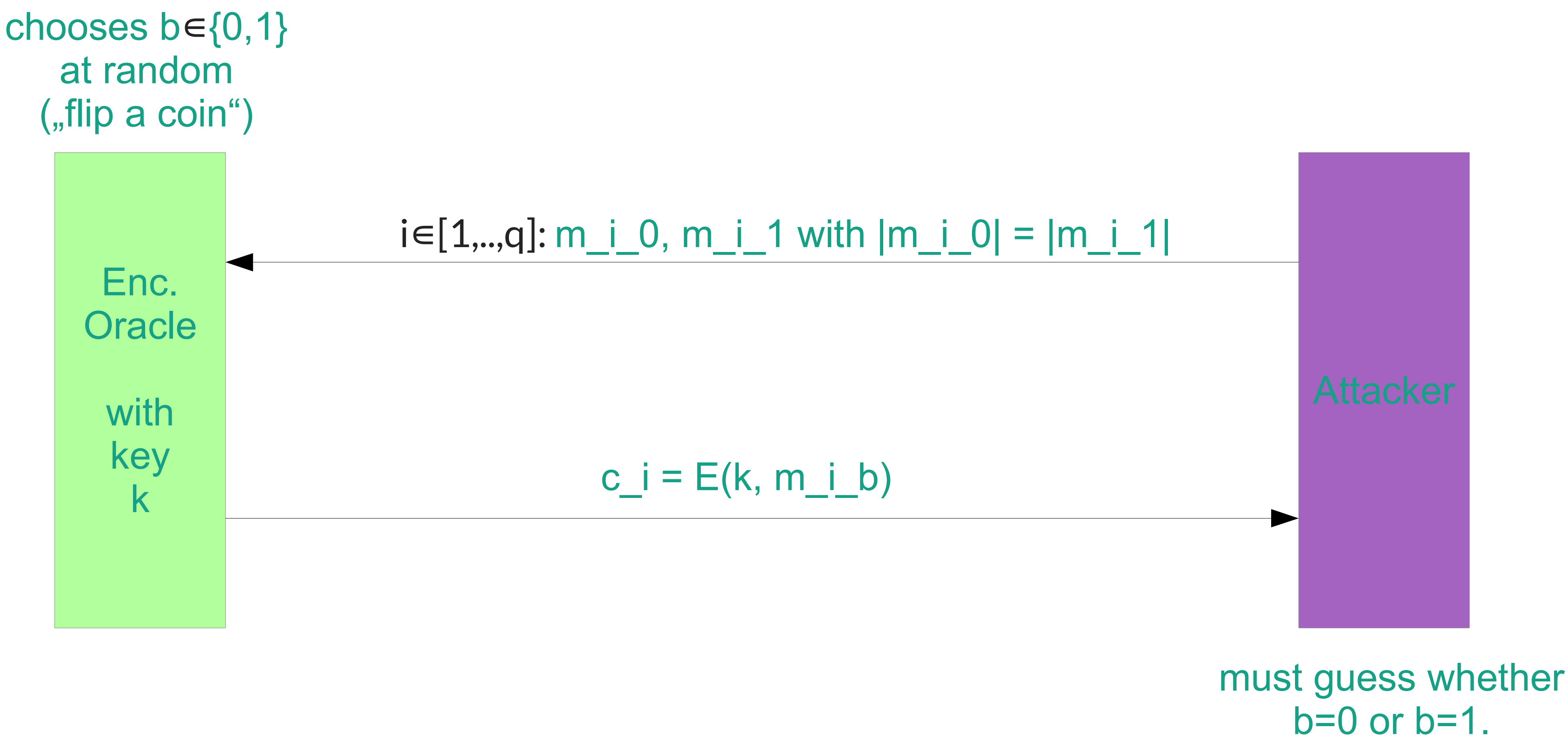
(„Indistinguishability under a Chosen Plaintext Attack“)

# Indistinguishability:

An „encryption oracle" is fed pairs of chosen plaintext messages m0, m1 and randomly returns either E(k, m0) or E(k, m1).

=> IND-CPA holds if we cannot tell if the result is the encryption of m0 or m1 with probability higher than 1/2.

# Security „game“ for IND-CPA with key reuse



# Real-world examples of Block Ciphers

# DES – The „Data Encryption Standard“

DES

Remains unbroken until today

But its short key sizes render it subject to brute force

DES

Shouldn't be used anymore

Neither its variants 3DES, DES-X – while sufficiently secure they are too slow, better alternatives exist

# AES – The „Advanced Encryption Standard“

# AES

First cryptographic primitive
that evolved from a public competition

Originally named Rijndael after its inventors

AES

Unbroken and widely adopted

De facto encryption standard today

Twofish & Serpent

AES „runner up“ algorithms

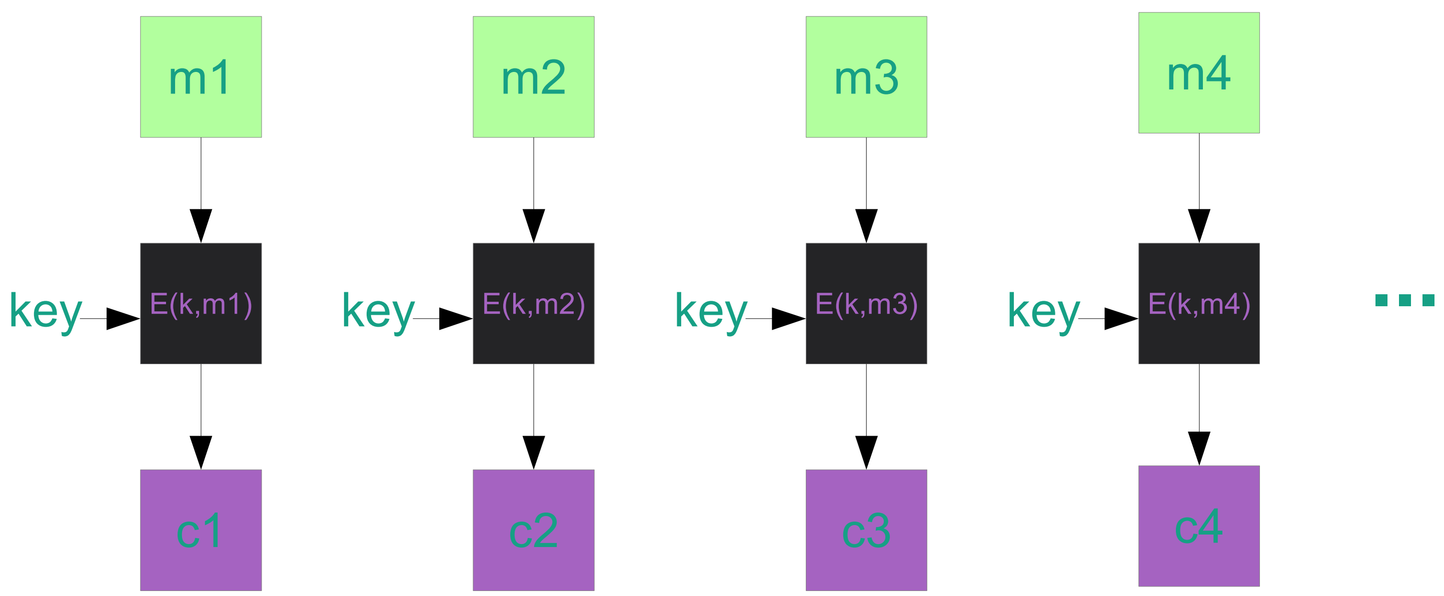
Unbroken and by some even considered more secure than AES

However much less adoption

# Modes of Operation

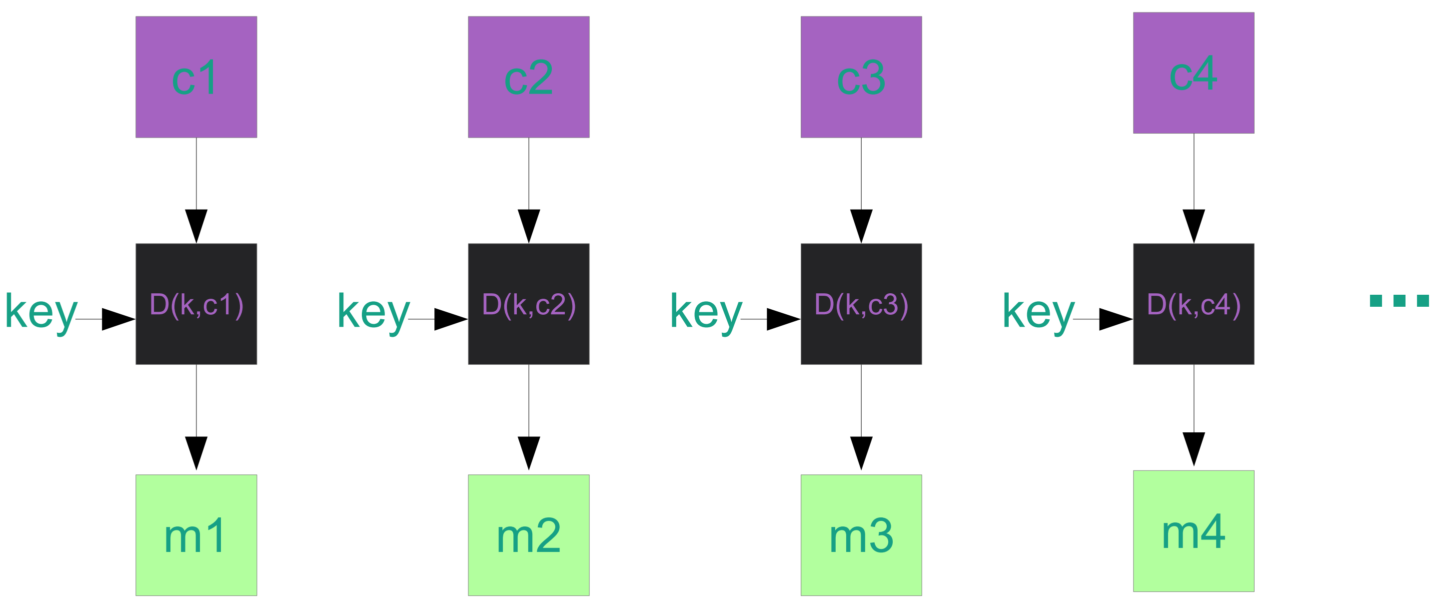
# ECB – Electronic Codebook mode

## Encryption



# ECB – Electronic Codebook mode

## Decryption

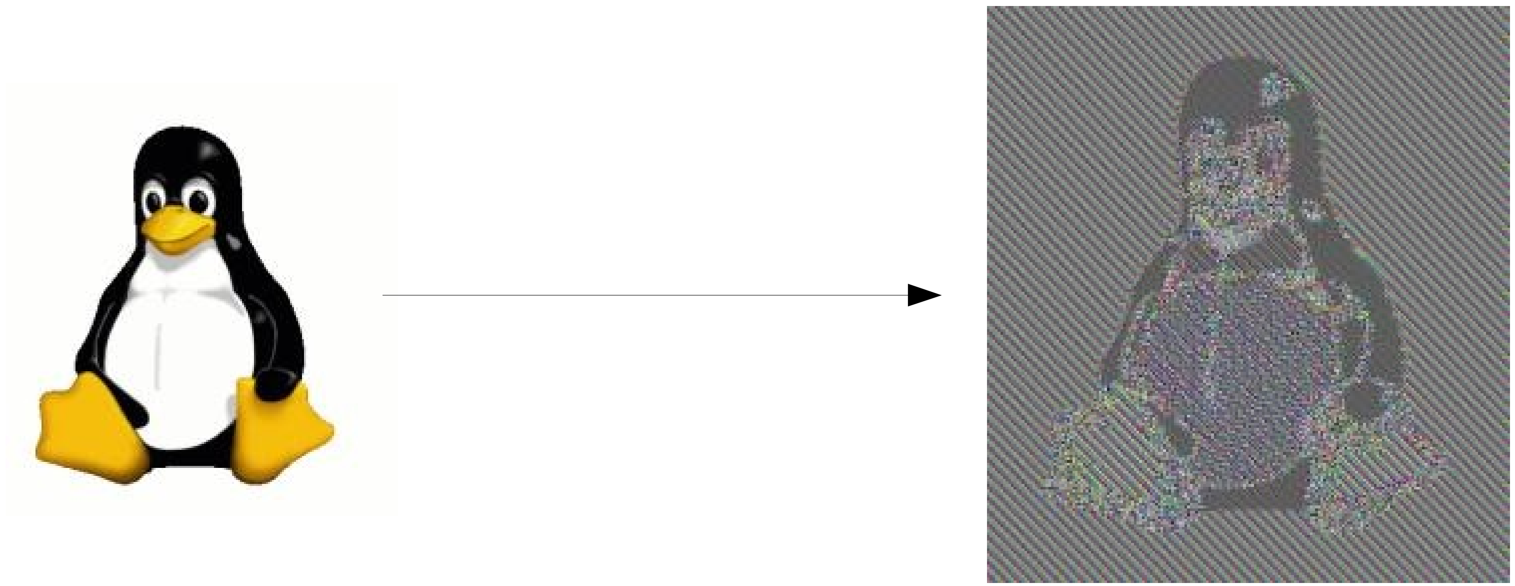


If $m_i = m_j$ for some $i \neq j$

then

$E(k, m_i) = E(k, m_j)$

=> ECB does not hide patterns in the plaintext!



Probably the most (in)famous image used when talking about crypto.
It's incredibly lame, I know, but it always drives home the point!

ECB encryption is clearly distinguishable from a real random permutation

=> ECB mode is not IND-CPA!

If ECB is so bad, why does it even exist ???

Perfectly fine if plaintext size never exceeds block size of the block cipher

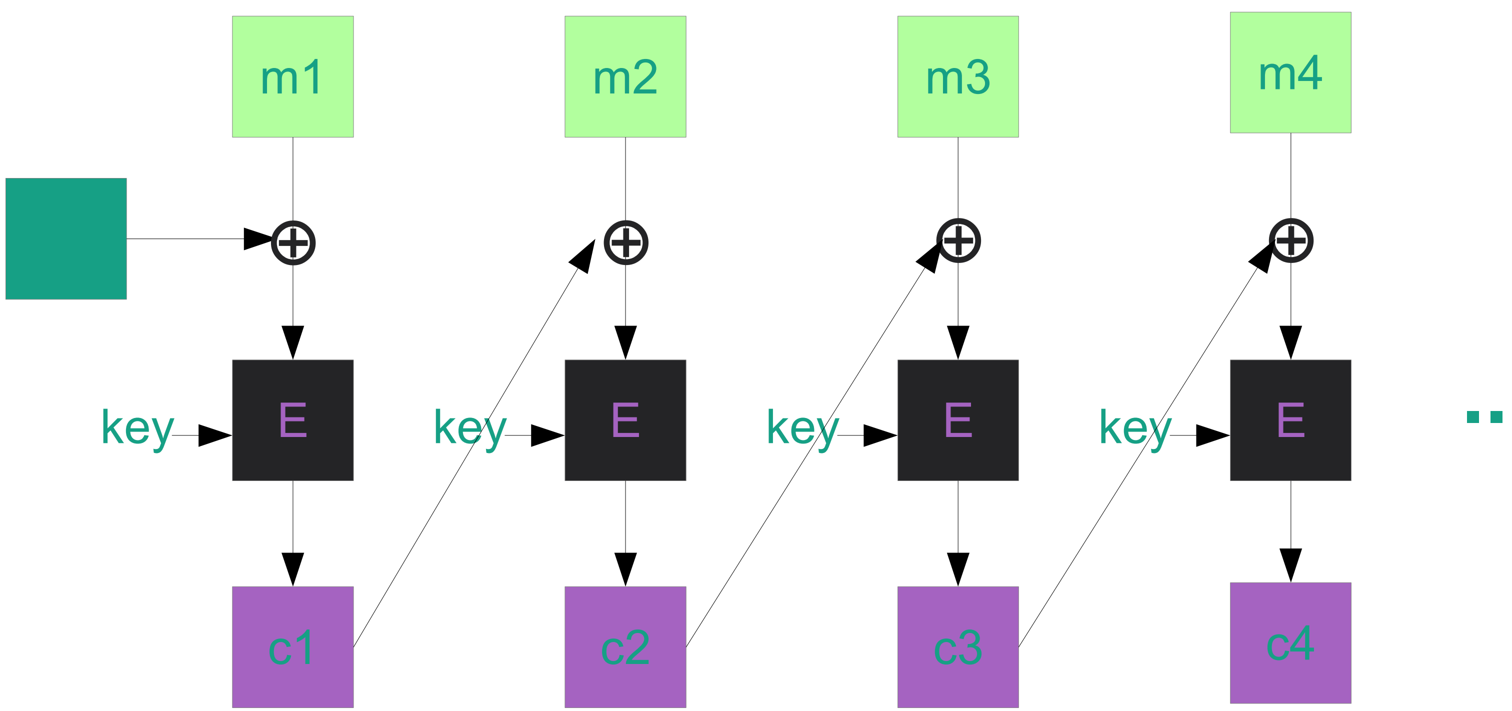
ECB represents the „raw“ algorithm, all other modes can be constructed from it

But, unless you have a very good excuse:

don't use ECB!

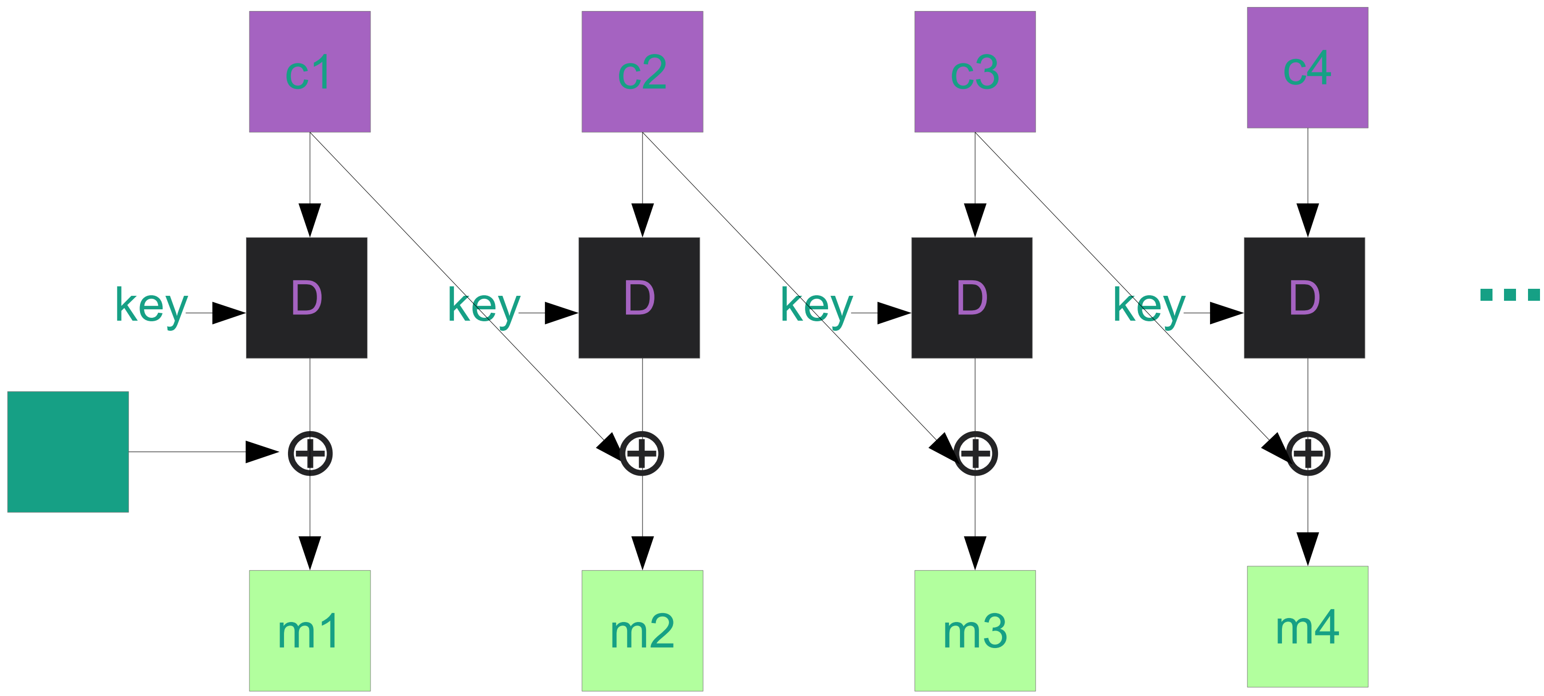
# CBC – Cipher Block Chaining mode

## Encryption



# CBC – Cipher Block Chaining mode

## Encryption



Encryption:

$$c_i := E(k, m_i \oplus c_{(i-1)})$$

$$c_0 := IV$$

Decryption:

$$m_i := D(k, c_i) \oplus c_{(i-1)}$$

$$m_0 := IV$$

$$D(k, c_i)$$

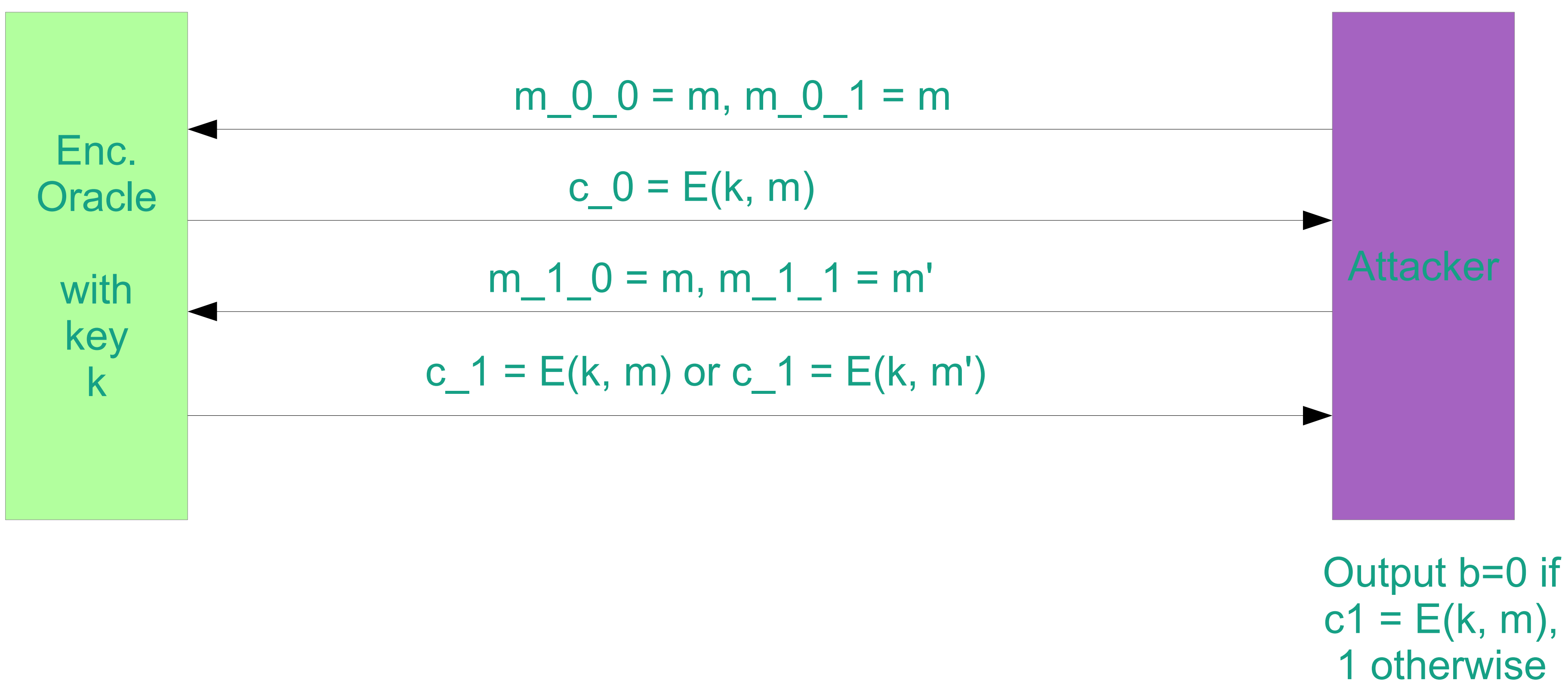
$$= D(k, c_i) \oplus c_{(i-1)}$$

$$= D(k, E(k, m_i \oplus c_{(i-1)})) \oplus c_{(i-1)}$$

$$= (m_i \oplus c_{(i-1)}) \oplus c_{(i-1)}$$

$$= m_i$$

# A cipher that always outputs the same ciphertext for the same message m cannot be IND-CPA



# IV – The Initialization Vector

We know: If E(k, m0) = E(k,m1) for m0 = m1, then the cipher cannot be IND-CPA.

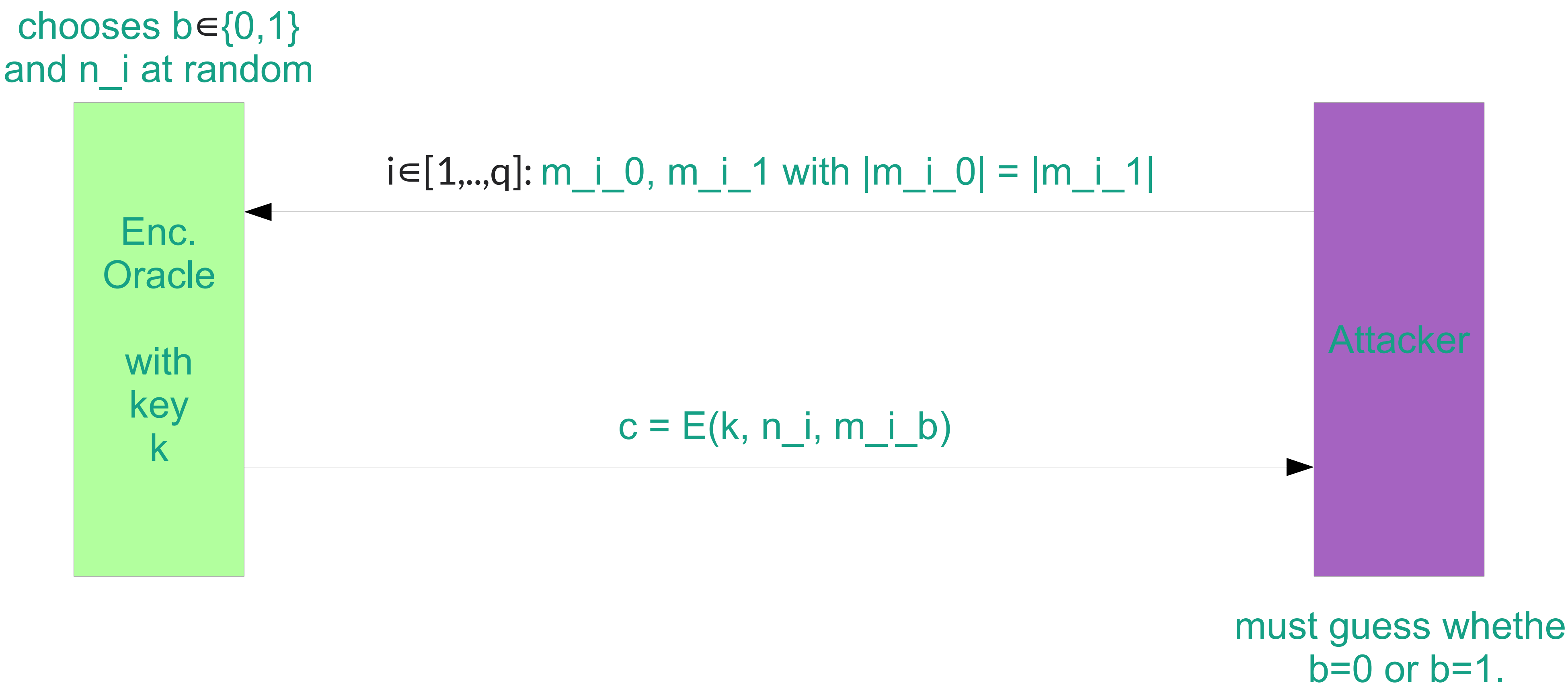
If we want to use a key k multiple times, it must output a different cipher text each time.

This is the purpose of the IV.

Need to extend our security model by
the notion of a „Nonce“ (Number used once)

An „encryption oracle“ is fed pairs of chosen plaintext messages and nonces (m0, n0), (m1, n0) and randomly returns either E(k, m0, n0) or E(k, m1, n0).

# Security „game“ for IND-CPA with nonces



# Choosing the IV

Secure pseudo-random value

Not predictable!

Never, ever the key!

A predictable IV is what was exploited in the BEAST attack on TLS!

# IV – choosing same as the key

(cf. http://www.cs.berkeley.edu/~daw/my-posts/key-as-iv-broken)

Don't expose the key if it's not required!

Even if it looks safe, do not play with fire!

What if my message is not
an exact multiple of the block size?

Padding

# Padding

We use a „padding scheme“ to fill the last block until it meets the cipher block size.

# Padding

Padding schemes are ad-hoc and do not improve the security

However, if implemented badly, they can diminish it

# PKCS#7 (PKCS#5) padding

AES-128 (block size = 16 bytes)

1 byte added: ... 01
2 bytes added: ... 02 02
3 bytes added: ... 03 03 03
...
15 bytes added: ... 0f 0f 0f ... 0f (15x)
0 bytes added: ... 10 10 10 ... 10 (16x)

# Zero Padding

Same as PKCS#7 padding, just always pad with zeroes

Ambiguous:
„Is it padding or does this 0 belong to the message?“

# ISO/IEC 7816-4 padding

c1...cn 01 (1 byte added)
c1...cn 38 fe 7b 04 (4 bytes added)
c1...cn da 02 3d e4 66 17 ae 53 ... 10 (16 bytes added)

The last byte of the padding indicates how many bytes were added, remaining padding bytes are arbitrarily chosen. If message is exact multiple of the block size, add another block of padding.

Which padding should I choose?

It doesn't really matter, but PKCS#7 seems to be the most widely adopted padding scheme.

Why can bad padding implementation
hurt the security?

(cf. http://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf)
(cf. https://www.usenix.org/legacy/event/woot10/tech/full_papers/Rizzo.pdf)
(cf. http://www.skullsecurity.org/blog/2013/padding-oracle-attacks-in-depth)

Revealing information whether a padding is valid or not
may completely break the cipher!

Information may be revealed

explicitly through
error messages (Java's BadPaddingException)

implicitly through
side channels like measuring execution time - vulnerable implementations return early for bad paddings

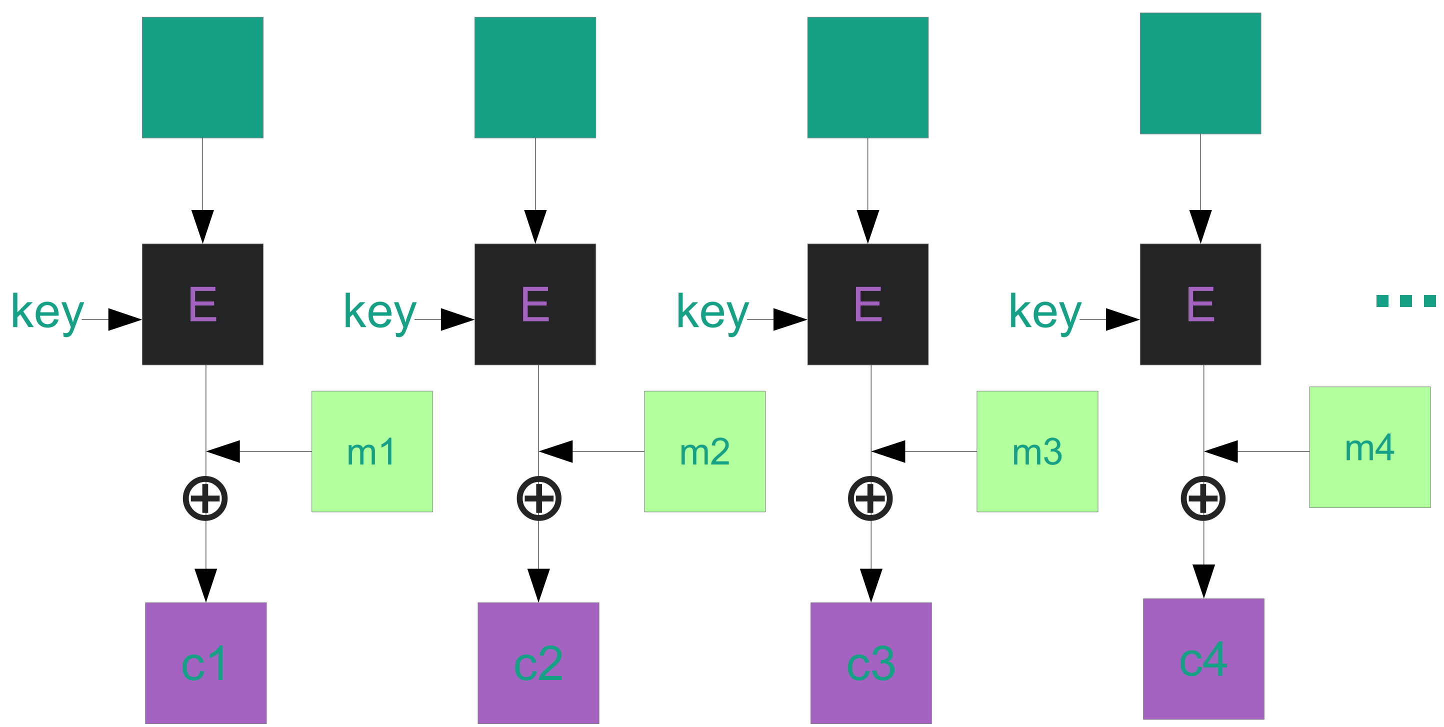
# Rule of Thumb

Do not reveal any information
if your crypto code failed

Use a Development/Production switch
to enable debugging during development

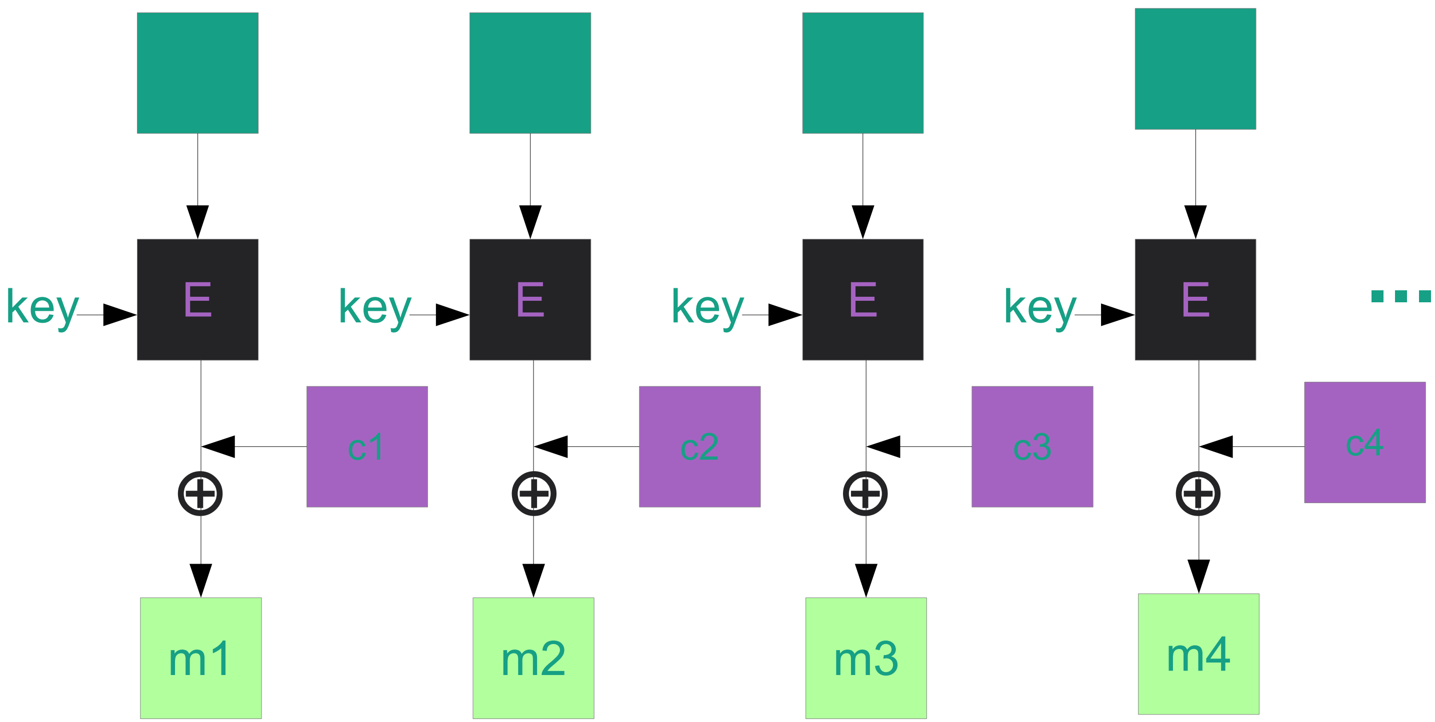
# CTR – Counter mode

## Encryption



# CTR – Counter mode

## Decryption



CTR mode effectively turns the block cipher into a stream cipher.

Only the encryption function is needed.

The cipher is used to encrypt a block with an increasing counter, which is then XORed to the plaintext/ciphertext.

It is crucial that the „counter blocks“ never repeat!

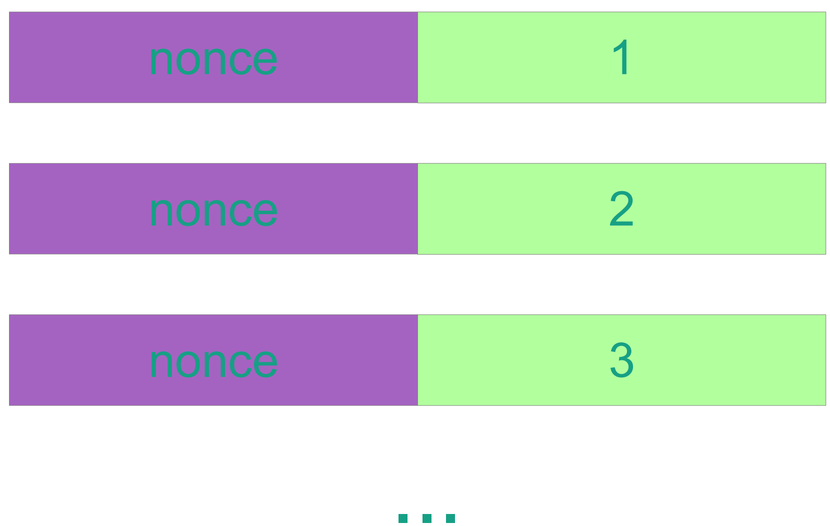
If for $i \neq j$

$$c_i = m_i \oplus E(k, n)$$
$$c_j = m_j \oplus E(k, n)$$

$$\Rightarrow c_i \oplus c_j = m_i \oplus m_j$$

# Solution:

Split the „counter block“
into a nonce part and a counter part



Nonce may even be predictable

With many, many ciphertext blocks observed
the probability for attackers to learn the key increases.

For AES, the exact bound depends on the cipher mode,
e.g. for CBC the key should be changed after 2^48
blocks encrypted. This is equivalent to 2^52 bytes,
roughly 1 Petabyte of data.

Still, it is considered good practice to regularly change keys!

What key size should I choose?

Generally, the longer the better

But AES-128 is just fine

AES-192/-256 of course are, too,
but they are also less performant

| | CBC | vs. CTR |
|---|---|---|
| mode of operation | block cipher | stream cipher |
| parallelizable | no | yes |
| dummy padding block | yes | no |
| ciphertext expansion | yes | no |
| change 128 bit key after n blocks | 2^48 | 2^64 |

What about OFB and CFB modes?

Forget about them.

If you need a stream cipher, either
choose CTR mode or a „real“ stream cipher

When should I choose a Stream Cipher?
When a Block Cipher?

There is no „better than the other“. But if you may choose freely, please be patient -
a recommendation will be given soon :)

So should I choose CTR or CBC?

Answer: Neither.

# Recap: Encryption of a transaction

Amount:000323 €

bf99EIk44fmUy

000323

Flip some bits – get rich!

# Malleability

This works with CBC and CTR as well!

We can flip bits while the encrypted ciphertext is in transit and the recipient will never know

This phenomenon is called ciphertext malleability, the ciphertext is malleable.

The IND-CPA security model does not cover these kinds of attacks

=> we need to tighten our security model

=> we need to be able to tell whether a given ciphertext is „authentic“

=> need further cryptographic primitives

# Other attacks on block ciphers

Linear cryptanalysis
Differential cryptanalysis

Side channel attacks such as
Power Analysis
Timing attacks
...

=> cf. literature recommendations

Linux PRNG:

- http://eprint.iacr.org/2012/251.pdf
- http://eprint.iacr.org/2006/086.pdf
- http://bit.ly/LaVify

UUIDs vs. (secure) Random numbers:

- http://bit.ly/LaV5sJ

Disk encryption (AES-XTS)

- http://csrc.nist.gov/publications/nistpubs/800-38E/nist-sp-800-38E.pdf
- http://www.cs.berkeley.edu/~daw/papers/tweak-crypto02.pdf
- http://www.cs.ucdavis.edu/~rogaway/papers/offsets.pdf

# CODE.

# Block Ciphers

All recent versions of OpenSSL provide ECB and CBC.

The newer ones also provide CTR mode.

# Block Ciphers

The default padding used in OpenSSL is PKCS#7 padding. It's either that or no padding at all.

If you need to have another padding, use the „no padding“ mode and implement padding manually.

# Block Ciphers

The IV can be safely published.

But how do we communicate it?

# Block Ciphers

Ad-hoc scheme: Send

IV || ciphertext

# Block Ciphers

Send it as out-of-band parameter

E.g. web service that expects both ciphertext and IV as parameters

# Block Ciphers

Use custom ASN.1 schema

Use CMS EncryptedData

# We have never talked about the actual algorithms.

## Symmetric crypto: Confusion & Diffusion

„Kill mathematical / statistical properties“
XOR / AND / Bit shifts / Carry bits
Analysis: mostly statistics & probability

vs.

## Asymmetric Crypto: mathematical structures

Base security on known hard problems
Domain objects are algebraic constructs (Finite fields, Elliptic Curves)
Highly structured

# Hash functions

Motivation:

We need to be able to tell that a message presented to us has not been changed during transit

Sender authors & sends document

|

Transit (Many things may happen)

|

Receiver receives document

Idea:

A deterministically computable „checksum“.

$$h : \{0, 1\}^n \rightarrow \{0, 1\}^m$$
$$h(x) = y$$

# Message Integrity

If we want to check the integrity of a message given an existing „checksum“ $y$, we compute $h(x) = y'$ and compare if $y = y'$ holds

# Requirements

0. We would want m to be much smaller than n in

$$h : \{0, 1\}^n \text{ -> } \{0, 1\}^m$$

h(x) = x is a perfectly fine „checksum“, but not really practical

Requirements:

1. It should be impossible to derive the actual message from the „checksum“

Say somebody intercepts E(k, m) and h(m), then being able to learn m from just h(m) defeats the purpose of encryption.

Requirements:

2. Given a pair x and $y=h(x)$ it should be impossible to find another x' with $h(x') = y$

If this were possible, somebody could forge a message while the checksum would still confirm the message's integrity

Requirements:

3. It should be impossible to find two arbitrary x, x' with $h(x) = h(x')$

Imagine using the checksum for generating database primary key values.
Two colliding values would cause an error!

Requirements:

4. Given just E(k,m) and h(m), we don't want anyone to be able to produce valid pairs E(k, m)' and h(m)' (e.g. by flipping bits etc.)

This is essentially what we need to defeat ciphertext malleability!

I have to disappoint you:

we'll have to postpone 4. to the next two chapters!

But 0., 1., 2. and 3. define

what is called a secure hash function
or
message digest

# Hash function

$$h: \{0, 1\}^{\infty} \text{ -> } \{0, 1\}^{n}$$

We allow a potentially unbounded message space, while the function values („hash“, „digest“) are of bounded length.

# Secure hash function

A hash function is called a secure hash function if it satisfies the following requirements:

# 1. Preimage Resistance

Given a hash function h and a $y = h(x)$,
it should be computationally infeasible to find the corresponding x.

# 2. Second Preimage Resistance
# or
# Weak Collision Resistance

Given a hash function h and a message x,
it should be computationally infeasible to find a different x' such that h(x') = h(x).

# 3. Collision Resistance

Given a hash function h,
it should be computationally infeasible to find
any two x, x' such that h(x') = h(x).

It turns out that Collision Resistance is the strongest of the three requirements.

It can be shown that collision resistance already implies preimage and second preimage resistance.

Idea:

Why not use checksum algorithms like CRC or Adler32 etc.?

=> They all fail badly at 3.

# Construction of modern hash functions

Instead of dealing with the messages $m \in \{0, 1\}^\infty$ directly, m is split up in n equal-sized chunks m_1...m_n (where the last block is padded).

# Construction of modern hash functions

m_1...m_n are then individually processed by a compression function

$h': \{0, 1\}^n -> \{0, 1\}^m$ with $m <= n$

where each result is combined with the next block of input in some specified manner, yielding a final output $y \in \{0, 1\}^n$

A famous hash function design is the Merkle-Damgard construction

(http://en.wikipedia.org/wiki/Merkle%E2%80%93Damg%C3%A5rd_construction)

The SHA-3 winner Keccak uses a new approach of using „sponge functions“

(cf. http://sponge.noekeon.org/)

Block ciphers can also be used to construct compression functions

In fact, many hash functions are constructed from an underlying block cipher

With Merkle-Damgard, it can be shown that if the

(iterated) compression function
operating on a finite domain
is collision-resistant,

then the hash function itself is collision-resistant.

Note, however, that collision resistance does not imply that collisions are impossible. In fact, collisions **must** exist („pigeon hole principle“):

We cannot map an infinite amount of values to a finite amount of values without ever repeating ourselves

Collision resistance is more in the sense of

„You'll win the lottery in all countries of the world at the same time while being struck by a lightning before this ever happens“.

See git „commit SHAs“ for an impressive example!

# Finding collisions by brute force

As with stream and block ciphers (by iterating through all possible keys), we may simply try to find a collision by choosing arbitrary, random values.

How many attempts will we need on average?

# Birthday Paradoxon

(cf. http://en.wikipedia.org/wiki/Birthday_problem)

„How many randomly chosen people does it take on average until two have the same birthday with a probability >= 50% ?“

Answer: 23!

# Birthday Paradoxon for integers

If we choose random integers
n_1, n_2, ... from {1, ...., 2^m},
how many must we choose until
we find two n_i, n_j with i != j and n_i = n_j
with probability >= 50% ?

Answer: ~2^(m/2)

# Birthday Paradoxon for hash functions

For a hash function h with a compression function

$h': \{0, 1\}^n \text{->} \{0, 1\}^m$

we must roughly compute 2^(m/2) hash values
with values randomly chosen from $\{0, 1\}^n$
to find a collision with probability >= 0.5

This is an upper bound!

We assume h' to be identically distributed
(as when we choose random values from $\{0, 1\}^m$).

In reality, h' is not a real random function,
thus collisions are only more likely because
h' is not identically distributed.

# Real-world hash functions

# MD5

Old
Broken
Don't

# SHA-1

While not broken yet

There have been more and more breakthroughs in its cryptanalyis lately

Don't use it in new designs

SHA-2

SHA-224
SHA-256
SHA-384
SHA-512

Unbroken, relatively fast

On some 64 bit machines
SHA-512 is faster than SHA-256

SHA-3

Keccak (Winner)
Skein
BLAKE
BLAKE 2
...

Use once available

# SHA-3

All hash functions are no longer vulnerable to length extension attacks (while SHA-1 & -2 are!)

(cf. http://blog.whitehatsec.com/hash-length-extension-attacks/)
(cf. http://www.skullsecurity.org/blog/2012/everything-you-need-to-know-about-hash-length-extension-attacks)
(cf. http://crypto.stackexchange.com/questions/3978/understanding-a-length-extension-attack)

# CODE.

# Hash functions

OpenSSL has:

MD5
SHA-1
SHA-224
SHA-256
SHA-384
SHA-512

no SHA-3 yet

# Hash functions

If you need to use hash functions stand-alone,

make sure that extension attacks are not a problem
or use SHA-3 algorithms.

In doubt (and with SHA-3 not available), use
HMAC.

# Hash functions

If performance is an issue, check whether SHA-512 might actually be faster on your system (especially for 64 bit OS) than SHA-256

# Hash functions

It is safe to ignore SHA-224 and SHA-384

# Message Authentication Codes (MACs)

Back to property 4.

4. Given just E(k,m) and h(m), we don't want anyone to be able to produce valid pairs E(k, m)' and h(m)' (e.g. by flipping bits etc.)

Idea:

Produce the „checksum“ or „tag“
using the secret key k:
$h(k, m) = y$

Then it should be computationally infeasible
to produce valid $E(k, m)'$, $h(k, m)'$
without knowing the key k

In fact, to ensure message integrity we absolutely need a key:

message || tag

If the function 'tag' is completely deterministic and unkeyed, everybody can change the message and simply recompute the tag!

# MACs built from ciphers

CBC-MAC
CMAC
PMAC
ECBC-MAC
NMAC

But most popular MAC

HMAC

(built from a hash function)

How can we build a MAC from a hash function?

MAC(k, m) := H( k || m)

Bad idea!

Given H(k || m) and by the properties
of the Merkle-Damgard construction
we can construct a valid H(k || m || padding || m')

(cf. links given in SHA-3 slide for length extension attacks)

# HMAC definition

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

with opad, ipad fixed constants and
H an arbitrary (secure) hash function.

Can be proven to be secure if H meets certain requirements.

# CODE.

HMAC is even considered secure
when used with SHA-1.
(cf. http://rdist.root.org/2009/10/29/stop-using-unsafe-keyed-hashes-use-hmac/)

Still, for consistency's sake,
let's use SHA-2 functions everywhere!

# HMAC

What is it with the ConstantTime::verify_equal?

Why can't we use == ?

# HMAC

Because it's wrong!!!

== is short-circuiting,
and becomes vulnerable to a timing attack.

Attacker can guess correct value in linear time!
http://codahale.com/a-lesson-in-timing-attacks/
http://rdist.root.org/2010/07/19/exploiting-remote-timing-attacks/

# HMAC

Use constant-time tag verification

Use a key length that exactly matches the digest output length

HMAC-SHA1 is still considered secure, but using HMAC-SHA-256 is good practice

# HMAC

Do not implement your own key-based Hash MAC – use HMAC instead!

# Encrypt-then-MAC

Use it if GCM is not, but
CTR/CBC mode are available

Use a master secret that is split up in an AES key portion and an HMAC key portion

Use constant-time array verification for comparing the HMAC tag

As an aside:

One key per purpose!

# Authenticated Encryption

Given our HMAC,
we should now be able to prevent
the „bit flipping attack“?

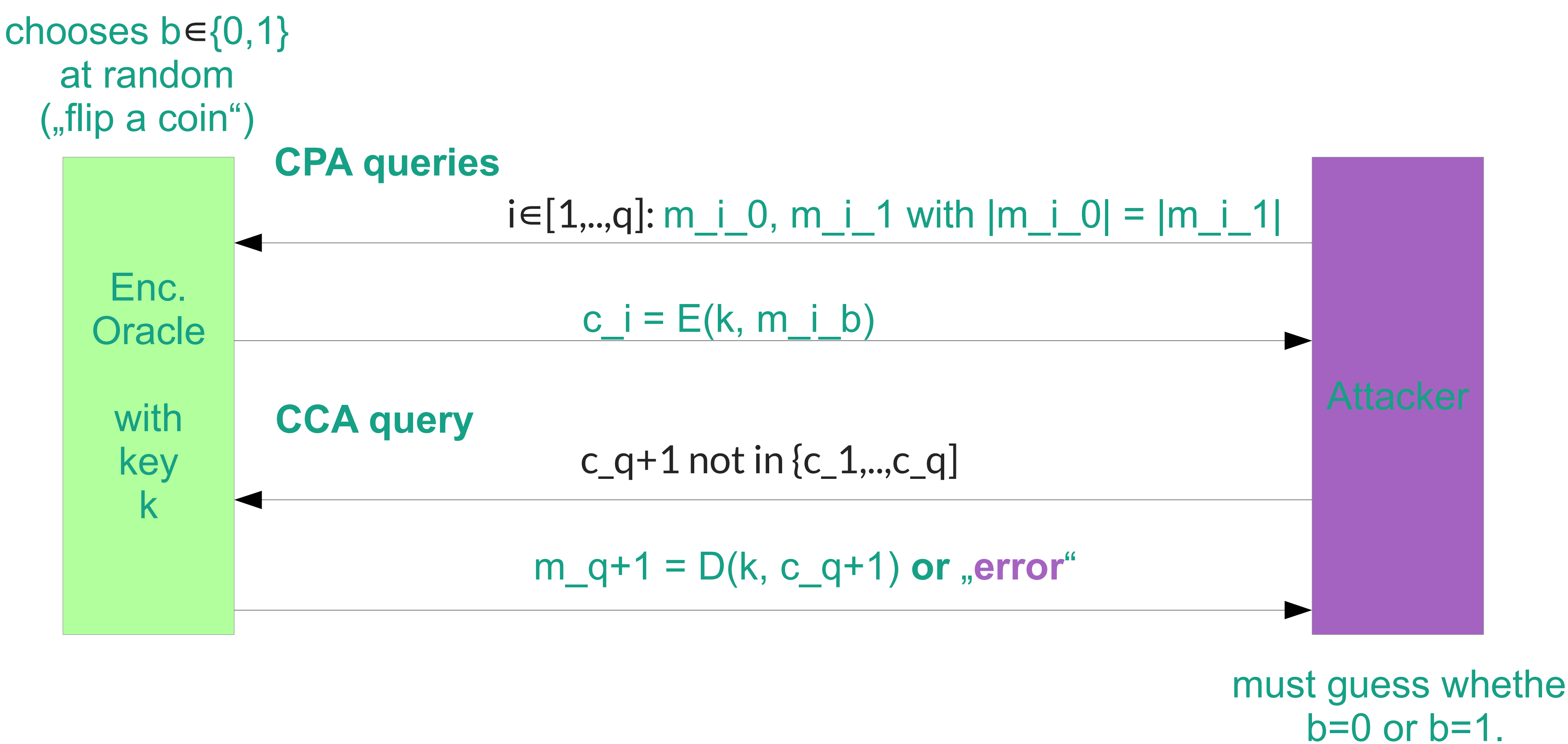
# Extending our Security Model

On top of IND-CPA, when given a ciphertext c,

it should be infeasible for the attacker to produce new ciphertexts c' that are accepted as valid ciphertexts.

# Indistinguishability under a Chosen Ciphertext Attack (IND-CCA)

# CCA security game



The additional „error" response is what the MAC can do for us.

The question remains:

How to combine MAC and encryption?

Three possibilities

1. First MAC the message, then encrypt message **and** the MAC tag, send the ciphertext

2. Encrypt the message, and produce the MAC tag separately, send both

3. First encrypt the message, then MAC the ciphertext and send ciphertext and MAC tag

# Three possibilities

1. „MAC-then-Encrypt“

2. „Encrypt-and-MAC“

3. „Encrypt-then-MAC“

None of the three is totally wrong

But only „Encrypt-then-MAC“ (3.)
can be proven to be IND-CCA unconditionally.

Intuitively: „Before I do anything else, I verify the integrity of what I received“

So „Encrypt-then-MAC“

is our holy grail?

Well...

Having to perform two operations is not optimal performance-wise.

Can't we authenticate in parallel while we decrypt?

# Authenticated Encryption Modes

GCM
OCB

EAX
CCM
CWC

Authenticated Encryption Modes

OCB

would be perfect

...but there are patents.

# Authenticated Encryption Modes

## GCM

(cf. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf)

Good, publicly available compromise. Technically it is a stream cipher mode similar to CTR.

# Conclusion:

Use GCM mode where available

Otherwise use Encrypt-then-Mac
with either CTR or, if not available, CBC

# CODE.

# Passwords and Keys

# The worst idea:

key = „imsosmartyoullneverguessthis“

# Why the worst idea?

A byte has 256 possibilities,

but there are just 52 characters
(uppercase/downcase)

Even if you add digits and special characters,
you hit significantly less than all 256 possibilities.

Additionally, „meaningful“ words/phrases contain considerably less entropy than real random data.

(cf. http://en.wikipedia.org/wiki/Entropy_(information_theory))

While we cannot induce
additional entropy into passwords,
we can use artificially slow algorithms
to derive keys from them.

This will also increase the
work that any attacker has to do
and will thus likely render the
attack computationally infeasible

Parallelization is a very powerful tool of attackers („PS3 farm“, FPGAs)

=> modern algorithms try to make parallelization as hard as possible

=> scrypt for example also tries to be „memory-hard“

# Password-based key derivation

Bad idea: key = SHA-1(password)

# Password-based key derivation

Bad idea: key = SHA-256(password)

# Password-based key derivation

To see why:

Imagine your password could either be 0 or 1 -> then there's also just two possible SHA-256 outcomes

Dictionary attack still works!

# Password-based key derivation

## Salts

Generate a random value („salt“),
and then compute

key = SHA-256(salt || password)

This is better, because nobody has precomputed dictionaries for all possible salt values obviously.

But it can still be computed way too fast, especially with dedicated (and affordable) hardware!

Ok, then

key = SHA-256(SHA-256(SHA-256(salt || password)))

?

The right direction. But instead of three applications of SHA-256, better try 20.000!

Still, this is ad-hoc, and better standards do exist!

# PBKDF2 (PKCS#5)

(cf. http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-5-password-based-cryptography-standard.htm)

# CODE.

# Key Generation with PBKDF2

http://www.ruby-doc.org/stdlib-2.1.0/libdoc/openssl/rdoc/OpenSSL/PKCS5.html

# Password Storage

Username / Password

What do we do with the password on the server?

# Password Storage

Store it in plaintext?

Shame on you!

# Password Storage

Encrypt it?

Not a good idea:

If the key is compromised, all passwords could be restored by an attacker

# Password Storage

Use a hash of the password?

This is the right direction. But as we already saw for password-based key derivation, hashes are way too fast.

# Password Storage

It turns out that the problems of password-based key derivation and password storage are very similar, and the same functionality may be reused.

# Password Storage

Test the optimal iteration count. It shouldn't be too low, but it should also not become a vector for DoS attacks! (but certainly well over 1000)

Use a fresh salt for every password!

Never ever use the password as the salt!

# Password Storage

Be careful when comparing whether things are equal

If in doubt, always use constant-time array comparison!

# Password Storage with PBKDF2

http://www.ruby-doc.org/stdlib-2.1.0/libdoc/openssl/rdoc/OpenSSL/PKCS5.html

# Password Storage BCrypt

https://github.com/codahale/bcrypt-ruby

# Password Storage with scrypt

https://github.com/pbhogan/scrypt

„Great, I have generated my key now.
But how do I give this key to my friend?“

# Key Exchange

Once a key is generated, it typically needs to be communicated to the recipient(s).

The problem is how to communicate it securely.

# Out-of-band measures

Over the phone (yay, NSA)

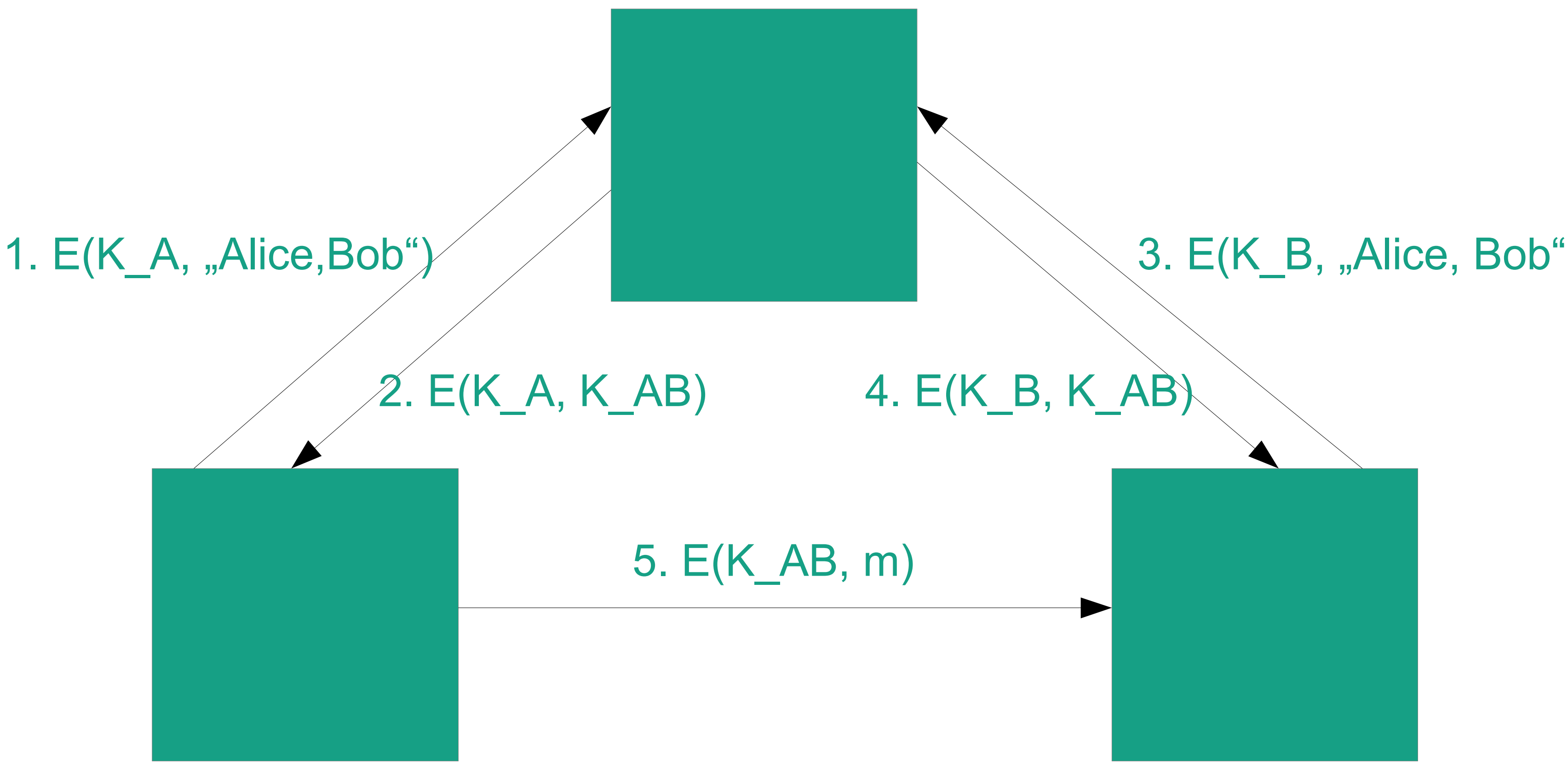
By snail mail (seriously?)

Meet in person

Out-of-band communication (partly) works, but is tedious and doesn't scale very well.

Ideally, we'd like to be able to communicate a secret key over an unsecure channel in an automated way.

# Trusted Third Party



# Trusted Third Party

This is very roughly the underlying idea of Kerberos

(cf. http://web.mit.edu/kerberos/)

# Trusted Third Party

Works well, but only in closed systems.

Is it possible to exchange keys ad-hoc, without a trusted third party?

Yes, but this requires asymmetric public/private key cryptography!

# Asymmetric Cryptography

(Public key cryptography)

Next time.

# Links & Resources

# Books

## Cryptography, Theory and Practice
Douglas Stinson

## Cryptography Engineering
Niels Ferguson, Bruce Schneier

## Understanding Cryptography
Christof Paar, Jan Pelzl

# Books

## Introduction to Modern Cryptography:
### Principles & Protocols
Jonathan Katz, Yehuda Lindell

## Foundations of Cryptography Vol. I & II
Oded Goldreich

# Links

http://www.reddit.com/r/netsec

http://news.ycombinator.com

http://lists.randombit.net/mailman/listinfo/cryptography

http://www.metzdowd.com/mailman/listinfo/cryptography

https://www.owasp.org

https://cryptocoding.net/index.php

https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards

Papers, papers, papers...

# Blogs

http://www.reddit.com/r/programming/

http://blog.cryptographyengineering.com/

https://www.imperialviolet.org/

Follow cryptographers etc. on Twitter!

# Source Code

https://github.com/openssl/openssl

https://github.com/bcgit/bc-java

http://hg.openjdk.java.net/jdk7/

...

# Online Classes

https://www.coursera.org/course/crypto

https://www.coursera.org/course/crypto2

https://www.udacity.com/course/cs387

# thank you

@_emboss_

https://github.com/krypt

http://martinbosslet.de

martin.bosslet@gmail.com