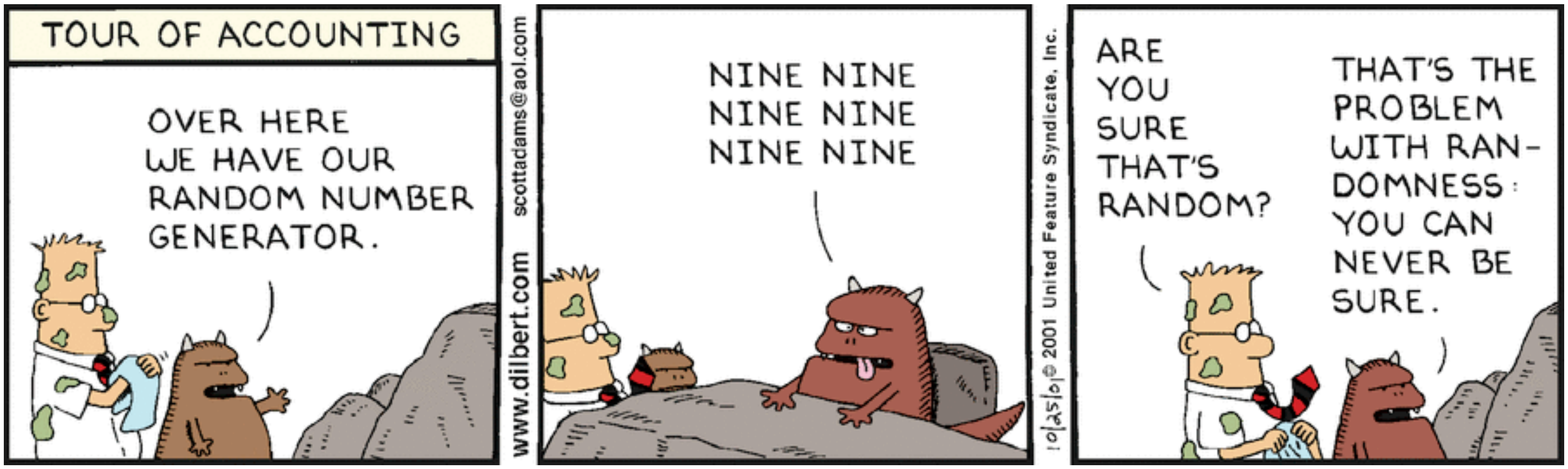
Randomness
and
Pseudorandomness


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
scottadams@aol.com
www.dilbert.com
NINE NINE NINE NINE NINE NINE
© 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

Review

Problem:

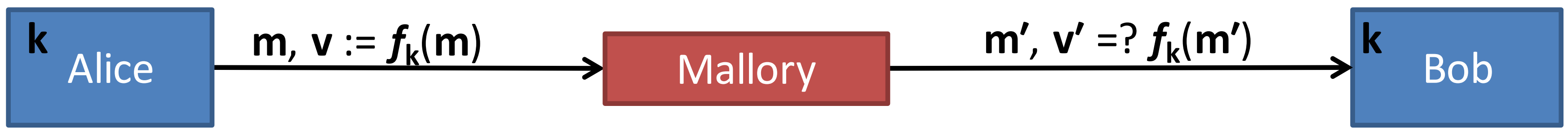
Integrity of message from Alice to Bob over an untrusted channel

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

Theoretical solution:

Random function

Practical solution:



e.g. “Attack at dawn”, 628369867…

Pseudorandom function (PRF)

f_k is indistinguishable in practice from a random **f**, unless you know **k**

Embodied by functions like HMAC-SHA256

Where do these random keys k come from … ?

Careful: We’re often sloppy about what is “random”

Review

Problem:

Integrity of message from Alice to Bob over untrusted channel

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

Solution:

Random function

Practical solution:

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

Embodied by functions like **HMAC-SHA256**

Where do these random keys **k** come from … ?

Careful: We're often sloppy about what is "random"

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”

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 they want
4. Mallory guesses **b**, wins if guesses correctly

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

Here's a *simple PRG that works:*

For some random **k** and PRF ***f***,

output: $\boldsymbol{f}_k(0) \,||\, \boldsymbol{f}_k(1) \,||\, \boldsymbol{f}_k(2) \,||\, \ldots$

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 eventually 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

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?

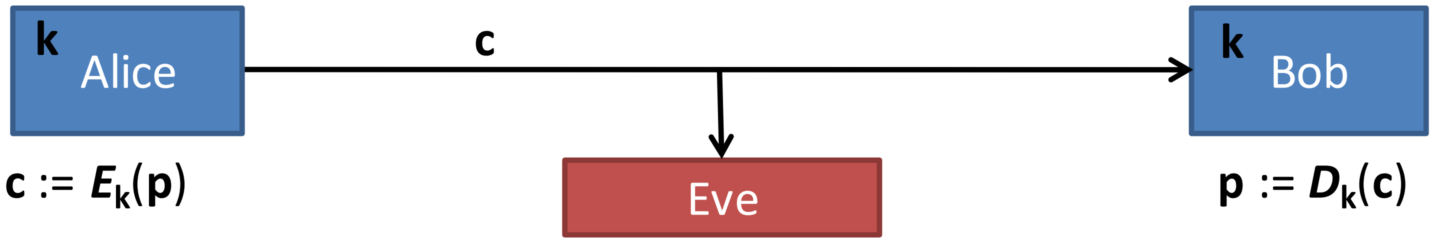
Modern OSes typically collect randomness, provide API to get it

- 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 eventually. (Initially may not be sufficiently random.)~~

Confidentiality

Confidentiality

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


k Alice
c
k Bob
Eve
$\mathbf{c} := E_k(\mathbf{p})$
$\mathbf{p} := D_k(\mathbf{c})$

Nomenclature

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 the letter 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?]

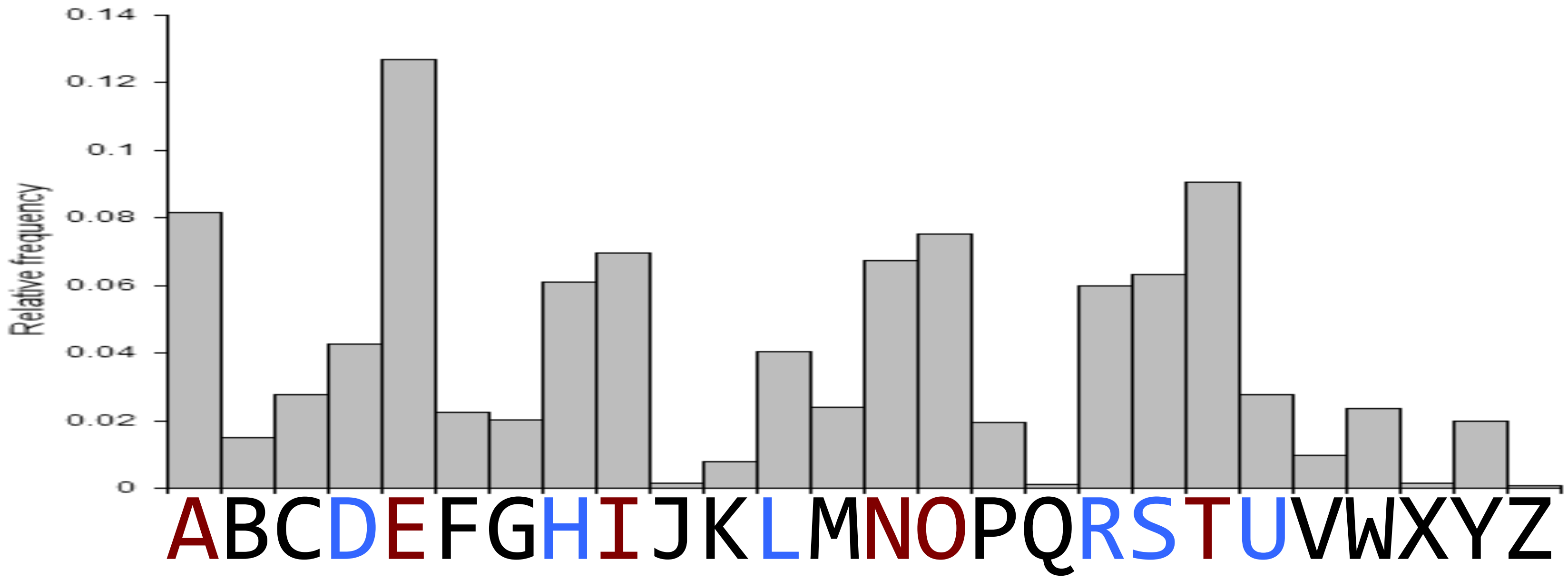
Cryptanalysis of the Caesar Cipher

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. X^2-square test

Later advance: Vigènere Cipher

First described by Bellaso in 1553,
later misattributed to Vigenère

Called *le chiffre indéchiffrable*

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

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

Encryption: $\mathbf{c_i} := (\mathbf{p_i} + \mathbf{k_{i \bmod n}}) \bmod 26$
Decryption: $\mathbf{p_i} := (\mathbf{c_i} - \mathbf{k_{i \bmod n}}) \bmod 26$

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

Easy, 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?]

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

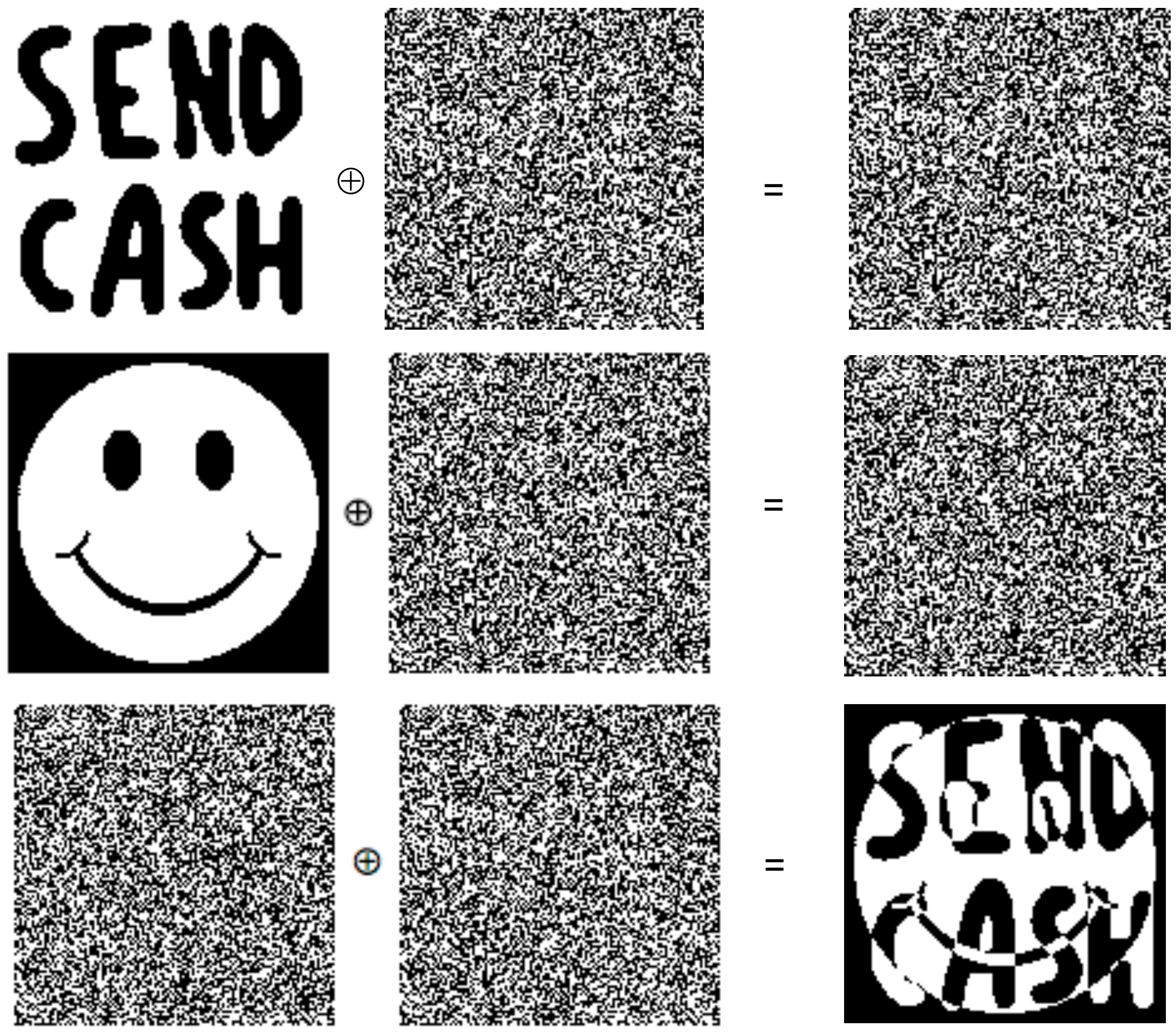
Adversary learns (**a** xor $\mathbf{k}_i$) and (**b** xor $\mathbf{k}_i$)

Adversary xors those to get (**a** xor **b**), which is useful to him [How?]

Provably secure [Why?]

Usually impractical [Why? Exceptions?]

One-time pad reuse example (from stackexchange)



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

(Recall: Secure **PRG** inputs a seed **k**, outputs a stream that is indistinguishable in practice 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>* reuse the key,

or the generator output bits

So Far

The Security Mindset

Message Integrity

Randomness /Pseudorandomness

Confidentiality: Stream Ciphers

Next Wednesday…

Block Ciphers and Cipher Modes