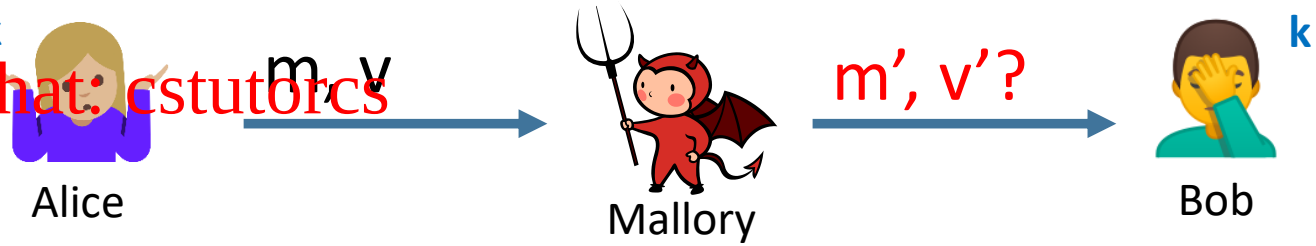
Cryptography Basics – (Pseudo)Randomness

ECEN 4133
Jan 21, 2021

Review

- Integrity of messages between Alice and Bob
- Alice appends bits that only Alice (and Bob) can compute
- Solution: Message Authentication Code (MAC)
  - Hash-based MAC (HMAC) used in practice, e.g. $v = \textbf{HMAC-SHA256}_k(M)$



- Where does **k** come from?
  - How do we generate it? [Today]
  - How do we share it with Alice and Bob, but not Mallory? [Next time]

Randomness

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”

True Randomness

Where do we get true randomness?

Want “indistinguishable from random” meaning: 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?

Getting a large amount of randomness

Difficult to collect lots of **true random**

Suppose we have 128-bits of true random (**k**), but want 1024-bits of random

- Can we **expand** our 128-bits? [Breakout]
  - Can we extend to arbitrary lengths?
  - Any caveats? How many unique "sequences" of 1024-bits can we produce with this technique?

Getting a large amount of randomness

"Pseudo" Randomness:

- Not truly random – usually an expansion of a (shorter) set of true random bits

One solution: Pseudo-random number generator (PRNG):

- Given 128-bit true random **k**
- **HMAC-SHA256$_k$(0), HMAC-SHA256$_k$(1), HMAC-SHA256$_k$(2), … HMAC-SHA256$_k$(n)**

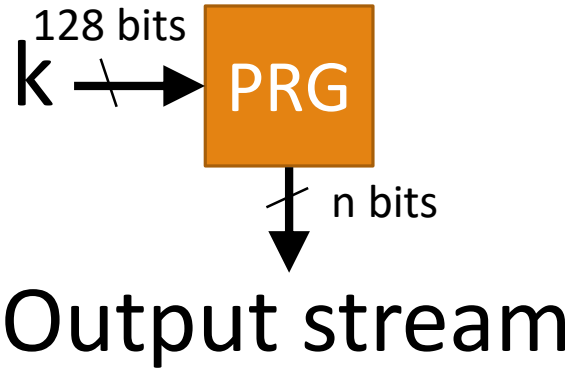
Is it secure?

- Can an adversary tell what will come next without knowing **k**?
- Given **HMAC-SHA256$_k$(a)**, (but not **k**), can an adversary predict **HMAC-SHA256$_k$(b)** for **b>a** ?

Pseudo-random generators (PRG)

Many different ways:

- Using hashes
- Using HMACs
- Using block ciphers (we'll talk about these next)



Beware that there also exist *non-cryptographic PRNGs*:

- Linear feedback shift register (LFSR)
- Linear Congruential Generator (LCG)
- Used by rand() / srand() / Math.random() – **Don't use for cryptography!!!**

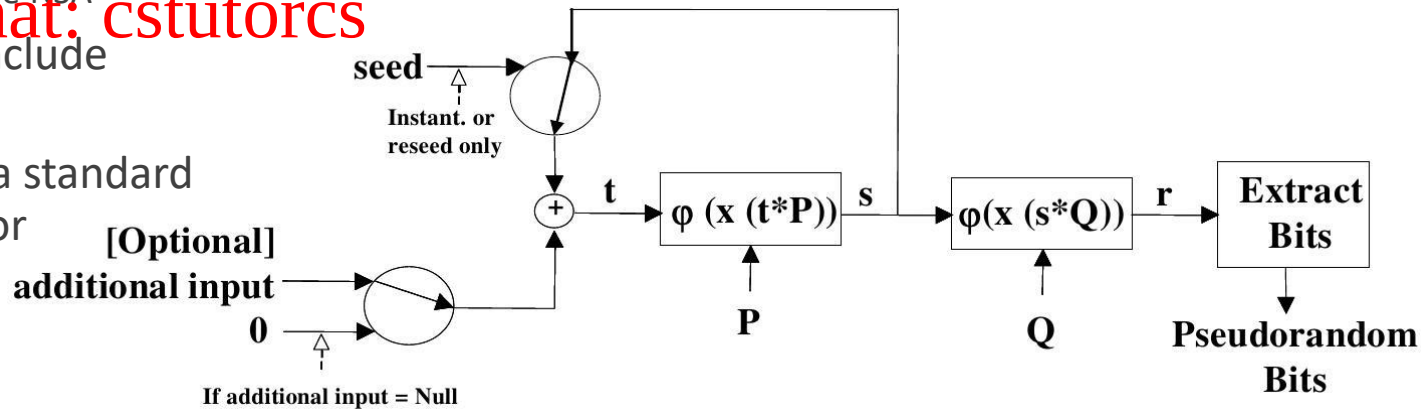
We are talking about *Cryptographically Secure PRNG* (**CSPRNG**)

- Should be difficult for adversary to predict future (or past!) outputs given some output

“Backdoored” CSPRNG

Dual_EC_DRBG

- Dual Elliptic Curve Deterministic Random Bit Generator
- Developed by the NSA in 2006, standardized by NIST
- Strange design, very slow, based off elliptic curve cryptography (next week!)
- If someone knows a mathematical relationship between P and Q, trivial to compute what comes next in the pseudorandom stream given current output (backdoor!!)
  - No explanation for how P and Q were chosen by the NSA
- NSA paid $10 million to RSA Security to include in their popular cryptographic library
- Snowden documents revealed this to be a standard developed solely by the NSA as a backdoor



Randomness in practice

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` *eventually*

Note: both /dev/random and /dev/urandom use a CSPRNG seeded from:

- Keystroke/mouse movement timing
- Network packet timing
- Scheduler / interrupt timing
- /dev/random tries to do “entropy accounting”: don’t give out more than has been “put in” to the pool

/dev/(u)random problems

/dev/random blocks – slow to read from

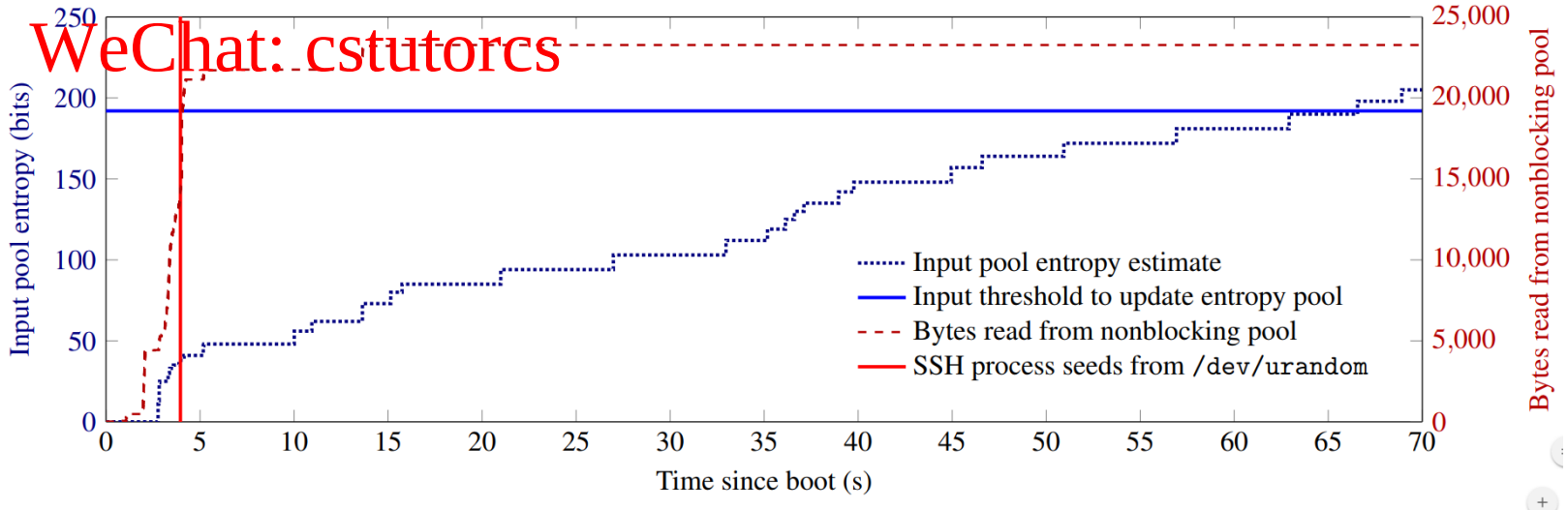
/dev/urandom doesn't block – **but might not be initialized at all!!!**

Embedded devices:

- Often don't have keyboard/mouse
- Might not be connected to Internet at first boot (no packets)
- Very slow to collect entropy!

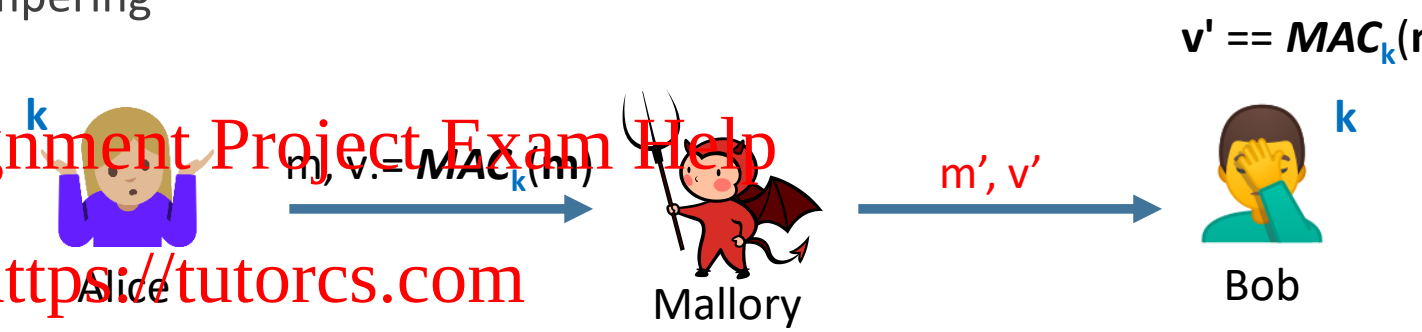
Solution:

- Use `getrandom()`
  - Added in Linux 3.17 (2014)
  - Blocks until pool has been initialized



Confidentiality

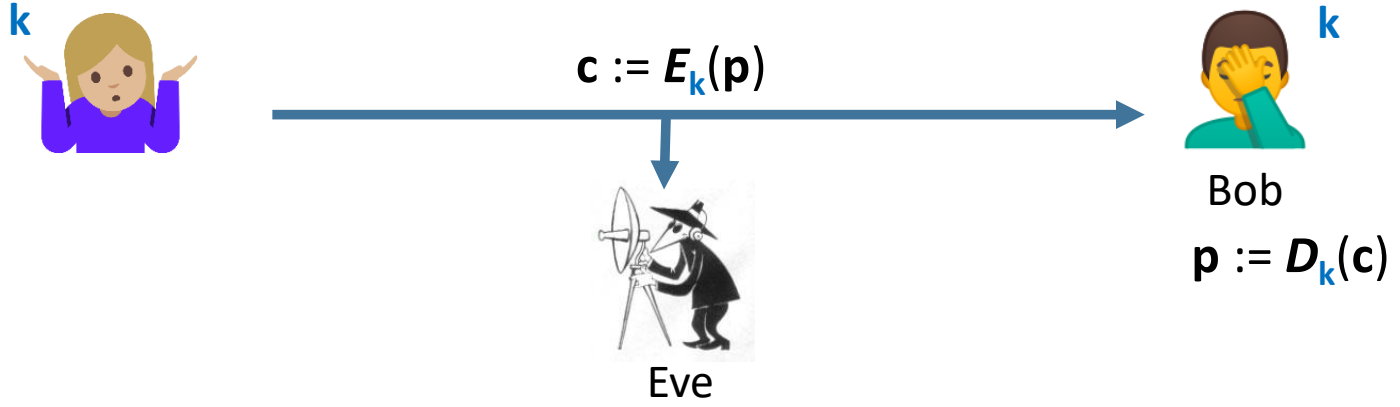
Integrity: prevent Mallory from tampering



Confidentiality: prevent eavesdropper (Eve) from learning the (plaintext) message

Terminology

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



Classical Cryptography

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?]

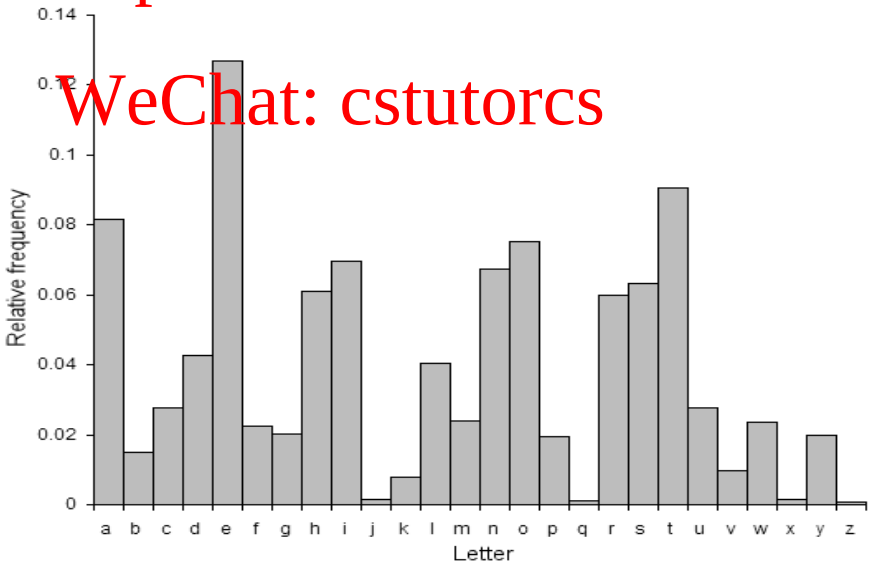
Cryptanalysis of the Caesar Cipher

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



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: $\mathbf{c}_i := (\mathbf{p}_i + \mathbf{k}_{i \bmod n}) \bmod 26$
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

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: **CSASTP**KVSIQUTGQU**CSASTP**IUAQJB

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?]

One-time Pad (OTP)

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

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

To decrypt: $\mathbf{p}_i = \mathbf{c}_i$ 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 might be useful [How?]

Provably secure [Why?]

Usually impractical [Why? Exceptions?]

Practical One-time Pad

Idea: Use a **pseudorandom generator** (CSPRNG) 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 *ever* reuse the key, or the generator output bits