Cryptography

1 – Secret-key encryption: applying masks

G. Chênevert

September 16, 2019

ISEN ALL IS DIGITAL! LILLE | yncréa

Today

Secret-key encryption

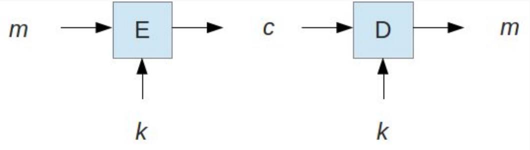
One-time pad

Stream ciphers

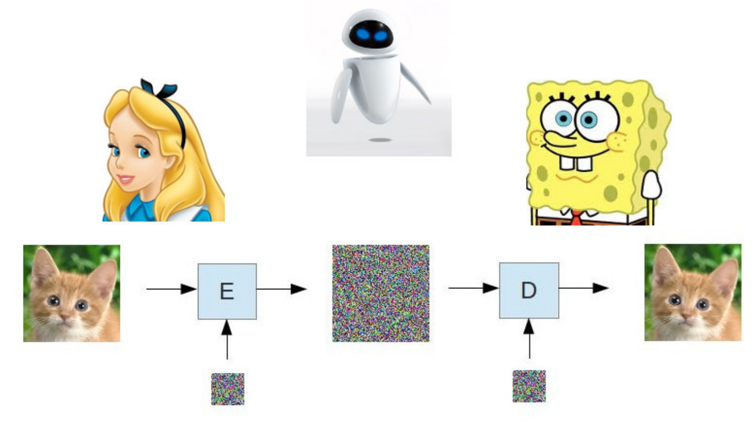
Secret-key encryption

Recall: a **symmetric cipher** consists of a pair of encryption/decryption functions

$$E : \mathcal{K} \times \mathcal{M} \longrightarrow \mathcal{C} \qquad \text{and} \qquad D : \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{M}$$



Secret-key encryption



Requirements

- **Correct decryption** : for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$,

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

- **Perfect secrecy** : knowledge of the ciphertext should give an attacker *no information whatsoever* about the plaintext, *i.e.*

$$\mathbb{P}[M = m \mid C = c] = \mathbb{P}[M = m]$$

with $M \in \mathcal{M}$ and $C \in \mathcal{C}$ considered as random variables.

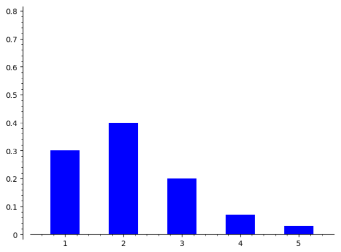
Example

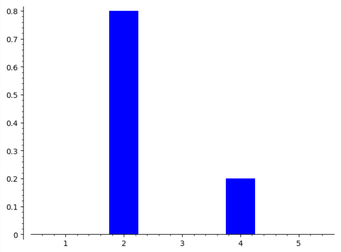
Bob: How many hot-dogs do you want?

Alice encrypts $m \in \mathcal{M} = \{1, 2, 3, 4, 5\}$ by adding to it a large *even* integer k.

Eve overhears ciphertext 876523987428763529987687 4 . . .

Her assessment of the possibilities for m changes: she gained some *information*.

Before: 

After: 

Perfect secrecy

Replaced in practice by **semantic security**:

no polynomial time algorithm should give any attacker a *non-negligible advantage*

i.e. there exists no (efficient) **ciphertext-only attack**

In pratice: **negligible** means $\leq \frac{1}{2^{128}}$.

Example with small key space

Suppose $|\mathcal{M}| = |\mathcal{C}| = 2^{1024}$, $|\mathcal{K}| = 2^8$.

Attack: given $c \in \mathcal{C}$,

- choose $k \in \mathcal{K}$ randomly,
- output $D(k, c)$.

Non-negligible probability of success!

$\implies$ key space should be large ($\geq 2^{128}$) NB: message space too!

Today

Secret-key encryption

One-time pad

Stream ciphers

The one-time pad

(Miller 1882, Vernam 1917)

Take $\mathcal{M} = \mathcal{C} = \mathcal{K} = G$ any finite abelian group:

Definition

$$\begin{cases} E(k, m) = m + k \\ D(k, c) = c - k \end{cases}$$

Example

with $G = (\mathbb{Z}/26\mathbb{Z})^n$

```
m = S("ENCRYPTASTRINGBYRANDOMLYSHIFTINGEVERYLETTER")

k = randkey(len(m))

print("plaintext:  ", m)
print("key:        ", k)
print("ciphertext: ", m + k)
```

```
plaintext:   ENCRYPTASTRINGBYRANDOMLYSHIFTINGEVERYLETTER
key:         UFHAXHFMPEFENHTZCCDKRSVCAHKIZTVZEVZCSXUTPDV
ciphertext:  ZTKSWXZNIYXNBOVYUDROGFHBTPTOTCJGJREURJZNJIN
```

cf. LAB0

In practice (from now on)

Use $G = (\mathbb{Z}/2\mathbb{Z})^n$

Group law: componentwise addition mod 2

aka bitwise XOR, or $\oplus$

Example

$$010011 \oplus 111000 = 101011$$

Notice: for all x we have $\ominus x = x$, *i.e.* $x \oplus x = 0$

Binary one-time pad

With $\mathcal{M} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}/2\mathbb{Z})^n$:

Definition

$$\begin{cases} E(k, m) = m \oplus k \\ D(k, c) = c \oplus k \end{cases}$$

Encryption and decryption are the same function!

Example (12 bits)

Alice:

$m = 111000111000 = \mathrm{E38}$

$k = 011011010111 = \mathrm{6D7}$

$c = m \oplus k = 100011101111 = \mathrm{8EF}$

Bob:

$c = 100011101111 = \mathrm{8EF}$

$k = 011011010111 = \mathrm{6D7}$

$m = c \oplus k = 111000111000 = \mathrm{E38}$

Example (128 bits)

In [1]:

```
from os import urandom

def xor(a,b):

    return bytes([x^y for x,y in zip(a,b)])

k = urandom(16)
```

In [2]:

```
# Alice

m = b"OTP on 128 bits!"

c = xor(m,k)

print("m =", m.hex())
print("k =", k.hex())
print("c =", c.hex())
```

```
m = 4f5450206f6e203132382062697473 21
```

In [3]:

```
# Bob

mm = xor(c,k)

print("c =", c.hex())
print("k =", k.hex())
print("m =", mm.hex())
```

```
c = 53fa61b076e05071ff70420aaecfb1e5
k = 1cae3190198e7040cd486268c7bbc2c4
m = 4f5450206f6e20313238206269747321
```

OTP is provably secure! (1/2)

Theorem

The one-time pad decrypts correctly.

Proof.

$$\begin{aligned} D(k, E(k, m)) &= (m \oplus k) \oplus k \\ &= m \oplus (k \oplus k) \\ &= m \oplus 0 \\ &= m. \end{aligned}$$

□

OTP is provably secure! (2/2)

Theorem (Shannon, 1949)

The one-time pad has perfect secrecy.

Proof.

Assuming K is uniformly distributed and independent from M,

$$\mathbb{P}[M = m,\ C = c] = \mathbb{P}[M = m,\ K = c \oplus m] = \frac{1}{2^n}\mathbb{P}[M = m],$$

$$\mathbb{P}[C = c] = \sum_m \mathbb{P}[M = m,\ C = c] = \frac{1}{2^n}\sum_m \mathbb{P}[M = m] = \frac{1}{2^n}$$

hence $\mathbb{P}[M = m \mid C = c] = \mathbb{P}[M = m]$.

$\square$

Drawbacks

- The key is as long as the message!

  But: still allows a transfer in secrecy (from m to k)

- The key should **never** be reused

  For if $c_1 = m_1 \oplus k$ and $c_2 = m_2 \oplus k$, then

$$c_1 \oplus c_2 = m_1 \oplus m_2 \text{ !}$$

Which is a serious violation of perfect secrecy.

Today

Secret-key encryption

One-time pad

Stream ciphers

One-time pad

With $\mathcal{K} = \mathcal{M} = \mathcal{C} = \{0,1\}^n$:

$$E(k,x) = D(k,x) = x \oplus k.$$

Perfect secrecy, security level n, but:

- key as large as message
- fresh key needed for every message
- *malleable*: more on that later

Stream ciphers

Idea: make the OTP practical (addressing first drawback)

Definition (binary additive stream cipher)

$$E(k, x) = D(k, x) = x \oplus G(k)$$

with $\mathcal{M} = \mathcal{C} = \{0, 1\}^n$, $|\mathcal{K}| = 2^m$, $m \ll n$ and

$$G : \{0, 1\}^m \to \{0, 1\}^n$$

a *cryptographically secure pseudo-random number generator* (CSPRNG)

Pseudo-random number generators

```
In [1]: import random

        # uses *insecure* but efficient Mersenne Twister PRNG

        random.seed(12345)

        for i in range(16):

            print(hex(random.randint(0,2**128))[2:-1])
```

```
6facaa5090e5e945452ec40a3193ca5
6ed4e94bdfc9e3b11fcff4545f811cb
bc428d42fa88269287f26aee175f0cd
25ece8452aa4857e8101e89a95c5fb9
d64a3ce030a1f6d513ed748bb80e3b0
56eaa3017576714a06057c82527122d
94820a06c555663f29ef41d0deea959
6a1eccdaa70ce1b51978cec0495cfa4
df8960ad1eab5cd83b788b660a4de3e
96af0dea41fad2962f927291ab721ab
213f191ff56ae7eaea80db0684ab561
f70ae8c026784184026530cdd50b612
282fe557578b24268a04f74f5987baf
9f3180427b1427081f1af1fac2e1dac
265015788e7ae9af1e8fcb74b2d4f32
f79fcaa0e47b342b2a3a46677eb14f8
```

Requirements for CSPRNGs

- *All* PRNGs are eventually periodic

  (deterministic stateful functions with a finite number of internal states)

  $\implies$ certainly want long period

- Most "standard" PRNGs are easily predictable!

  $\implies$ **related-key attacks** on the underlying OTP

Linear congruence generator

Definition

Given *seed* x_0, generates a pseudo-random sequence $(x_n)_{n=1}^{\infty}$ with

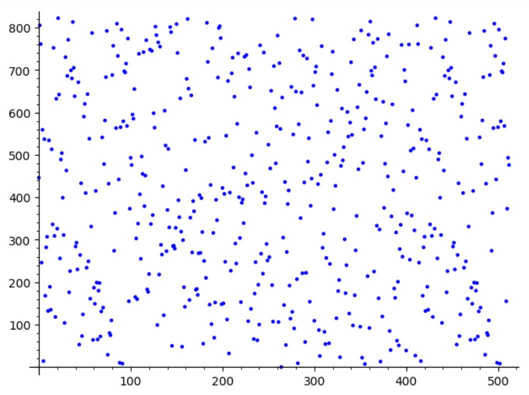
$$x_{n+1} = (ax_n + b) \,\%\, p$$

with a, b fixed constants (integers) and p a prime number.

The knowledge of three consecutive terms is enough to recover a and b!

Hint: the points (x_n, x_{n+1}) all lie on the "line" $y \underset{p}{\equiv} ax + b \ldots$

Example: $p = 823$, $a = 816$, $b = 635$, $x_0 = 446$



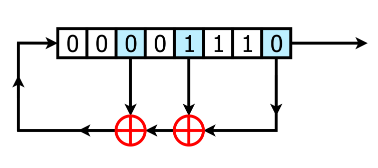
In practice: LFSRs

Would like to take $p = 2$, but not very interesting...

$\implies$ instead: output bit is a fixed linear combination of previous output bits

(closely related to polynomial multiplication!)

Linear feedback shift registers



Algebraic interpretation of LFSRs

Choose a degree d irreducible polynomial $f(x)$ over $\mathbb{F}_2$

e.g., $f(x) = x^3 + x + 1$, $d = 3$

and pick a root α of f (somewhere!)

$$\rightsquigarrow \mathbb{F}_2(\alpha) = \{ a_0 + a_1\alpha + \cdots + a_{d-1}\alpha^{d-1} \mid a_0, a_1, \ldots, a_{d-1} \in \mathbb{F}_2 \}$$

field with 2^d elements

Algebraic interpretation of LFSRs

Given $x_0 \in \mathbb{F}_2(\alpha)$, define $x_{n+1} := \alpha \cdot x_n$ (and output the new a_0)

Period is $2^d - 1$ if f is *primitive* (and $x_0 \neq 0$)

Can be generalized to work with matrices (famous Mersenne Twister)

Still very much like a linear congruence generator! (with $\beta = 0$ ...)

$\implies$ use *nonlinear combinations* of outputs of LSFRs

Some (in)famous stream ciphers

That use *linear* combinations of LSFRs:

- CSS
- GSM
- Bluetooth E0

Some weaknesses found:

- RC4 (used in TLS/SSL and WEP)

Current recommendations

The eSTREAM project (ECRYPT 2008) proposes

- HC-128, Rabbit, Salsa20, SOSEMANUK (software-oriented)
- Grain, MICKEY, Trivium (hardware-oriented)

(all force the PRNG to use a **nonce** as initial value)

Still need to be careful to seed the CSPRNG with enough entropy: using PID or timestamps is not a good idea!

$\Longrightarrow$ better use the system entropy pool *e.g.* `/dev/urandom`

Weekly Jupyter lab

In teams of $n = n_{\mathrm{CSI}} + n_{\mathrm{CIR}} + n_{\mathrm{new}}$ where:

- $2 \leq n \leq 4$
- $n_{\mathrm{CSI}},\ n_{\mathrm{CIR}},\ n_{\mathrm{new}} \leq 2$

You are encouraged to come up with a hacker team name for your team.

We will use Jupyter with Python 3: either from a local SageMath (or Anaconda) install or online on CoCalc.

Get the archive at https://gch.ovh/crypto (submit on Campus by Monday).