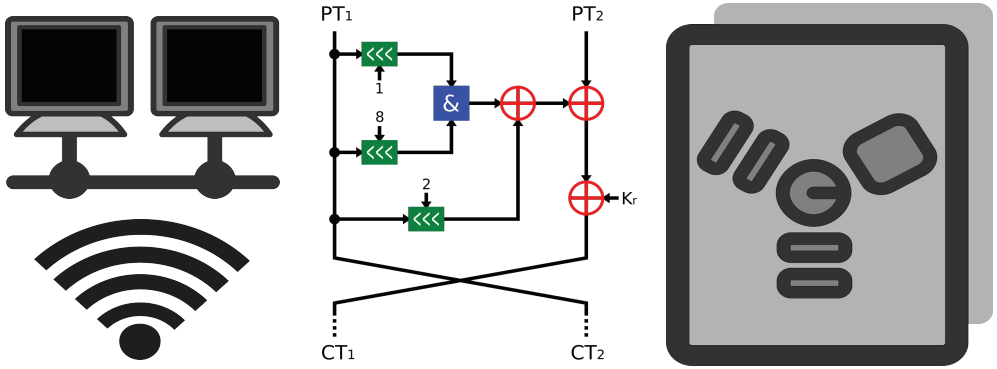
CSE 40567 / 60567: Computer Security



Cryptography 3

Homework #2 has been released. It is due Thursday, Feb. 6th at 11:59PM

See **Assignments Page** on the course website for details

Cryptographic Algorithms

Random Number Generation

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"

-Knuth quoting von Neumann

- Where do random numbers come from?
  - Produced by computers (Pseudo-random)
  - Measured from physical phenomena (Real Random)

Pseudo-random Sequences

1. It looks random. Passes all statistical tests of randomness that we can find*

2. It is unpredictable. Computationally infeasible to predict what the next random bit will be, given complete knowledge of the algorithm, hardware, and all previous bits

$$s_1, f(s_1) = 01011110\ldots$$
$$s_2, f(s_2) = 11010111\ldots$$
$$s_3, f(s_3) = 10101110\ldots$$

- Seed s_x is treated as a secret
- A seed will yield the same sequence when reused

*D. Knuth, The Art of Computer Programming, Volume 2, Seminumerical Algorithms, 2nd Edition, 1981

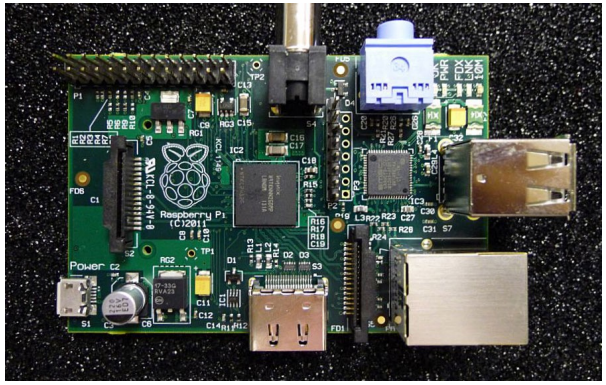
Real Random Sequences

3. It cannot be reliably reproduced. If you run the generator twice with exactly the same input, you will get two completely unrelated random sequences

Sources of entropy

Laptop hard drive exposed

Timing of read/write head movement



Noise from a reverse bias transistor, e.g., Raspberry Pi HRNG

Sources of random numbers

- `/dev/random` - blocking pseudo-random number generator (hash implementation via SHA)
  - Handy way to check the entropy pool:
    `$ cat /proc/sys/kernel/random/entropy_avail`
- `/dev/urandom` - unlimited non-blocking pseudorandom number generator
- Python: `import random` (Mersenne Twister)

```
>>> random.seed(1)
>>> random.random()
```

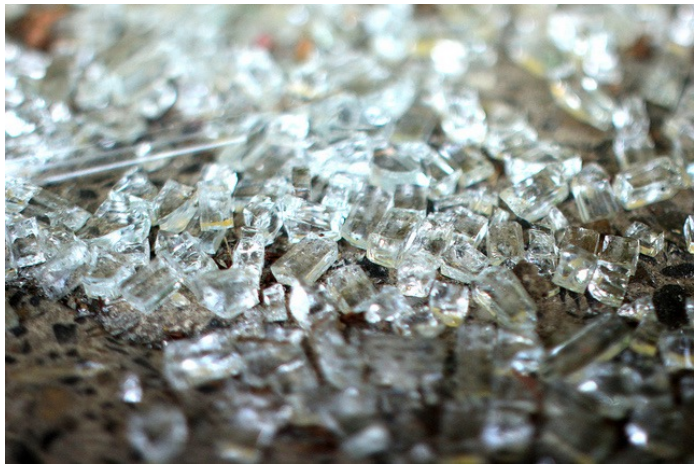
Random number generation in GPG

```
$ gpg --gen-key

gpg (GnuPG) 1.4.11; Copyright (C) 2010 Free Software
Foundation, Inc. This is free software: you are free to
change and redistribute it. There is NO WARRANTY, to the
extent permitted by law.

We need to generate a lot of random bytes. It is a good idea
to perform some other action (type on the keyboard, move the
mouse, utilize the disks) during the prime generation; this
gives the random number generator a better chance to gain
enough entropy.

Not enough random bytes available.  Please do some other work
to give the OS a chance to collect more entropy! (Need 284
more bytes)
```

One-way functions

- Input maps to unique output
- Given x, it is easy to compute $f(x)$
- Given $f(x)$, it is hard to compute x

Analogy: breaking a window



Broken Glass (cc) BY 2.0 Christian Schnettelker

It's easy to smash a window into a thousand pieces; not easy to put it back together.

Recall hash functions from Data Structures

- *Hash Tables* are useful data structures that allow searching in $O(1)$ time
  - Facilitated by a hash function

    Example:

```
size_t precision = 2;
size_t hash(const char* str)
{
   return (*(size_t*)str)>> precision;
}
```

Requirements for cryptographically strong hash functions

- Must be **collision-free**: hard to generate two inputs with the same hash value
- The output must not be dependent on the input in any discernible way
- A single bit change in the input changes, on average, half of the bits in the hash value
- Given a hash value, it is computationally infeasible to find the input that hashes to that value

md5

- 128-bit hash function

Commonly used as checksum for downloads:

```
$ md5sum ubuntu-14.04.3-desktop-amd64.iso
cab6dd5ee6d649ed1b24e807c877c0ae ubuntu-14.04.3-desktop-amd64.iso
```

- Input: 512-bit blocks, divided into 32-bit sub-blocks
- Output: Set of four 32-bit blocks, concatenated
- After input padding, main loop processes input through four rounds

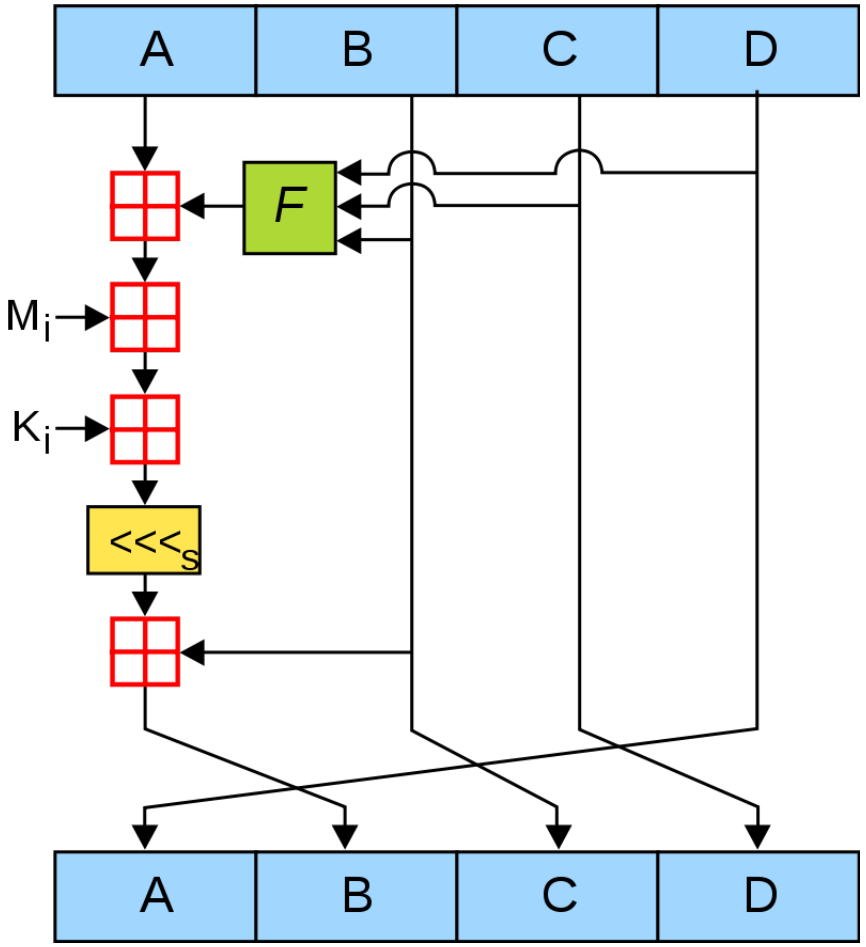
One md5 round

F = non-linear function

M_i = sub-block of the message

K_i = constant

Chaining variables



One md5 operation BY-SA 3.0 Surachit

Non-linear functions

Four non-linear functions, a different one for each round:

$$F(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$$

$$G(X, Y, Z) = (X \wedge Z) \vee (Y \wedge (\neg Z))$$

$$H(X, Y, Z) = X \oplus Y \oplus Z$$

$$I(X, Y, Z) = Y \oplus (X \vee ((\neg Z))$$

If the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of the result will also be

Collision Attack

- Note that md5's output space is finite (128 bits), thus collisions *must* exist
  - But it should be infeasible to find them
- Modular differential attack: Wang and Yu, Eurocrypt 2005

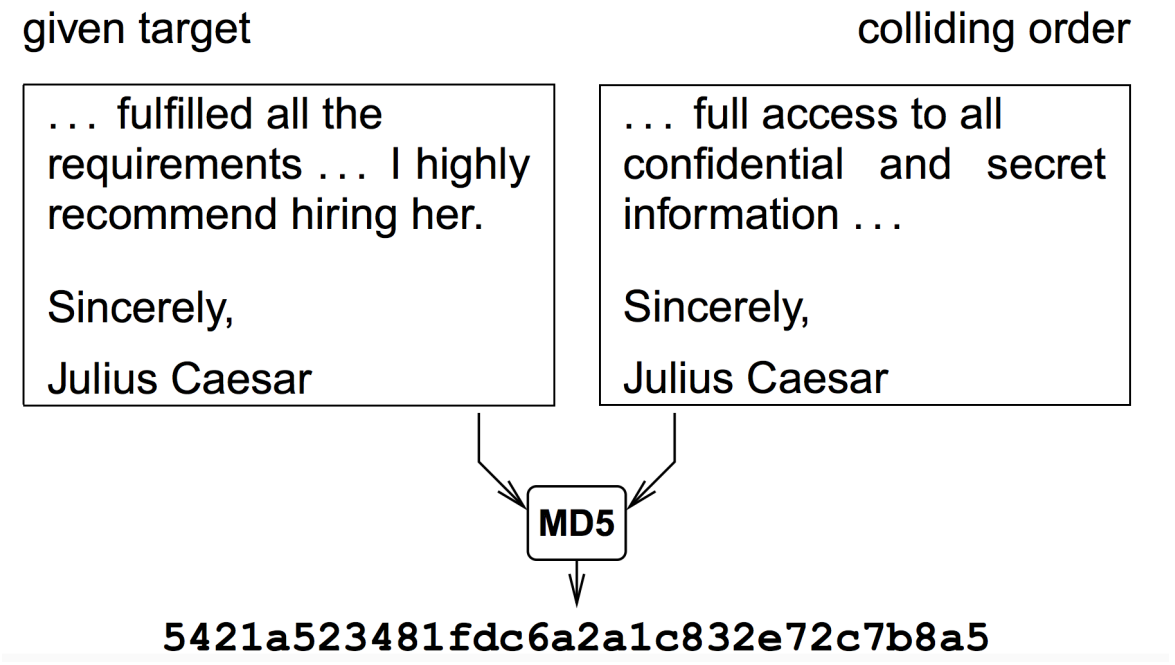


Image Credit: Lucks and Daum, Eurocrypt 2005

Modular differential attack

We want to find a pair (M_0, M_1) and (M_0', M_1') such that:

$(a, b, c, d) = \text{md5}(a_0, b_0, c_0, d_0, M_0)$,

$(a', b', c', d') = \text{md5}(a_0, b_0, c_0, d_0, M_0')$,

$\text{md5}(a, b, c, d, M_1) = \text{md5}(a', b', c', d', M_1')$,

- Use modular integer subtraction as the measure of difference
  - Finding the first blocks (M_0, M_0') takes about 2^{39} md5 operations
  - Finding the second blocks (M_1, M_1') takes about 2^{32} md5 operations

X. Wang and H. Yu, "How to break MD5 and Other Hash Functions," Eurocrypt 2005

Finding useful collisions

Lucks and Daumen, Eurocrypt 2005

1. Use advanced document language, e.g., postscript
2. Find random strings R_1 and R_2 and concatenate to some preamble:

$$X_1 = \textbf{preamble}; \text{put}(R_1);$$

$$X_2 = \textbf{preamble}; \text{put}(R_2);$$

$$\text{md5}(X_1) = \text{md5}(X_2)$$

Append a string S to both X_1 and X_2:

$$\text{md5}(X_1 \,||\, S) = \text{md5}(X_2 \,||\, S)$$

More info: http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/

Finding useful collisions

Target documents are T_1 and T_2:

$$Y_1 = \overbrace{\textbf{preamble}; \text{put}(R_1);}^{x_1} \underbrace{\text{put}(R_1); \text{if}(=) \text{ then } T_1 \text{ else } T_2}_{S}$$

$$Y_2 = \underbrace{\textbf{preamble}; \text{put}(R_2);}_{x_2} \overbrace{\text{put}(R_1); \text{if}(=) \text{ then } T_1 \text{ else } T_2}^{S}$$

Viewing Y_1: $R_1 = R_1$, thus T_1 is displayed.

Viewing Y_2: $R_2 \neq R_1$, thus T_2 is displayed.

Two postscript files

Julius. Caesar
Via Appia 1
Rome, The Roman Empire

May, 22, 2005

To Whom it May Concern:

Alice Falbala fulfilled all the requirements of the Roman Empire intern position. She was excellent at translating roman into her gaul native language, learned very rapidly, and worked with considerable independence and confidence.

Her basic work habits such as punctuality, interpersonal deportment, communication skills, and completing assigned and self-determined goals were all excellent.

I recommend Alice for challenging positions in which creativity, reliability, and language skills are required.

I highly recommend hiring her. If you'd like to discuss her attributes in more detail, please don't hesitate to contact me.

Sincerely,

Julius Caesar

Julius. Caesar
Via Appia 1
Rome, The Roman Empire

May, 22, 2005

Order:

Alice Falbala is given full access to all confidential and secret information about GAUL.

Sincerely,

Julius Caesar

a25f7f0b29ee0b3968c860738533a4b9

SHA-256

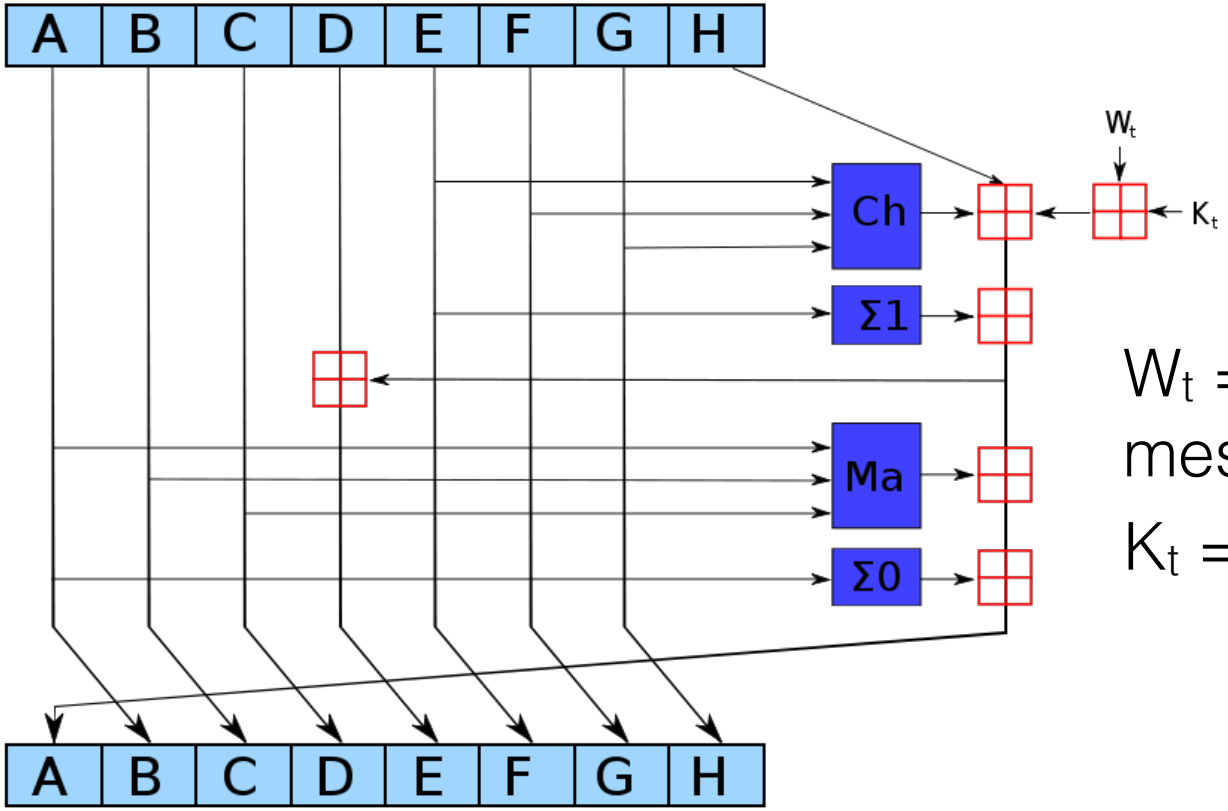
- If you need a one-way hash function, this is the one to use
- 256-bit SHA-2 hash function designed by the NSA

  Should be used as checksum for downloads:

```
$ sha256sum ubuntu-14.04.3-desktop-amd64.iso
756a42474bc437f614caa09dbbc0808038d1a586d172894c1
13bb1c22b75d580  ubuntu-14.04.3-desktop-amd64.iso
```

- Input: 512-bit blocks, divided into 32-bit words
- Output: After 64 rounds, the final result is the sum, by individual words modulo 2^{32}, of the result of this transformation and the original eight word input

One SHA-2 Round



A schematic that shows the SHA-2 algorithm

W_t = t^{th} 32-bit word of the message schedule

K_t = constant

Operations performed in each round

$$\mathrm{Ch}\,(E, F, G) = (E \wedge F) \oplus (\neg E \wedge G)$$

$$\mathrm{Ma}\,(A, B, C) = (A \wedge B) \oplus (A \wedge \mathrm{C}) \oplus (C \wedge \mathrm{C})$$

$$\sum_0 (A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$

$$\sum_1 (E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$$

SHA-256 in LibreSSL / OpenSSL

```
#include <openssl/sha.h>

void sha256(char *string, char outputBuffer[65])
{
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256_CTX sha256;
    SHA256_Init(&sha256);
    SHA256_Update(&sha256, string, strlen(string));
    SHA256_Final(hash, &sha256);
    int i;
    for(i = 0; i < SHA256_DIGEST_LENGTH; i++)
    {
        sprintf(outputBuffer + (i * 2), "%02x", hash[i]);
    }
    outputBuffer[64] = 0;
}
```

Symmetric Key Cryptography

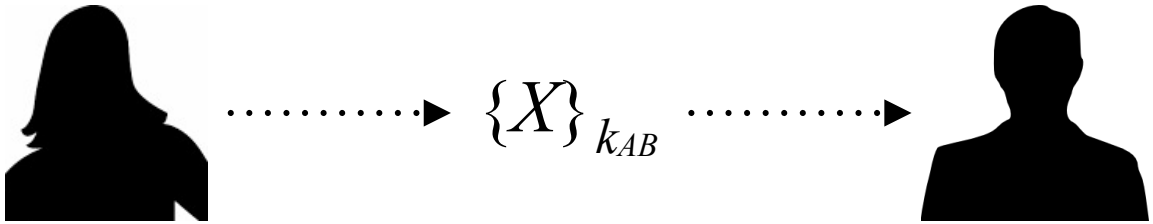
How do two actors communicate securely?

1. Alice and Bob agree on a cryptosystem and key

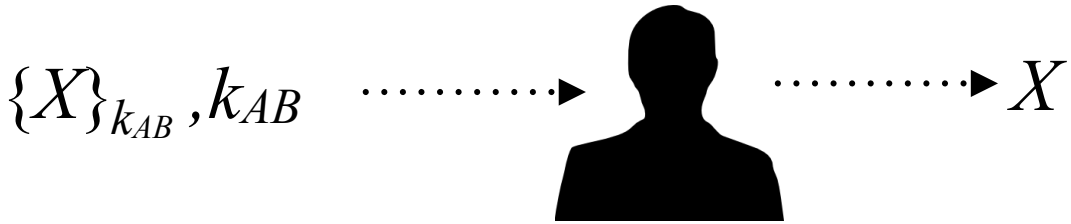
2. Alice encrypts her *plaintext* message using the chosen cryptosystem and key. This creates a *ciphertext* message.

Symmetric Key Cryptography

3. Alice sends the ciphertext message to Bob.

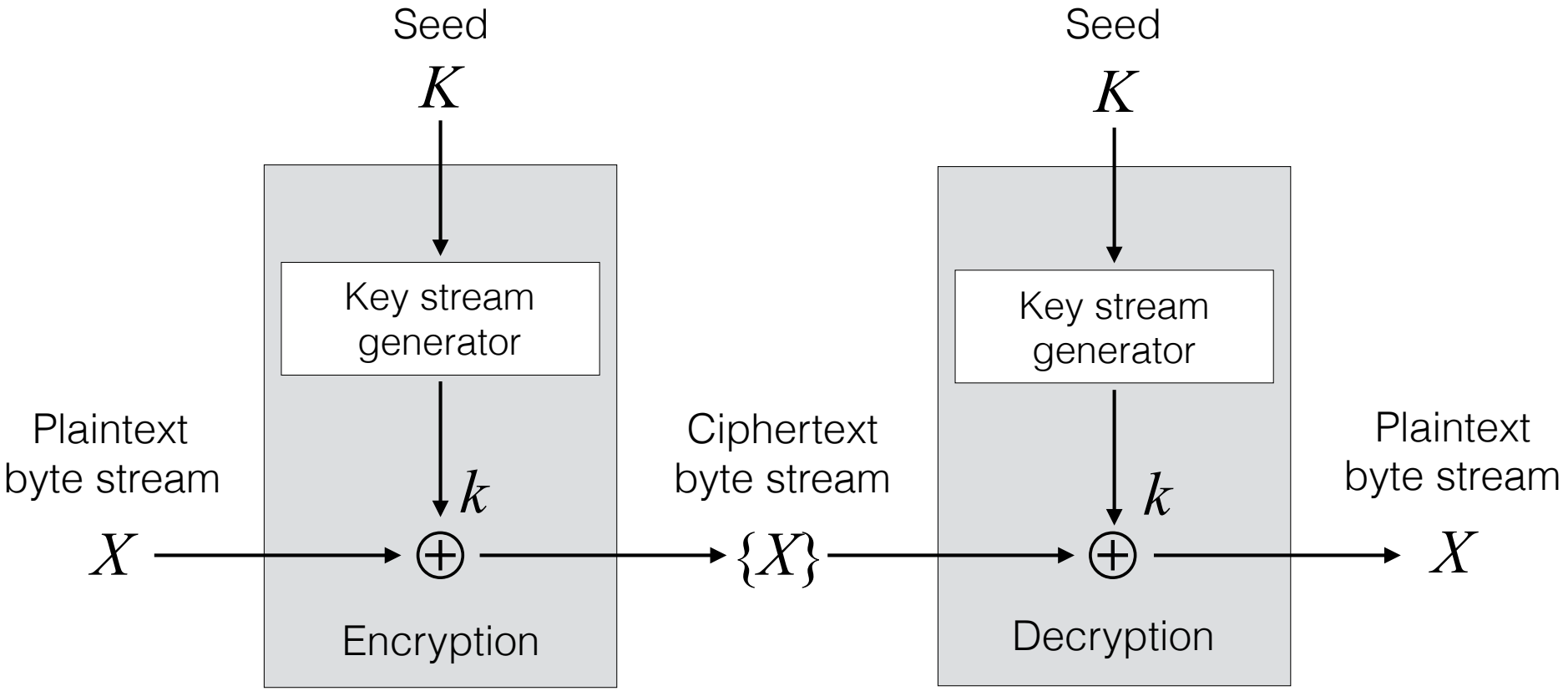


4. Bob decrypts the cipher text message with the same cryptosystem and key



Stream Ciphers

Some encryption algorithms use a non-repeating stream of key elements to encipher characters of a message



Stream Ciphers

Definition: Let E be an encryption algorithm, and let $E_k(X)$ be the encryption of message X with key k. Let a message $X = x_1x_2\ldots$, where each x_i is of a fixed length, and let $k = k_1k_2\ldots$. Then a *stream cipher* is a cipher for which $E_k(X) = E_{k1}(x_1)E_{k2}(x_2)\ldots$.

If the key stream k of a stream cipher repeats itself, it is a *periodic* cipher.

Synchronous Stream Ciphers

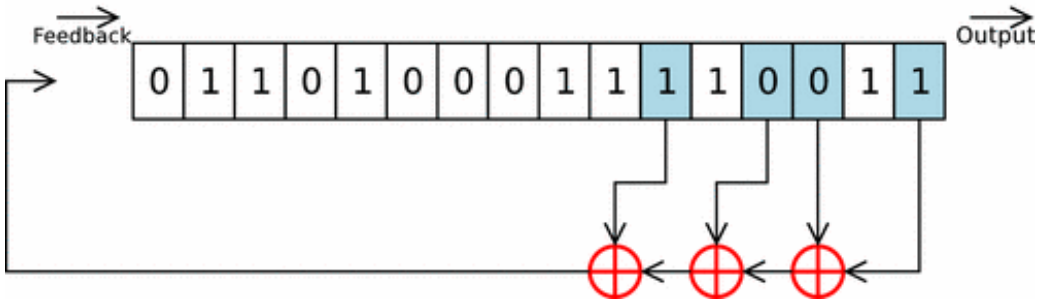
- Goal: simulate a random, infinitely long key
  - Extract bits from a register to use as the key
    - Contents of the register change on the basis of the current contents of the register

Period of 8:

10110001 10110001 10110001…

Linear Feedback Shift Register

Definition: An *n-stage linear feedback shift register* (LFSR) consists of an n-bit register $r = r_0 \ldots r_{n-1}$ and an n-bit *tap sequence* $t = t_0 \ldots t_{n-1}$. To obtain a key bit, r_{n-1} is used, the register is shifted one bit to the right, and the new bit $r_0 t_0 \oplus \ldots \oplus r_{n-1} t_{n-1}$ is inserted.



Linear Feedback Shift Register Example

Tap sequence: 10 01

Init.→

Current register	Key	New bit	New register
00 10	0	01⊕00⊕10⊕01 = 0⊕0⊕0⊕0 = 0	00 01
00 01	1	01⊕00⊕00⊕11 = 0⊕0⊕0⊕1 = 1	10 00
10 00	0	11⊕00⊕00⊕01 = 1⊕0⊕0⊕0 = 1	11 00
11 00	0	11⊕10⊕00⊕01 = 1⊕0⊕0⊕0 = 1	11 10
11 10	0	11⊕10⊕10⊕01 = 1⊕0⊕0⊕0 = 1	11 11
11 11	1	11⊕10⊕10⊕11 = 1⊕0⊕0⊕1 = 0	01 11
01 11	1	01⊕10⊕10⊕11 = 0⊕0⊕0⊕1 = 1	10 11
10 11	1	11⊕00⊕10⊕11 = 1⊕0⊕0⊕1 = 0	01 01
01 01	1	01⊕10⊕00⊕11 = 0⊕0⊕0⊕1 = 1	10 10
10 10	0	11⊕00⊕10⊕11 = 1⊕0⊕0⊕0= 1	11 01
11 01	1	11⊕10⊕00⊕11 = 1⊕0⊕0⊕1 = 0	01 10
01 10	0	01⊕10⊕10⊕01 = 0⊕0⊕0⊕0 = 0	01 10
00 11	1	01⊕00⊕10⊕11 = 0⊕0⊕0⊕1 = 1	00 11
10 01	1	11⊕00⊕00⊕11 = 1⊕0⊕0⊕1 = 0	10 01
01 00	0	01⊕10⊕00⊕01 = 0⊕0⊕0⊕0 = 0	01 00
00 10	0	01⊕00⊕10⊕01 = 0⊕0⊕0⊕0 = 0	00 01

Period of 15, key is 010001111010110

Self-synchronous Stream Ciphers

Some stream ciphers obtain the key from the message itself. Two strategies for this:

1. Draw the key from the plaintext (Vigenère cipher from book):

key: XTHEBOYHASTHEBA

plaintext: THEBOYHASTHEBAG

ciphertext: QALFPNFHSLALFCT

2. Draw the key from the ciphertext:

key: XQXBCQOVVNGNRTT

plaintext: THEBOYHASTHEBAG

ciphertext: QXBCQOVVNGNRTTM

Block Ciphers

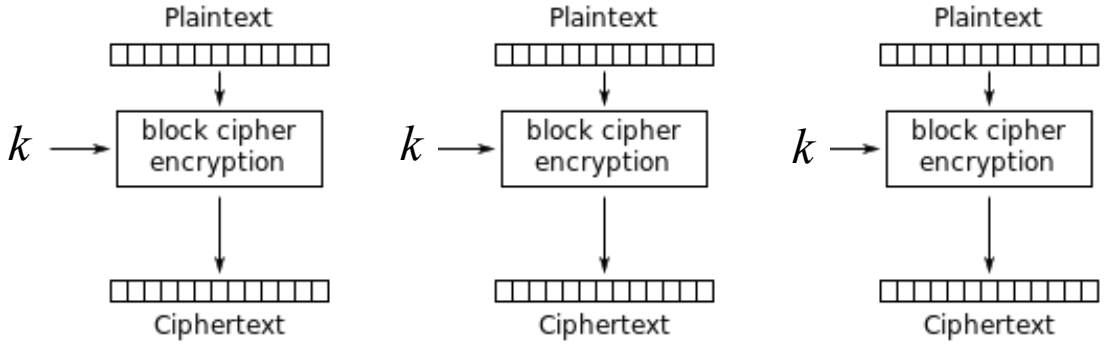
Definition: Let E be an encryption algorithm, and let $E_k(X)$ be the encryption of message X with key k. Let a message $X = x_1 x_2 \ldots$, where each x_i is of a fixed length. Then a *block cipher* is a cipher for which $E_k(X) = E_k(x_1)E_k(x_2)\ldots$.

Example: AES is a block cipher. It breaks the message into 128-bit blocks and uses the same 128-, 192- or 256-bit key to encipher each block

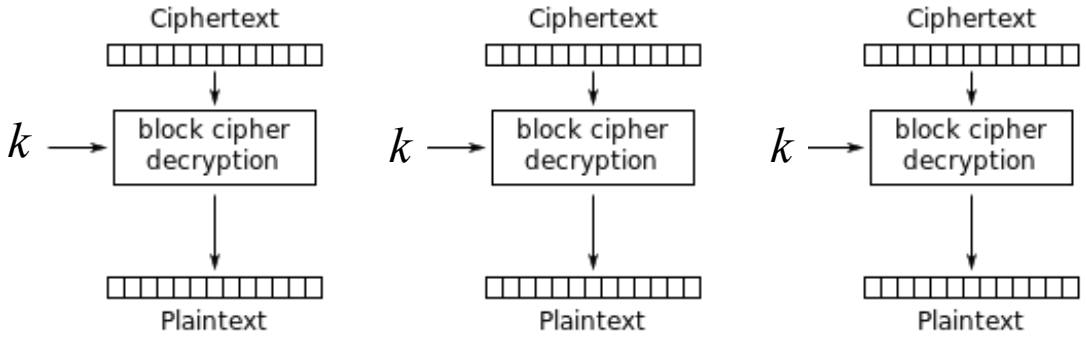
Block Ciphers

Some encryption algorithms divide a message into a sequence of parts, or blocks, and encipher each block with the same key.

Simplest mode of operation (ECB):

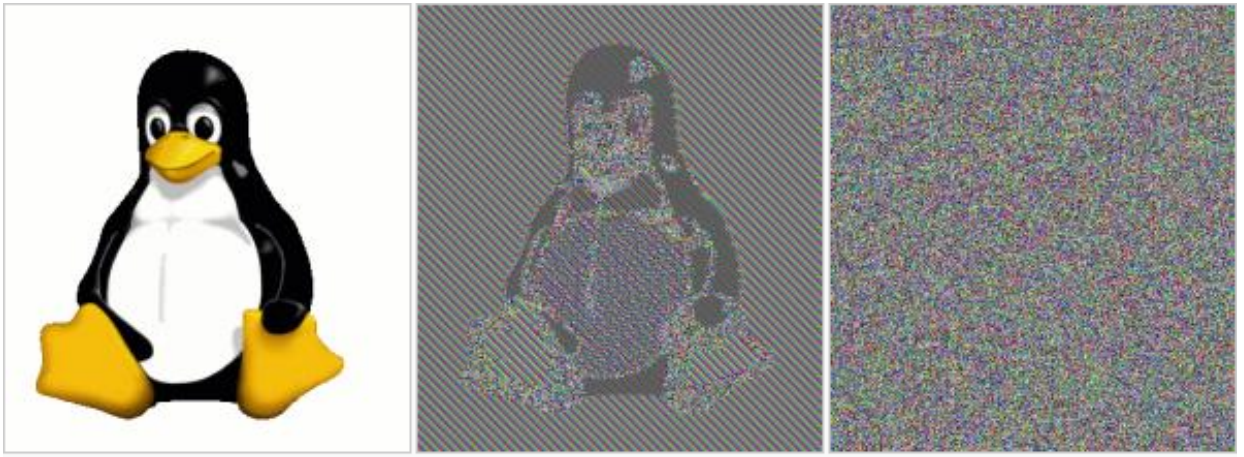


Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

The ECB Penguin



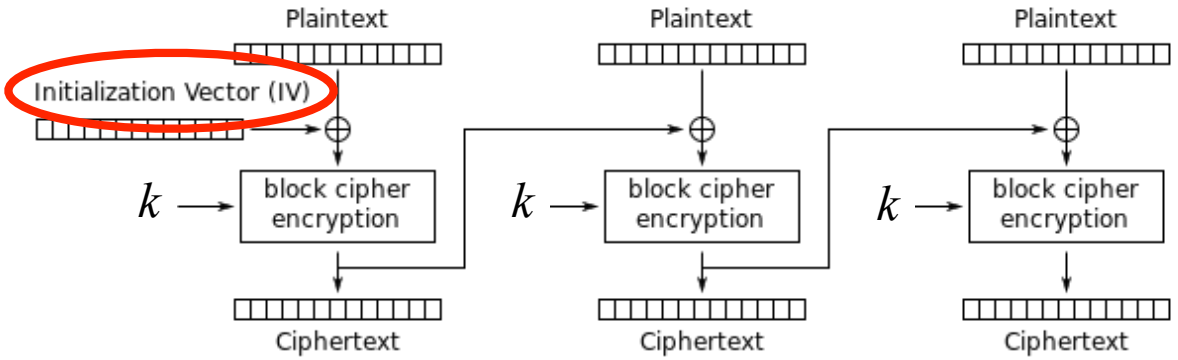
Original image | Encrypted using ECB mode | Modes other than ECB result in pseudo-randomness

Image Credit: https://blog.filippo.io/the-ecb-penguin/

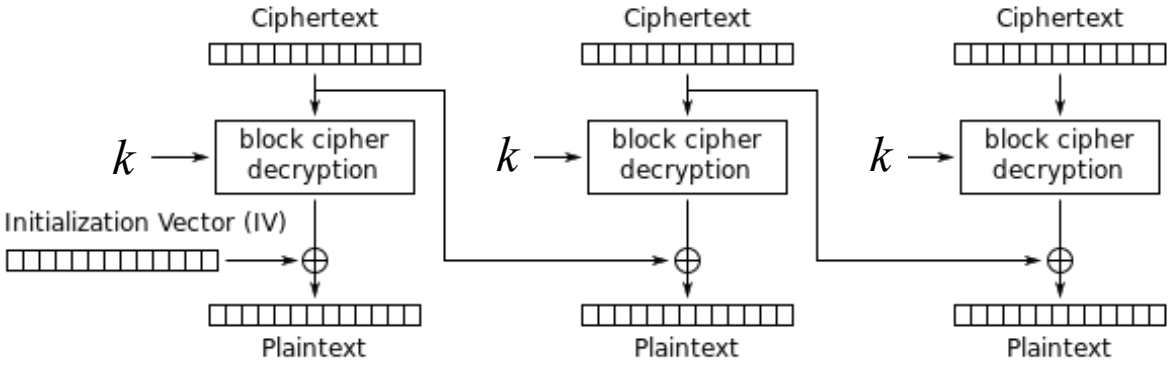
Cipher Block Chaining (CBC)

ECB should not be used if encrypting more than one block of data with the same key (advice: just avoid it)

Alternative: make each ciphertext block dependent on all blocks processed up until that point



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Initialization Vector Pitfall

- Predictable IVs for each transaction
  - CBC mode: enables online attacks with *chosen-plaintext*

Example: Alice's medical record for a specific condition

1. Assume Mallory can predict IVs: IV_M and IV_A
2. Mallory's chosen plaintext: $X_M = IV_M \oplus IV_A \oplus \text{“false”}$
3. Application encrypts: $\{X_M\} = E_k(IV_M \oplus X_M) =$
$$E_k(IV_M \oplus (IV_M \oplus IV_A \oplus \text{“false”}))$$
$$\{X_M\} = E_k(IV_A \oplus \text{“false”})$$
4. Mallory compares ciphertexts: $\{X_M\} = \{X_A\}?$

Choosing Initialization Vectors

Always use a random IV for each transaction

The IV can be made public after encryption.
Why is this secure?

The IV is only used to ensure that the same plaintext encrypts to different ciphertexts. After it is used, there is no harm in releasing it, as it leaks no information about the plaintext.