CYBERSECURITY LAB #4

Giacomo Gori – Tutor didattico

g.gori@unibo.it

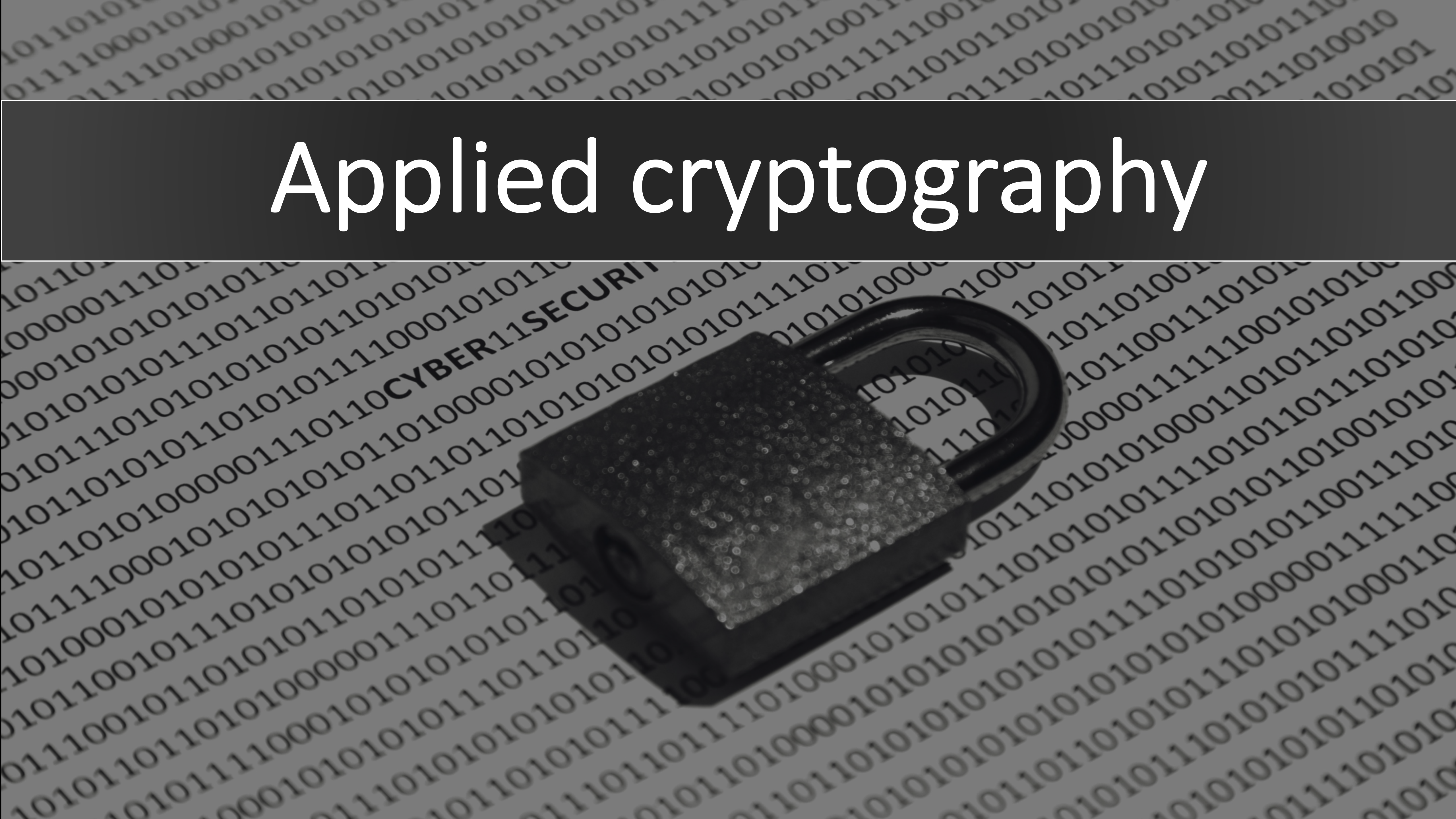
Exercise

Decrypt the files uploaded on virtuale, hints included

Write a **small** report containing **the steps and the decrypted flags** and upload it on Virtuale

Remember: write name, surname and the number of the lab session on the report!

Applied cryptography



AES

(Advanced Encryption Standard)

Symmetric-key algorithm

- Key length: 128, 192 or 256 bits

Block cipher

- Block size: 128 bits

Lightweight

- Low RAM consumptions
- High speed

Block cipher modes

Confidentiality-only modes

- **ECB** (Electronic Code Block)
- **CBC** (Cipher Block Chaining)
- **CFB** (Cipher Feedback)
- **OFB** (Output Feedback)
- **CTR** (Counter)

OpenSSL

- We will use **OpenSSL to play around with crypto algorithms**
- OpenSSL is an open-source library that implements Basic cryptographic primitives
  - Hashing algorithms
  - SSL and TLS protocols
  - Various utilities (prime number generator, PRNG, ...)
- It comes with a handy **command line interface** (CLI)
  - We can do everything from our terminal

OpenSSL
Cryptography and SSL/TLS Toolkit

Basic usage for AES

Encryption of a simple text file using AES-256 in ECB mode:

```
openssl aes-256-ecb -e -in (plaintxt) -out (ciphertxt)
```

Decryption:

```
openssl aes-256-ecb -d -in (ciphertxt) -out (plaintxt)
```

Reasoning about the key..

The key in AES must be 128, 192, 256 bits in length…

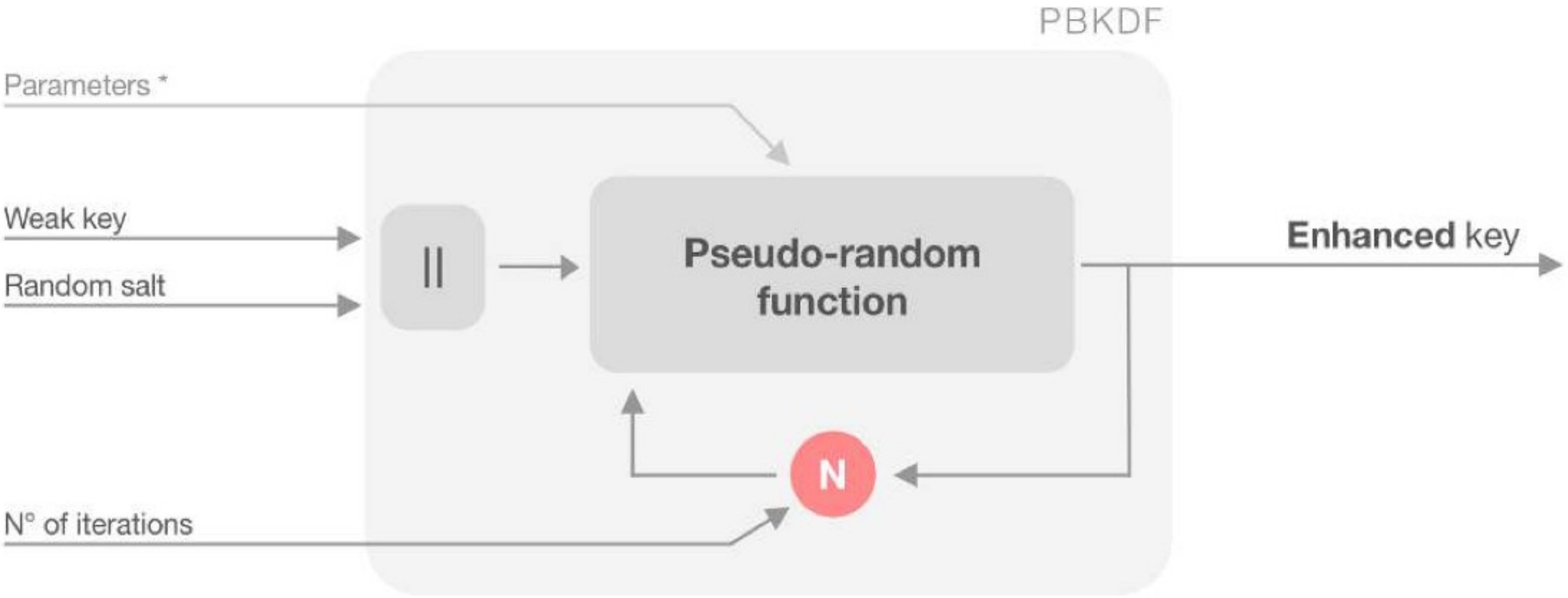
So, can’t we use a human-friendly password to protect our data?

Key concept: Our human-friendly but weak password is used to generate a stronger (enhanced) key with higher entropy

This functions are called **Password Based Key Derivation Functions (PBKDFs)**

In OpenSSL use

- *-p*, to print the actual enhanced key, salt and IV (if used)
- *-nosalt*, to disable the usage of salting to increase the key randomness



By default, OpenSSL **applies a trivial PBKDF**

- If salting is not enabled
  - *key = sha256(passphrase)*
- If salting is enabled
  - *key = sha256(passphrase || salt)*

A better option is to **use more iterations** or **PBKDF2**

- Use the flag *–iter (number of iterations)*, or *–pbkdf2*

Reasoning about the file size..

Size of the plaintext and ciphertext *may* be different

- Ciphertext > plaintext

This happens for two reasons

- **The salt is stored** in the header of the ciphertext (unless –nosalt is used)
- **The plaintext is padded** before being encrypted (*ECB* and *CBC* modes only)
    - *Ciphertext size is always multiple of the cipher block size (128-bit = 16 bytes)*

Visualizing an encrypted file using a normal text editor (or printing on the console) **can't work:**

- The plaintext usually contains ASCII **printable** characters..
- But the ciphertext contains **non-printable** characters

When dealing with such kind of data, we need to view our files using **hexdumps**

- This way, we can visualize binary data encoded in hexadecimal format, e.g.:
  - *0x0a = "\n"*
  - *0x00 = NULL*
  - *0x41 = "A"*

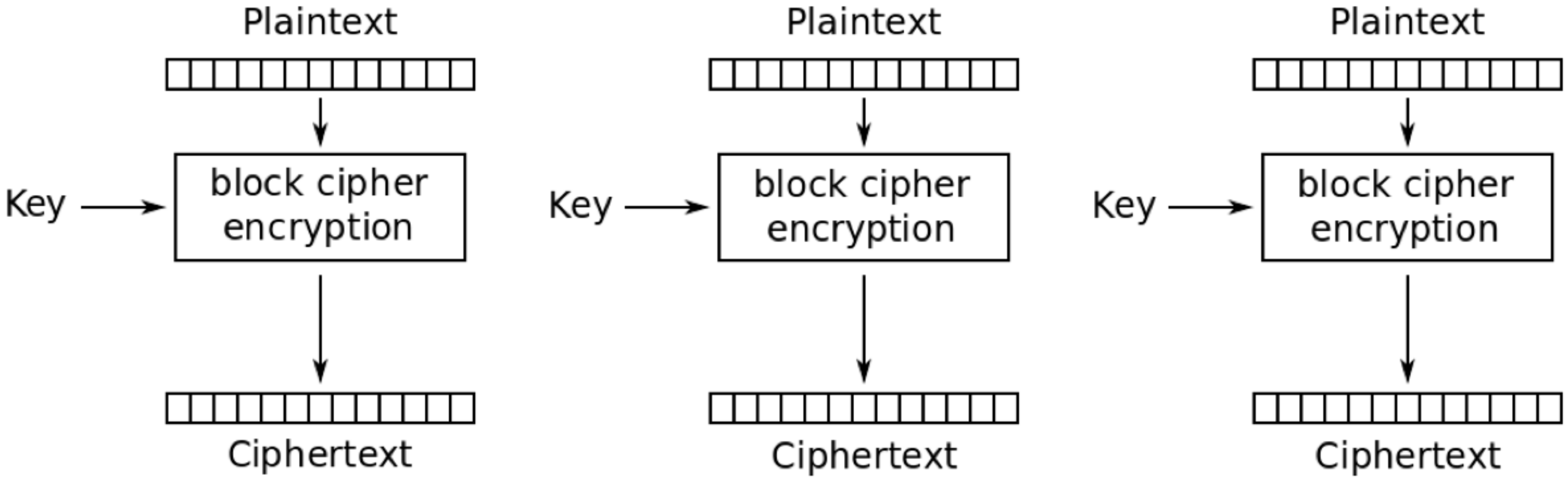
Use **xxd** to visualize the hexdump of a given input file

Weaknesses of ECB mode

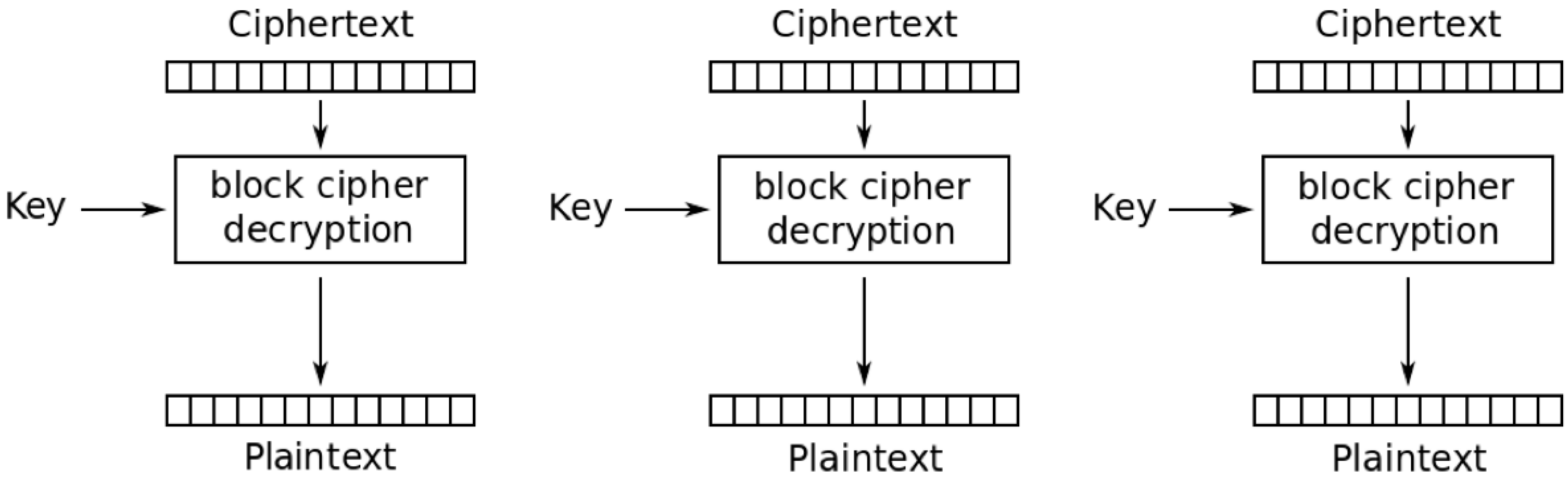
ECB mode

ECB mode **lacks diffusion:**

Identical plaintext blocks produce identical ciphertext blocks



Electronic Codebook (ECB) mode encryption

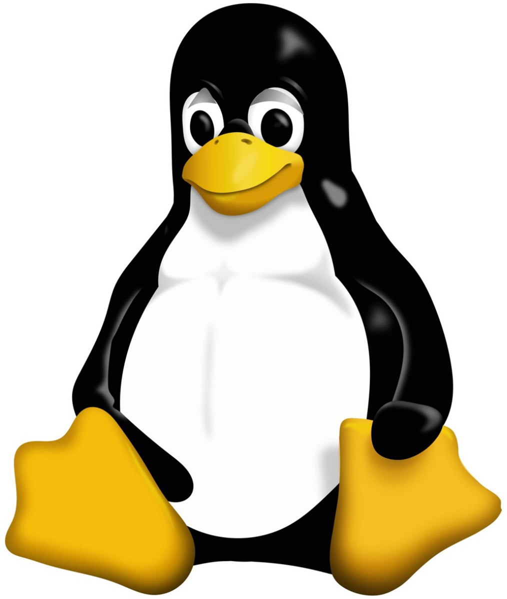


Electronic Codebook (ECB) mode decryption

We can verify this behaviour encrypting a simple bitmap image

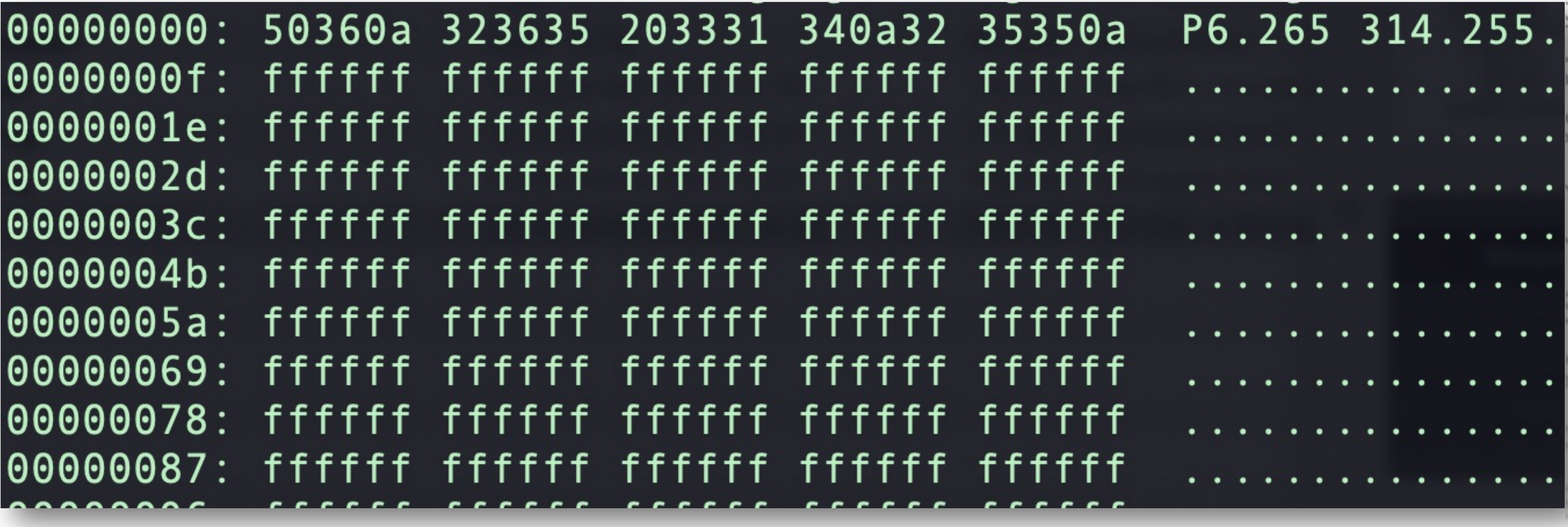
The Tux experiment

- Let’s **encrypt the Linux (*tux*) logo** in **ECB** mode and see what happens
- For the sake of simplicity, the input file will be a simple **bitmap**
  - *.ppm format*



PPM (Portable PixMap) seems a bit exoteric, but in reality **it’s the simplest image format**.

- You can see it using: *xxd –g 3 –c 15 tux.ppm*



```
00000000: 50360a 323635 203331 340a32 35350a  P6.265 314.255.
0000000f: ffffff ffffff ffffff ffffff ffffff  ...............
0000001e: ffffff ffffff ffffff ffffff ffffff  ...............
0000002d: ffffff ffffff ffffff ffffff ffffff  ...............
0000003c: ffffff ffffff ffffff ffffff ffffff  ...............
0000004b: ffffff ffffff ffffff ffffff ffffff  ...............
0000005a: ffffff ffffff ffffff ffffff ffffff  ...............
00000069: ffffff ffffff ffffff ffffff ffffff  ...............
00000078: ffffff ffffff ffffff ffffff ffffff  ...............
00000087: ffffff ffffff ffffff ffffff ffffff  ...............
```

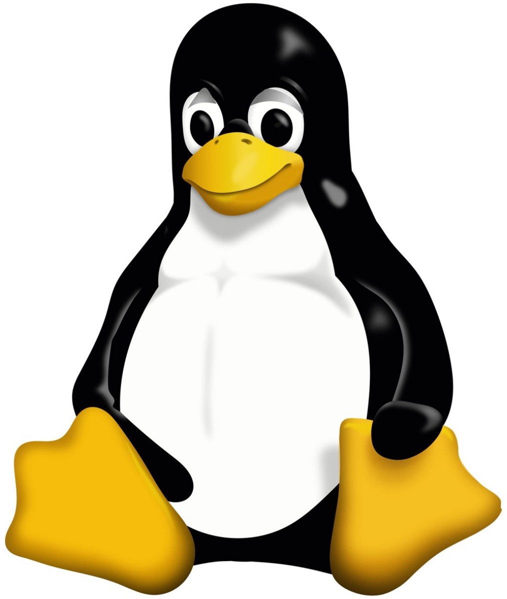
- You may want to install GIMP to view the image
  ```
  sudo apt update && sudo apt install gimp
  ```

- **Split header and body in two different files**
  ```
  head -n 3 Tux.ppm > Tux.header
  tail -n +4 Tux.ppm > Tux.body
  ```

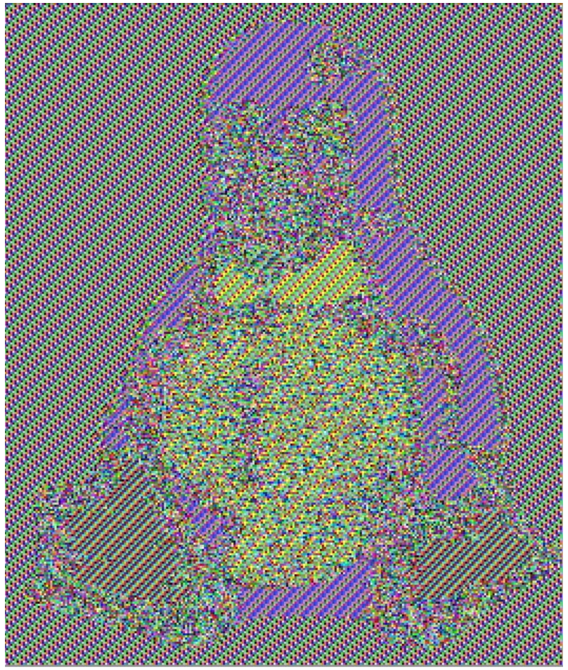
- **Encrypt the body**
  ```
  openssl aes-256-ecb -e -in Tux.body -out Tux.body.ecb
  ```

- **Reassembling everything together**
  ```
  cat Tux.header Tux.body.ecb > Tux.ecb.ppm
  ```

- Now look at the image…. **Is it familiar?**



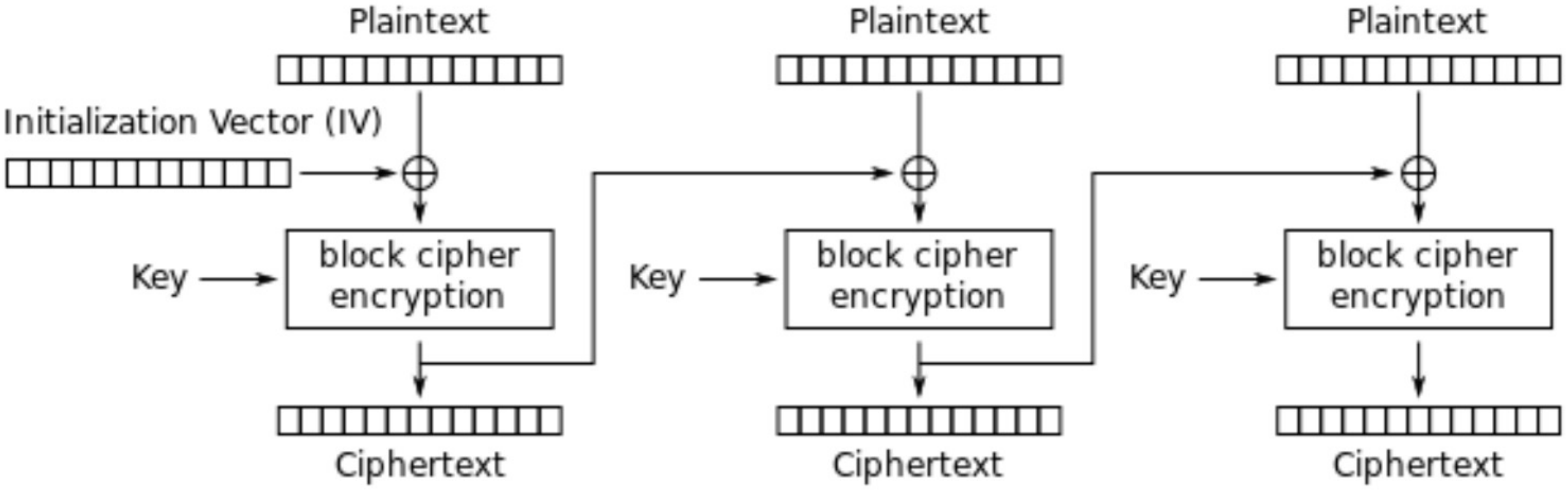
Original



ECB encrypted

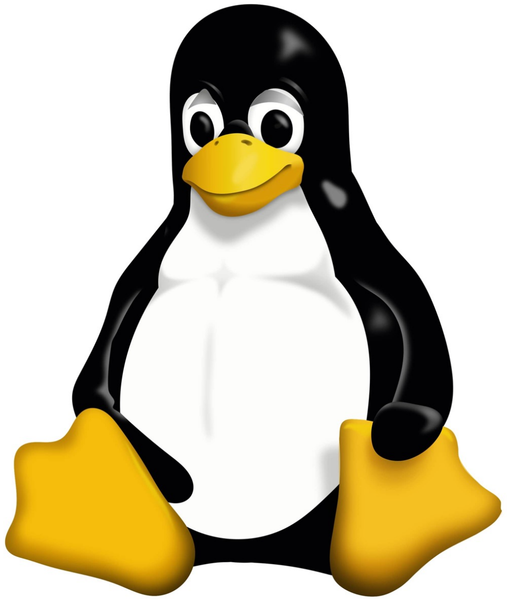
- **Try to repeat the experiment with CBC mode!**
  - *But you must provide an IV with the –iv option*

- **CBC hide away patterns in the plaintext** thanks to the **XOR-ing of the first plaintext block with an IV**, before encrypting it
  - Moreover, it involves **block chaining** as every subsequent plaintext block is XOR-ed with the ciphertext of the previous block

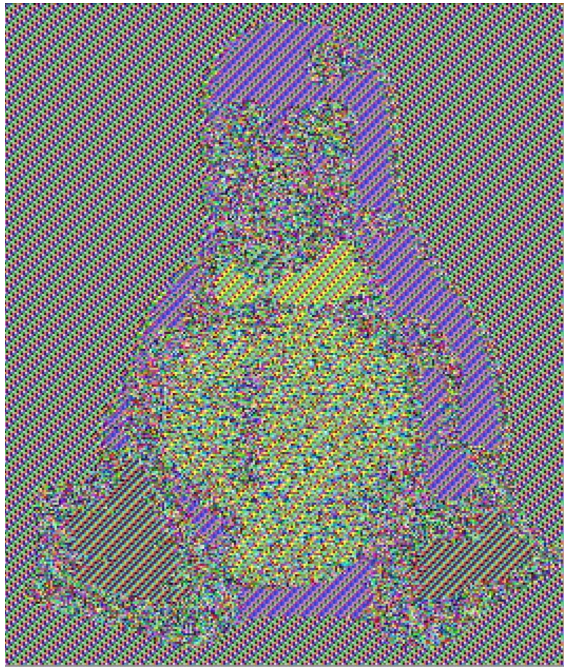


Cipher Block Chaining (CBC) mode encryption

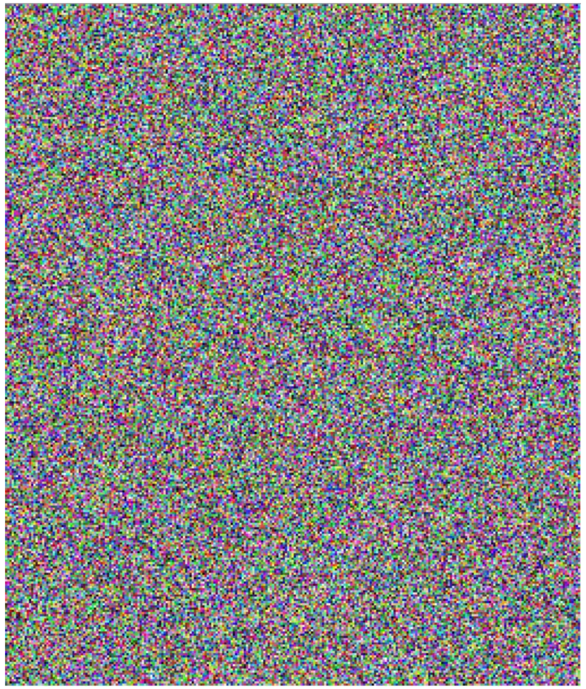
Try **CBC mode** yourself!



Original



ECB encrypted



CBC encrypted

Exercise: decrypt the files

Steps:

1. Download the files on Virtuale
2. Understand the **modes** (CBC,ECB,…)
3. Find the **passwords** and **use them to decrypt**
4. Write steps and the **FLAG** in the report