PV181 Laboratory of security and applied cryptography

Introduction to Applied Cryptography

Part 1, seminar 1: Symmetric cryptography

Łukasz Chmielewski
Email: chmiel@fi.muni.cz

Consultations: A406, 9.00-11.00 on Fridays

Brief Overview

- Motivation / Goals
- Introduction to Cryptographic Primitives:
  - Symmetric vs. Asymmetric
  - RNG, Hash Functions
- Standards / Test Vectors
- Symmetric Cryptography

Motivation

Why do we need it?

- TLS: on our Web-browsers
- Cards: Payment cards

- Wireless communications:
  - Wifi, Bluetooth
- Mobile communications
- Encrypted data storage
- Crypto-currencies

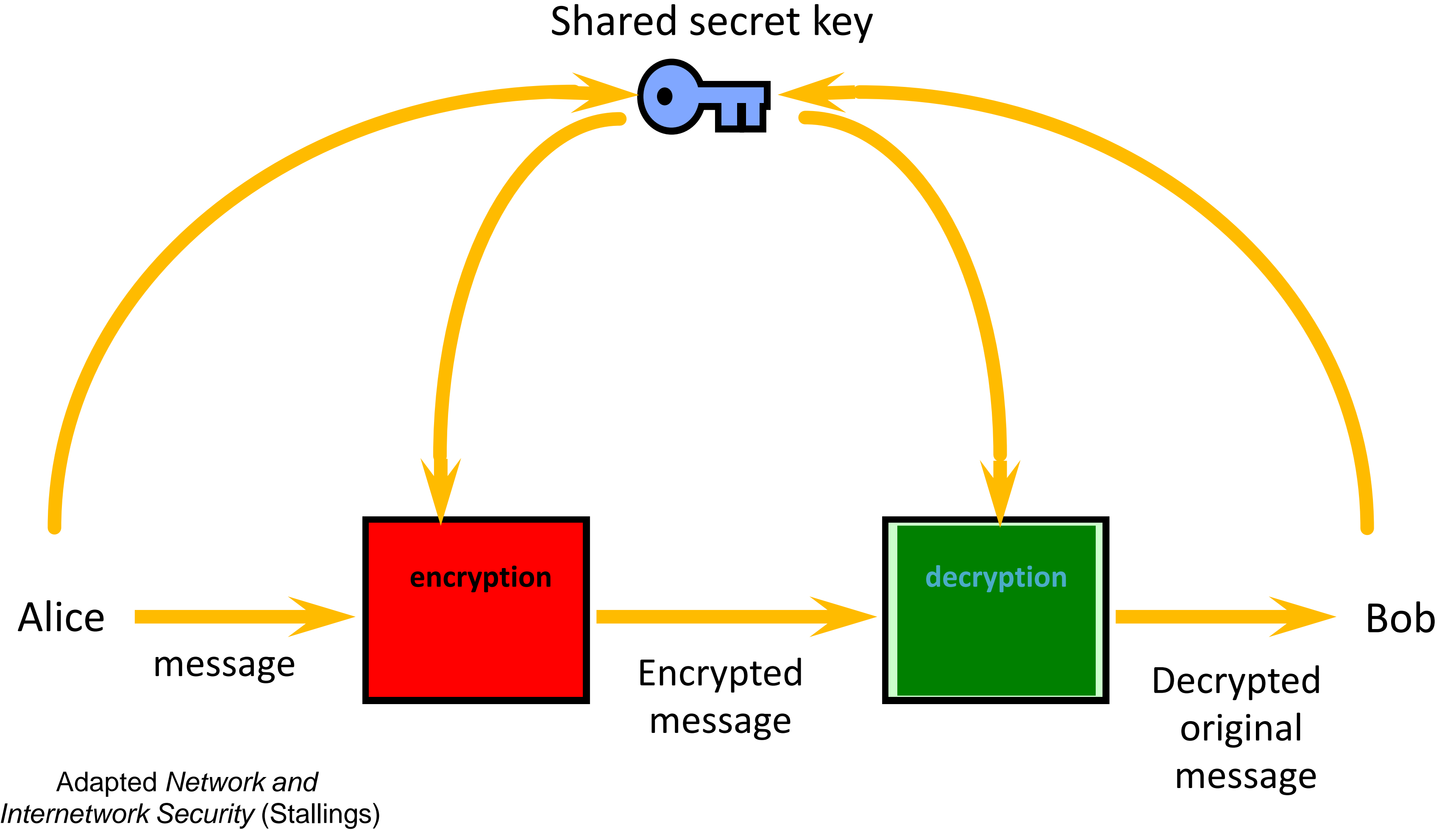
Goals of Cryptography

- Confidentiality (privacy) - preventing open access
  - **ciphers**
- Authentication:
  1. Entity – identity verification – various (password, MAC, …)
  2. Data origin – identity of message originator – **MAC**
- Integrity - preventing unauthorized modification
  - **Hash functions**
- Authentication + Confidentiality:
  - **Authenticated Encryption**
- Non-repudiation - preventing denial of actions
  - **Digital signatures**

Crypto primitives
(covered today)

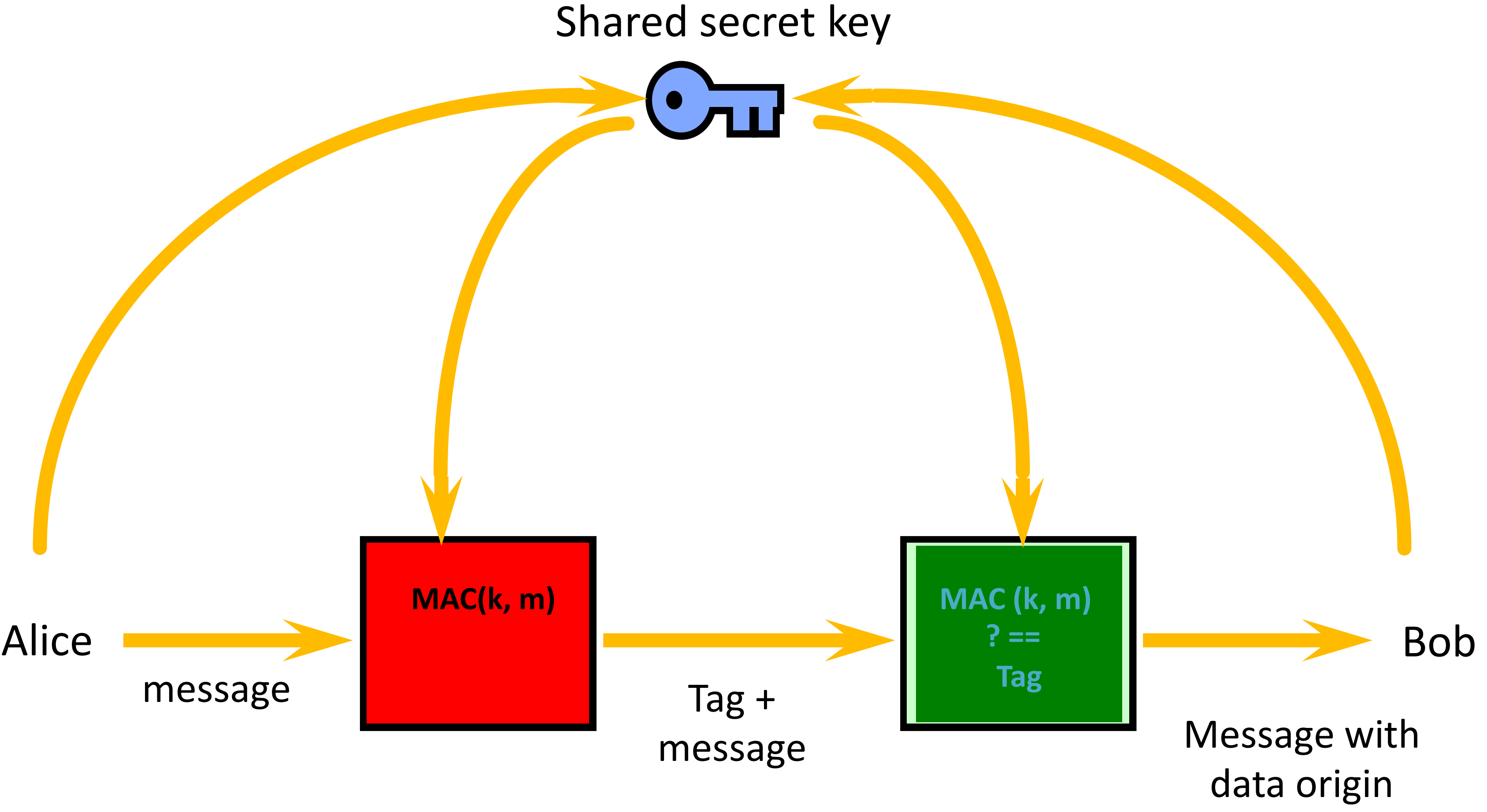
- **Ciphers** – encryption/decryption of data using **key**
  - Symmetric ciphers – **same** key for enc/dec
  - Asymmetric ciphers – **different** key for enc/dec
- **Random number generators** (RNGs)
  - Key generation – covered by Seminar 2.
- **Hash functions** – “unique” fingerprint of data
- Based on other primitives:
  - MAC, PBKDF, Digital signatures
- **Authenticated Encryption** – often Cipher + MAC

Symmetric cryptosystem

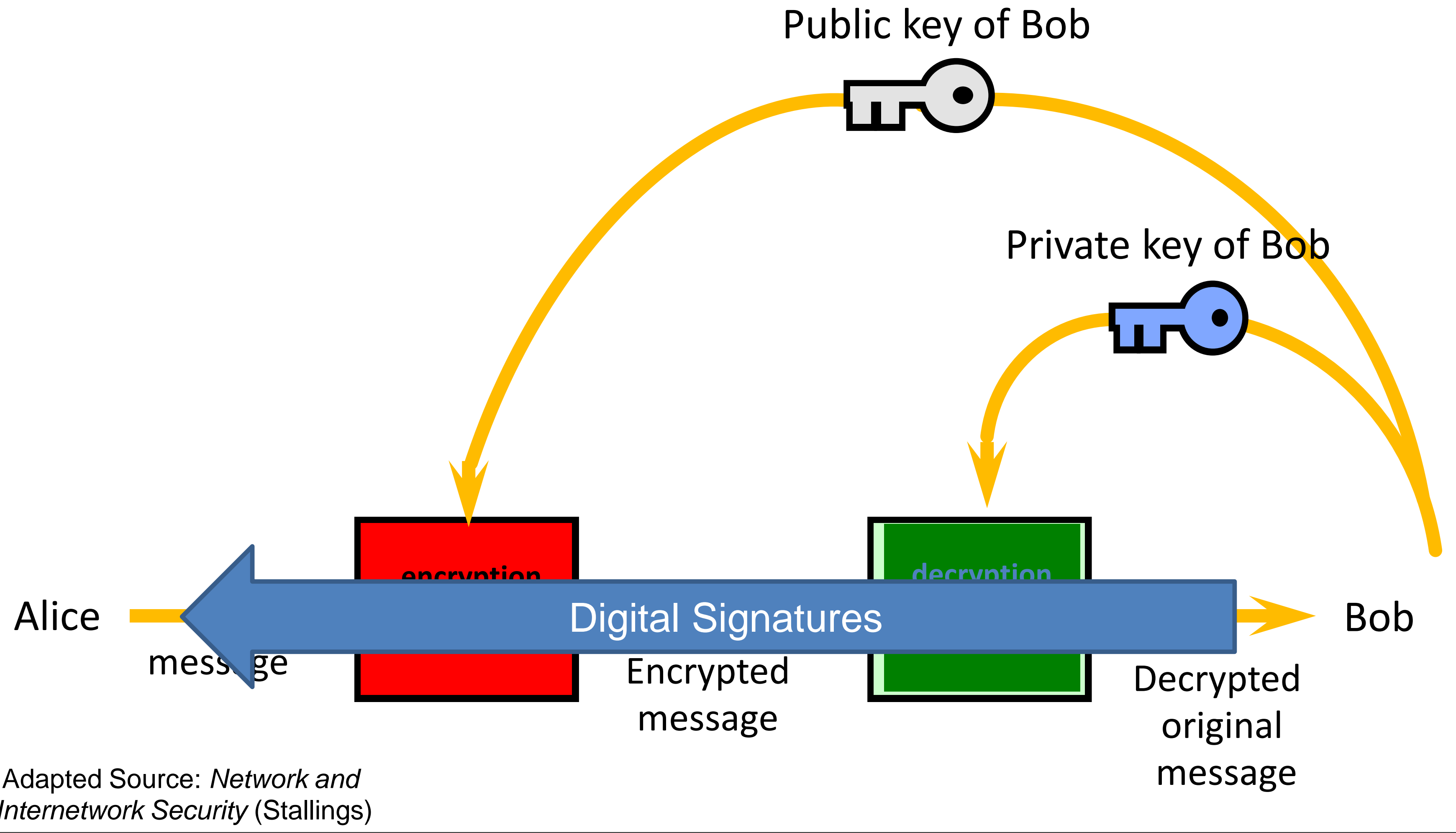


Adapted *Network and Internetwork Security* (Stallings)

Message Authenticated Code (MAC)



Asymmetric cryptosystem



Adapted Source: *Network and Internetwork Security* (Stallings)

Random number generators
(separate seminar)

- Used to generate: keys, IV, …

1. Truly RNG - physical process
   - aperiodic, slow
2. Pseudo RNG (PRNG) – software function
   - deterministic, periodic, fast
   - initialized by **seed** – fully determines random data

- Combination often used:
  - truly RNG used to generate **seed** for PRNG
  - dev/urandom, dev/random in Linux, **Fortuna** scheme

Standards

(separate seminar)

Everything is defined in standards:

- implementation, settings, usage, etc.
- **If you need something look into standard**

Different types:

- FIPS PUB 197 – AES block cipher
- RFC1321 – md5 hash function
- NIST SP,…

Covered by a future seminar.

Implementation testing - test vectors

- Examples of input/output (and also intermediate) for the reference implementation
- MD5 defined in RFC 1321:
  - MD5(“”) = d41d8cd98f00b204e9800998ecf8427e
  - MD5(“message digest”) = f96b697d7cb7938d525…
- AES defined in FIPS197:
  - Plaintext: 00112233445566778899aabbccddeeff
  - Key 000102030405060708090a0b0c0d0e0f
  - Ciphertext 69c4e0d86a7b0430d8cdb78070b4c55a

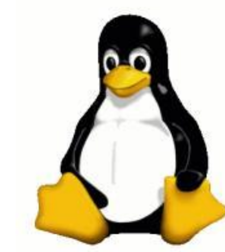
You will try this test vector for AES later!

Symmetric cryptography

Block cipher (1)

- Input divided into blocks of fixed size (e.g 128 bits)
  - Padding - message is padded to complete last block

- Different modes of operation:
  - Insecure basic ECB mode – leaks info
  - Secure modes: CBC, OFB,CFB,CTR, GCM …



- CBC, OFB,CFB need initialization
  - Initialization vector (IV) – must be known

Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Block cipher (2)

- Formally a function: $E : \{0,1\}^n \times \{0, 1\}^k \rightarrow \{0, 1\}^n$
- Usual scenario:

$y_a = E(\ ,\ \text{"Hello, Bob"})$

$D(\ ,y_a) = \text{"Hello, Bob"}$

y_a

y_b

$D(\ ,\ y_b) = \text{"Hello, Alice"}$

$y_b = E(\ ,\ \text{"Hello, Alice"})$

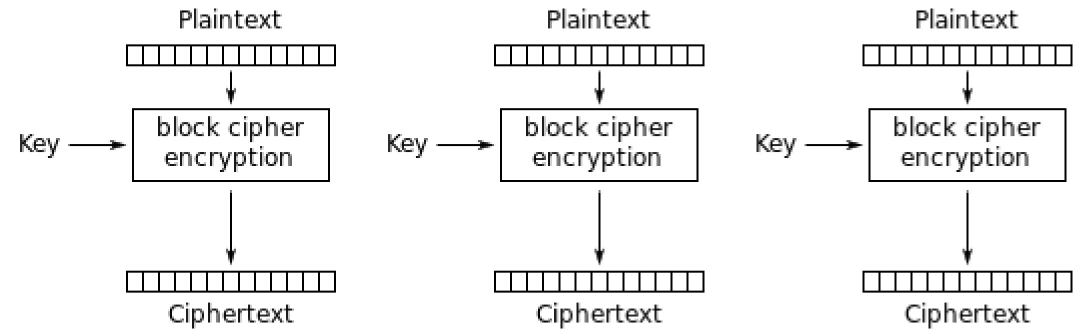
Block ciphers - padding

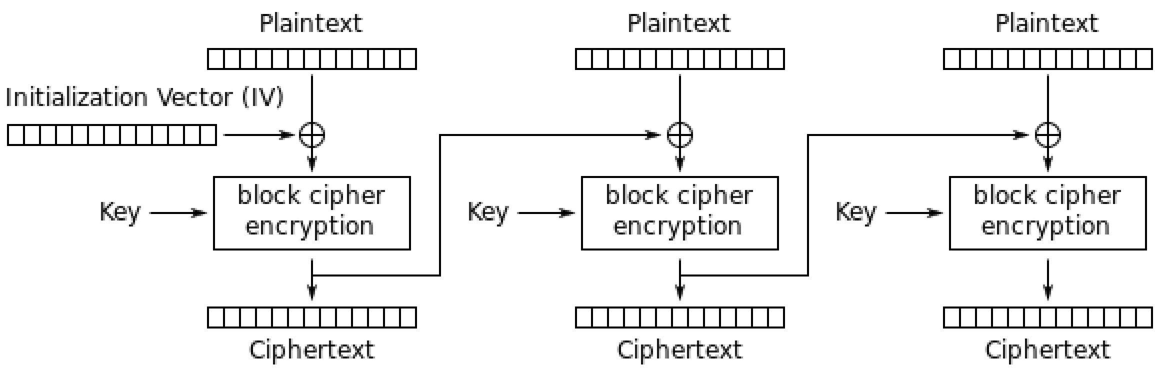
Standard	*method*
ANSI X.923	`... | DD DD DD DD DD DD DD DD | DD DD DD DD` **`00 00 00 04`** `|`
ISO 10126	`... | DD DD DD DD DD DD DD DD | DD DD DD DD` **`81 A6 23 04`** `|`
PKCS7	`... | DD DD DD DD DD DD DD DD | DD DD DD DD` **`04 04 04 04`** `|`
ISO/IEC 7816-4	`... | DD DD DD DD DD DD DD DD | DD DD DD DD` **`80 00 00 00`** `|`
Zero padding	`... | DD DD DD DD DD DD DD DD | DD DD DD DD` **`00 00 00 00`** `|`

What could be the problem here?

Length…

Block ciphers: ECB vs CBC mode

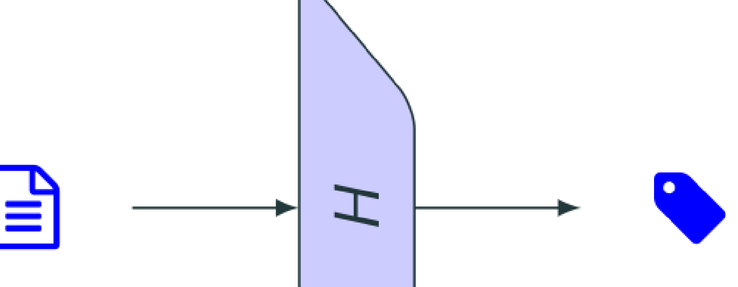




Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Hash function

- **Cryptographic** hash function
  $H : \{0, 1\}^* \rightarrow \{0, 1\}^m$
- Arbitrary input size and fixed output size e.g. 256 bits.



- Function is not injective (there are "collisions").
- Hash is a compact representative of input (also called imprint, (digital) fingerprint or message digest).
- Hash functions often used to protect integrity. First the has is computed and then only the hash is protected (e.g. digitally signed).

Hash functions - examples

- MD5
  - Input: „Autentizace“.
  - Output: 2445b187f422458303788851 1d5411c7 .
  - Output 128 bits, written in hexadecimal notation.
  - Input: „Cutentizace“.
  - Output: cd99abbba3306584e90270bf015b36a7.
  - A single bit changed in input → big change in output, so called “Avalanche effect”
- SHA-1
  - Input: „Autentizace“.
  - Output: 647315cd2a6c953cf5c29d36e0ad14e395ed1776
- SHA-256
  - Input: „Autentizace“.
  - Output: a2eb4bc98a5f71a4db02ed4aed7f12c4ead1e7c98323fda8ecbb69282e4df584

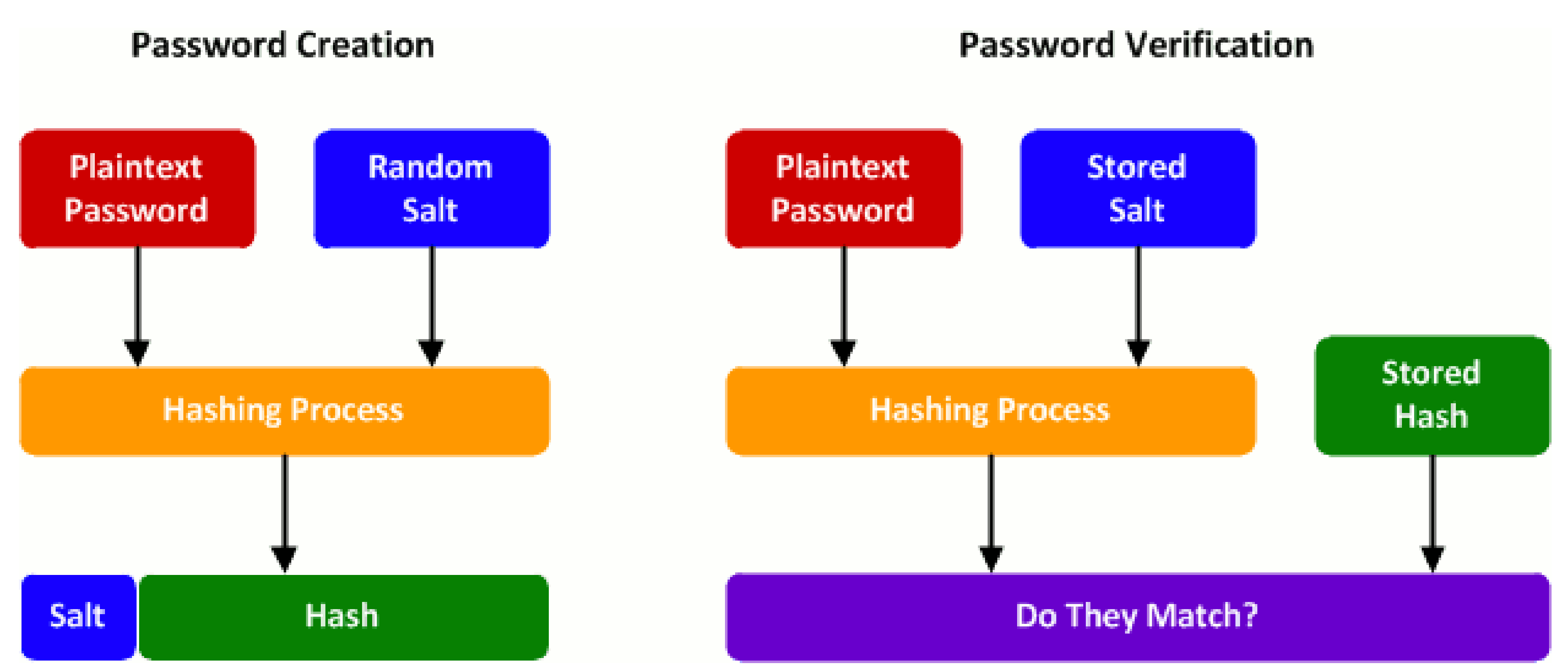
Secure Hash Algorithm (SHA)

- **SHA-1**
  - NIST standard, collision found in 2016, 160 bits hash
- **SHA-2**
  - function family: SHA-256, SHA-384, SHA-512, SHA-224
  - defined in FIPS 180-2
  - Recommended
- **SHA-3**
  - New standard 2015
  - Keccak sponge function family: SHAKE-128, SHA3-224, …
  - defined in FIPS 202, used in FIPS-202, SP 800-185
  - Recommended

Password protection password hashing & salting

1. Clear password could be stolen:
   - store hash of password
     hash = H(password)
   - Checking: password is correct if **hash** matches
2. Attack (brute force or dictionary)
   - trying possible passwords “aaa”, “aab”…“zzz” – N tests
   - N test for single but also for 2,3,… passwords **!!!**
3. Slow down attack - increase password size:
   - random “**salt**” is added to password,

Password protection password hashing & salting

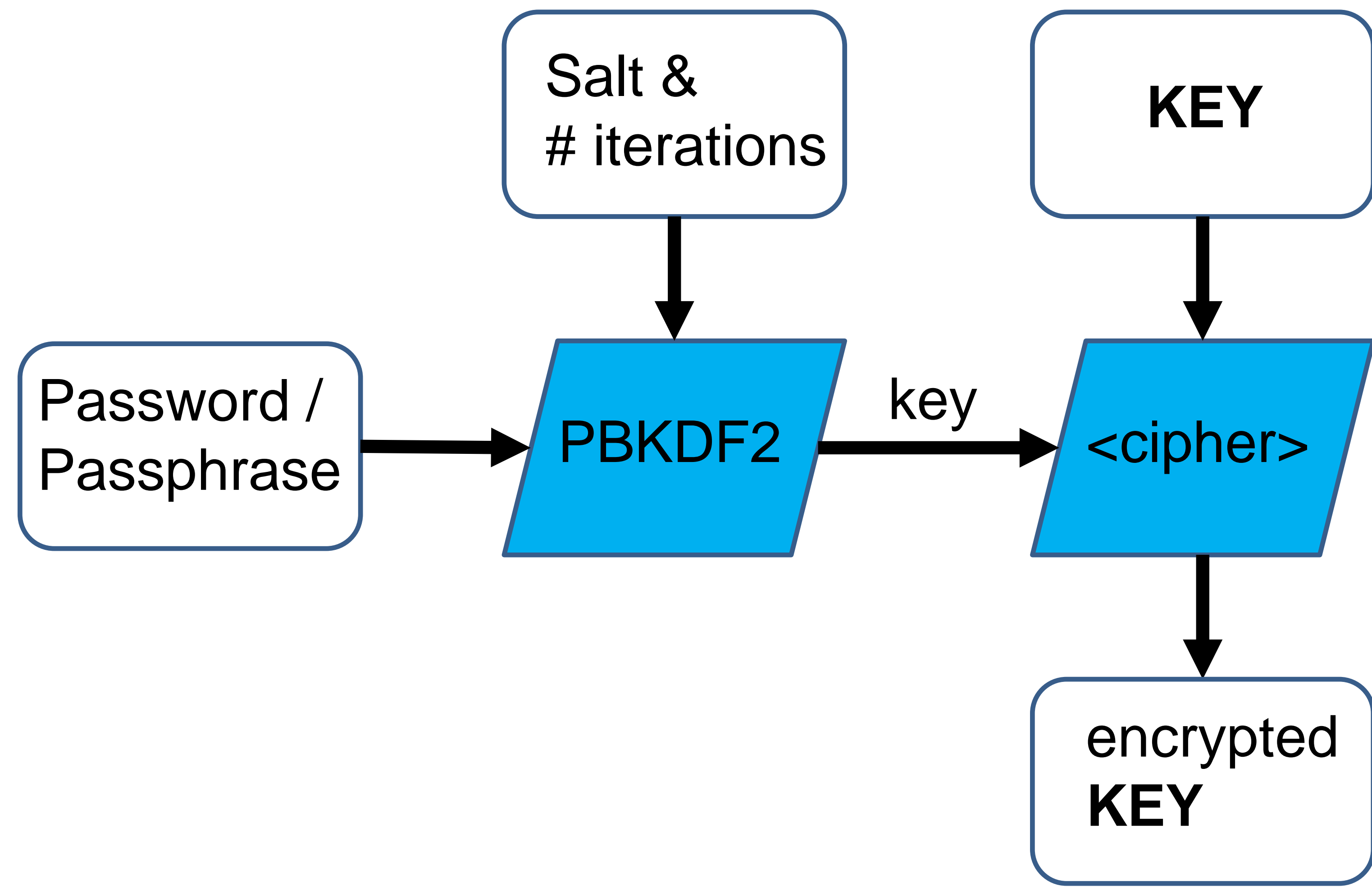


Source: http://blog.conviso.com.br/worst-and-best-practices-for-secure-password-storage/

Key/password protection

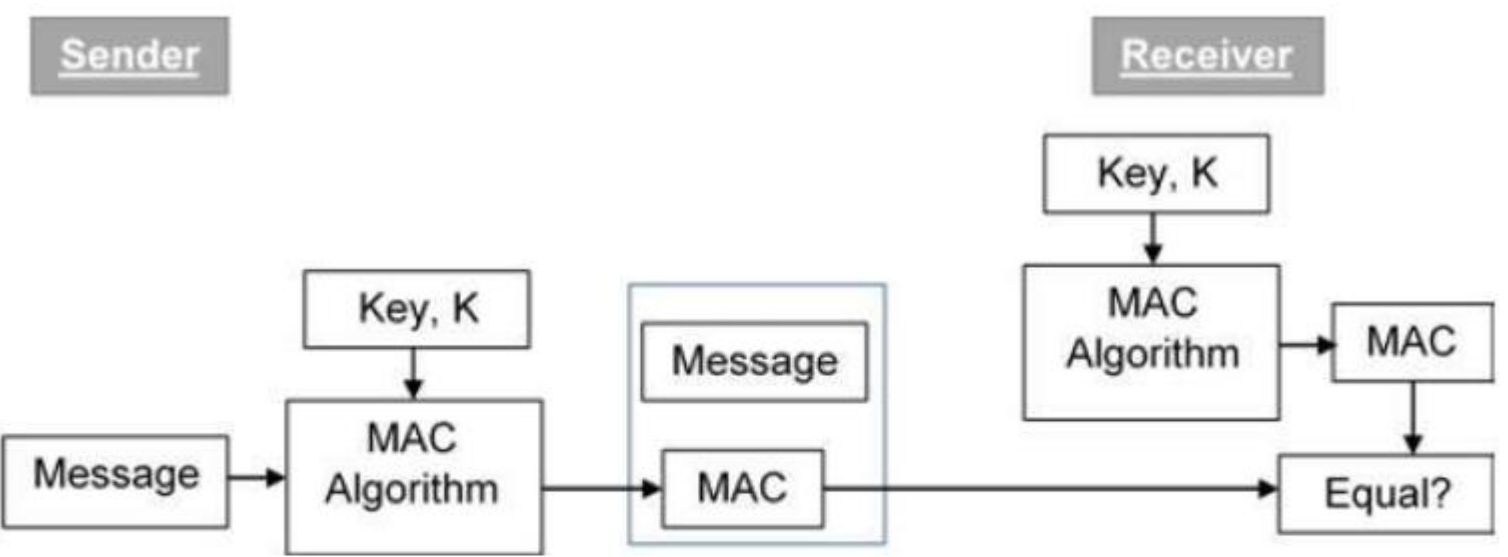
- Encryption (session) key – encrypted using other key – could be derived from password
- Insufficient entropy of passwords
  - Only 200 millions of guesses for ****** password
  - salt protects database of passwords not single password (more info here)
- Password Based Key Derivation Function(PBDKF):
  - 2 types PBDKF2 is newer (see PKCS#5)
  - Slow down hashing (hence attack)
  - Iterate (c times) hash function $K = H^c(pwd \mid salt)$

PBKDF2



Message authentication code (MAC)

- – Based on block cipher (MAC) or hash function (HMAC)
  – Key + message → algorithm → fixed size block MAC



Source: https://www.tutorialspoint.com/cryptography/message_authentication.htm

Authenticated Encryption (AE)

- Only encryption can not provide data integrity

$y_a = E(🔑, \text{"Hello, Bob"})$ — y_a → — $D(🔑, y_a) = \text{"Hello, Bob"}$

$D(🔑, y_b) = \text{"Hello, Alice"}$ — ← y_b — $y_b = E(🔑, \text{"Hello, Alice"})$

  - Why? What happens if y_a is tampered by the adversary?
- Only authentication does not provide confidentiality
- AE provides both: authentication & confidentiality
- Together with ciphertext, a tag/digest is generated from plaintext and/or ciphertext
- Examples: AES-GCM mode, ASCON, etc. For more, see: http://competitions.cr.yp.to/caesar-submissions.html

Practical part

- We will use crypto functions as blackbox
  - No details of the cryptographic algorithms/functions
- We will use the `pyca/cryptography` library in Python
  - version >= 3.6
- A very good documentation: https://cryptography.io/en/latest/
- I have uploaded a simple Python script that we will use:
  - `demo.py`
- Pair up to discuss your results after you complete tasks
  - Some tasks are just analyses, not completing the code.
- @Everyone: try to complete the tasks on your own but
  - Ask your partner or me in case of problems ☺

Extra Information (Hash Functions)

- Available hash algorithms from the library:
  - SHA-2 family, SHA-3 family, Blake2…
- Other weak algorithms:
  - MD5, SHA-1 (against both collision attacks exist)
  - Strongly discouraged to use the above two
- Import the necessary module:
  `from cryptography.hazmat.primitives import hashes`
- Provide the choice of hash algorithm e.g.
  `hashes.SHA256()`
- SHA3 family supports 224, 256, 384 and 512 digest sizes, using `hashes.SHA3_n`, where `n` is the digest size

Extra Information (Encryption with mode)

- Import the modules Cipher, algorithms, modes
- To encrypt with AES a mode of operation must be provided; Also IV (initial value) or nonce is required.
- E.g. : `cipher = Cipher(algorithms.AES(key),modes.CBC(iv))`
- The input data may not be multiple of the block size (e.g. 128) ⇒ padding is required
- ECB or CBC require padding but many other modes: CTR, OFB, CFB and GCM do not (!).
- Padding: `from cryptography.hazmat.primitives import padding`
  - In PKCS7: the value of each byte is the total number of bytes to be added. 01 (for 1 byte padding), 02 02 (for 2 bytes padding) etc.
  - `padding.PKCS7(128).padder()`: here 128 is the size of block in bits

Extra Information (AE & HMAC)

- AES-GCM:
  - `from cryptography.hazmat.primitives.ciphers.aead import AESGCM`
- Associated data: additional data that is authenticated but not encrypted
- For this use: `authenticate_additional_data(add_data)`
- All AEs support key geneneration (`AESGCM.generate_key(128)`) and up-to 2^{32} data size (for both input and associated data).
- AESCCM: AES with counter in CBC mode:
  - `from cryptography.hazmat.primitives.ciphers.aead import AESCCM`
- Hash-based MAC takes an input and a secret key
  - `from cryptography.hazmat.primitives import hmac`

Extra Information (HKDF)

- HMAC-based extract-and-expand KDF (do not use for password storage):
  - `from cryptography.hazmat.primitives.kdf.hkdf import HKDF`
- `HKDF(salt,sourcekey,contxtinfo,L) → K[1]‖K[2]‖...‖K[t]`, where `L` is the output length of HKDF in bits
- Two step process:
  - `PRK = HMAC(salt,sourcekey)`, the first argument is the HMAC key (extract)
  - K[1] = HMAC(PRK, contxtinfo‖0)
  - K[i+1] = HMAC(PRK,K[i]‖contxtinfo‖0), where 1 ≤ I ≤ t (expand)
    Here `t = ⌈L/h⌉` and `K[t]` is truncated to first `L (mod h)` bits and h is the output length of HMAC.

Assignment 1

- This is a programming assignment worth **10 points**. Please upload your (python) scripts via the course webpage.
- The deadline for submission (normally) would be September 27th, 2023, 08:00.
  However, this is assignment 1 so I give time till

  End (23:59) of Friday 29th of September.
- Your answer should be contained in one .py file. Please name the submission file as <uco_number>_ hw1.py.
- It must contain comments so that it is reasonably easy to understand how to run the script for evaluating each answer.
- Soft deadline: -3 points for every started 24 hours

Assignment 1 - Tasks

1. Write a function that can read and encrypt the file alice.txt using AES-GCM with 192 and 256-bit keys. The keys must be cryptographically secure (You can not use the key generating function of AESGCM). The output of the (authenticated) encryption should be written in a file say alice_enc.txt. Call this function from your main function. [2.5 points].
2. Write a function that will decrypt alice_enc.txt. Your function should accept additional data for encryption and the decryption process should return an error if the additional data used is not the same as the one used for encryption. Call this function from your main function. [3 points]
3. Make sure that your encryption and decryption code would work for large files (larger than your RAM). Assume that the file would not fit into your memory. [1.5 points]
4. The key to the above-authenticated encryption should be generated using HKDF with SHA3 256. Run the functions again for this key. [1.5 points]
5. Suppose you want to generate 384 bit key for encrypting two files with two different AES instantiations namely with 128 and 256 bit keys. Write a function that will return two such keys using HMAC. Call the function from the main. [1.5 points]

Good luck!!!