M183 Applikationssicherheit Implementieren
15

By Jürg Nietlispach

Recap # 14

?

Recap # 14

Data Integrity

- Encryption
    - Polyalphabetical Substitution
    - Rotating Substitution
    - DES / AES
    - *One-Time Pad*

«One Time Pad»

In cryptography, the **one-time pad** (**OTP**) is an encryption technique that cannot be cracked, but requires the use of a one-time pre-shared key the **same size as, or longer than, the message** being sent. In this technique, a plaintext is paired with a random secret key (also referred to as *a one-time pad*)

Encryption

```
      H       E       L       L       O  message
   7 (H)   4 (E)  11 (L)  11 (L)  14 (O) message
 + 23 (X)  12 (M)   2 (C)  10 (K)  11 (L) key
 = 30      16      13      21      25     message + key
 =  4 (E)  16 (Q)  13 (N)  21 (V)  25 (Z) (message + key) mod 26
      E       Q       N       V       Z  → ciphertext
```

Decryption

```
      E       Q       N       V       Z  ciphertext
    4 (E)  16 (Q)  13 (N)  21 (V)  25 (Z) ciphertext
 -  23 (X)  12 (M)   2 (C)  10 (K)  11 (L) key
 = -19       4      11      11      14     ciphertext – key
 =   7 (H)   4 (E)  11 (L)  11 (L)  14 (O) ciphertext – key (mod 26)
      H       E       L       L       O  → message
```

«One Time Pad»

Attempts of Cryptoanalysis

- Is the key really random?
- Can the key really be exchanged secretly

Lab – One Time Pad

1. Plaintext Analysis Tool (Letter Frequency)
2. Encryption Engine for a Plaintext
3. Key is dynamically generated and displayed
4. Ciphertext is generated and displayed
5. Decryption Engine using the key

Properties of Symmetric Key Systems

1. The encryption and decryption keys are the same.
2. Communicating parties must have the same key before they can achieve secure communication.

How is the Key exchanged securely?
- Second Communication Channel?
- Trusted Courier?
- **Asymmetric / Public Key System**

Asymmetric / Public Key Systems

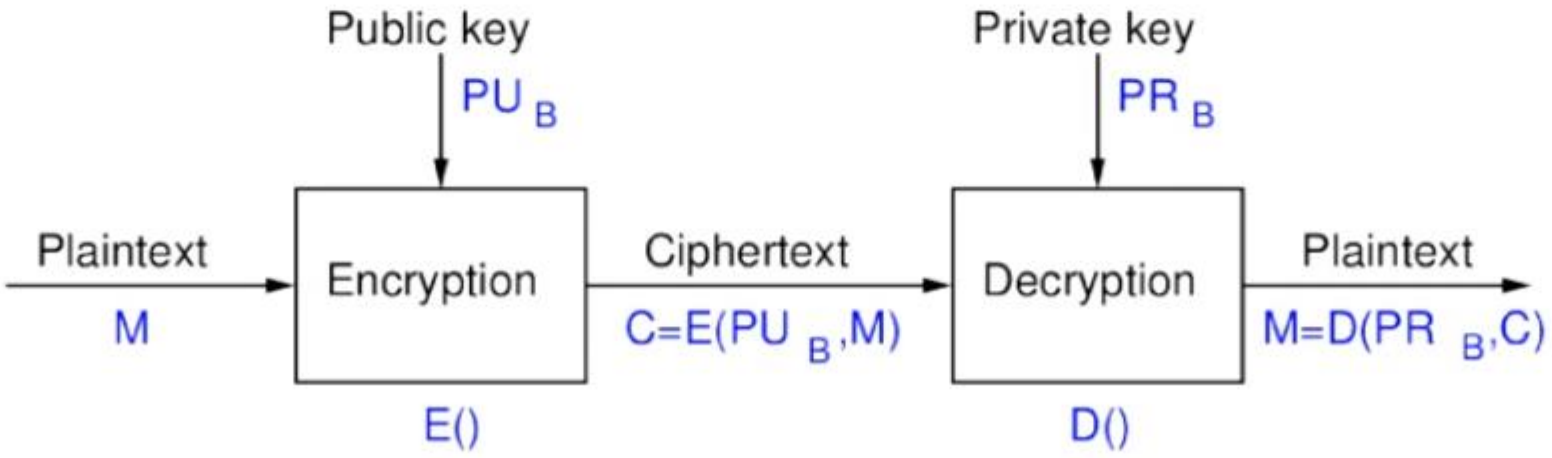
Benefit: no (secure) key exchange necessary!

Idea: Use separate keys for encryption and decryption, one key is publicly accessible!

Examples

- Diffie-Hellmann-Key Exchange, RSA, Digital Certificates

How Public Key Systems Work

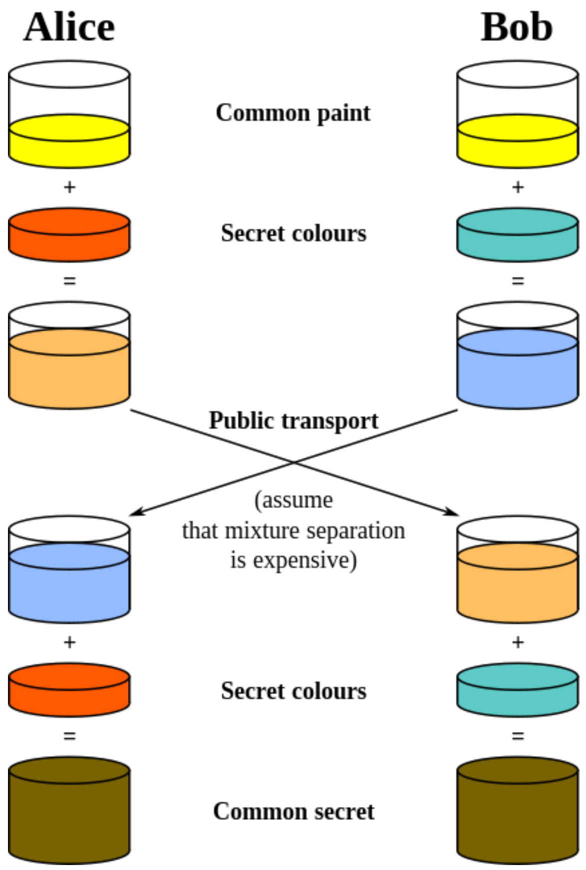


1. Plaintext is encrypted with the public Key of the receiver
2. Ciphertext is decrypted with the private Key of the receiver

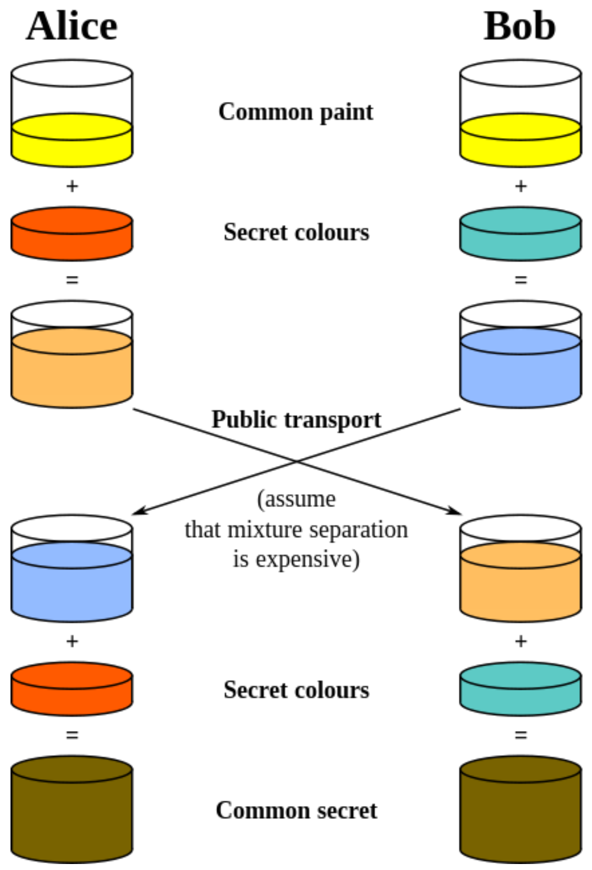
Diffie-Hellmann Key Exchange

The **Diffie–Hellman** key exchange method **allows two parties** that have **no prior knowledge** of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Diffie-Hellmann Key Exchange – Illustration Example



Diffie-Hellmann Key Exchange – Mixed Example 1



1. Alice comes up with two prime numbers **g** and **p** and tells Bob what they are (common paint)

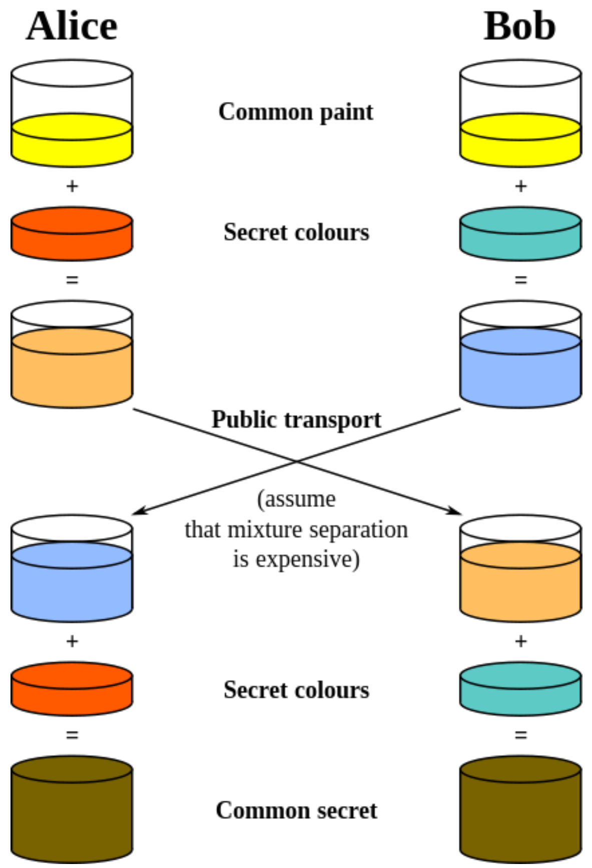
2. Alice then pick a secret number (**a**), but don't tells anyone. Alice computes g^a *mod* **p** **("A")** and send that result back to Bob.

3. Bob does the same thing, but we'll call his secret number **b** and the computed number g^b *mod* **p** ("**B**")

4. Now, Alice takes the number Bob sent and do the exact same operation with *it*. So that's B^a *mod* **p**.

5. Bob does the same operation with the result Alice sent me, so: A^b *mod* **p**.

Diffie-Hellmann Key Exchange – Mixed Example 1



1. Alice wählt zwei Primzahlen **g** und **p** und kommuniziert diese an Bob (gemeinsame Grundlage)

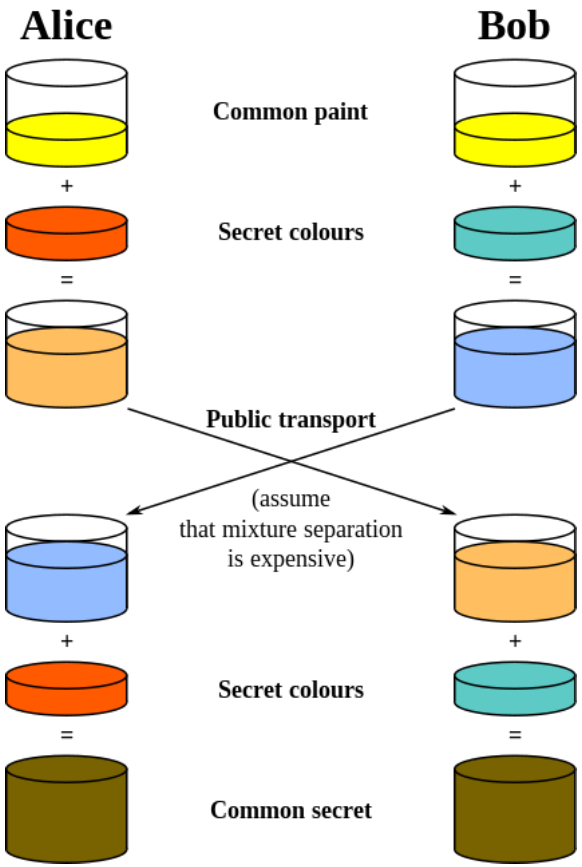
2. Alice wählt dann eine geheime Zahl (**a**) aus, welche nur sie kennt. Alice berechnet dann g^a *mod* **p** **("A")** und sendet die Zahl an Bob.

3. Bob macht dasselbe – er wählt **b** als seine geheime Zahl und sendet g^b *mod* **p** ("**B**") an Alice

4. Alice kann nun Bobs Zahl nehmen und mit ihrer Geheimzahl anreichern: B^a *mod* **p**.

5. Bob macht nun dasselbe (mit Alices Zahl **A**) und erhält so: A^b *mod* **p**.

Diffie-Hellmann Key Exchange – Mixed Example 2



The "magic" here is that the answer I get at step 5 is *the same number* you got at step 4.

Reason:

$$(g^a \mathit{mod}\ p)^b \mathit{mod}\ p = g^{ab} \mathit{mod}\ p$$
$$(g^b \mathit{mod}\ p)^a \mathit{mod}\ p = g^{ba} \mathit{mod}\ p$$

Diffie-Hellmann Key Exchange - Numeric Example

1. Alice and Bob agree to use a modulus $p = 23$ and base $g = 5$ (which is a primitive root modulo 23).
2. Alice chooses a secret integer $a = 4$, then sends Bob $A = g^a \bmod p$
   - $A = 5^4 \bmod 23 = 4$
3. Bob chooses a secret integer $b = 3$, then sends Alice $B = g^b \bmod p$
   - $B = 5^3 \bmod 23 = 10$
4. Alice computes $s = B^a \bmod p$
   - $s = 10^4 \bmod 23 = 18$
5. Bob computes $s = A^b \bmod p$
   - $s = 4^3 \bmod 23 = 18$
6. Alice and Bob now share a secret (the number 18).

Cryptanalysis Diffie-Hellmann

In order to crack the encryption, we have to solve the equation using logarithms after (exponents) a or b (private keys of Alice and Bob) – g and p are given (Public Keys).

$$(g^a \; mod \; p)^b \; mod \; p = g^{ab} \; mod \; p$$
$$(g^b \; mod \; p)^a \; mod \; p = g^{ba} \; mod \; p$$

This is known as the **discrete logarithm problem** which is «**assumed»** (so far) to be very hard to solve when large numbers are used for a and b!

Info: Logjam-Attacks (https://en.wikipedia.org/wiki/Logjam_(computer_security))

Are there any **Known** Hard Problems?

Yes: Integer Factorization!

=> Which is used in the RSA Algorithm

RSA (Rivest-Shamir-Adleman)

Is an asymmetric cryptosystem which can be used for **encryption of data** as well as creating **digital signatures**.

How RSA Works

Basic Principle: Find three very large positive integers *e*, *d* and *n* such that

$$(m^e)^d \equiv m \pmod{n}$$

And: even knowing *e* and *n* (public keys) or even *m* it can be **extremely difficult to find *d***

The RSA algorithm involves four steps:

- Key generation (too complicated),
- key distribution (trivial)
- **encryption and**
- **decryption**

How RSA works – encryption & decryption

Ciphertext (at position i in character code) is calculated as follows:

```
ci_i = pl_i ^ e mod n
```

e = Public Key
n = Part 2 of the Public Key (n = p * q, two secret & large prime numbers)

For Plaintext (at position i in character code), with known private key d

```
pl_i = ci_i ^ d mod n
```

RSA – Example Encryption

P_i = 7 (Plaintext Character in Character Code)
N = 143 (Public Key Part 1)
E = 23 (Public Key Part 2)

=> $C_i = 7^{23} \bmod 143$ => 2

Cryptanalysis RSA

In order to crack the encryption, we have to reverse the following equation after m

$$c \equiv m^e \pmod{n}$$

Given n, e and m^e, it is very difficult to find m (**n-th root problem, factorization problem**)

Know Attacks

- Guessing of the private key (d)
- Cycle Attack (similar to above)
- Common Modulus

Properties of Public Key Systems

1. Can be used for encrypting data with the public key

2. Can be used for sender verification / authentication
   - In case the a message was encrypted with the private key of the sender – the only way to decrypt the message is by using the public key of the sender!

3. Can be used for both (encrypt it with the private key and the public key)!
   - We can ensure, that the message is encrypted and sender is verified!

Problem: Calculations compared to Symmetric Variants 1000x slower!

More Info: https://www.youtube.com/watch?v=GSIDS_lvRv4

Applications

Often used: Hybrid Variants due to speed! I.E. D-H- Key Exchange of a Symmetric Key (AES)

File System Security

- NTFS, PGP

Traffic Security

- TLS / SSL
- SFTP
- Digital Certificates

Database Security

- AES

Lab – SSL / Digital Certificates

Idea
- Create self signed certificate and
- Use it for SSL-Traffic on Localhost

For Windows Users -> see Tutorial

For MAC Users see: https://gist.github.com/nrollr/4daba07c67adcb30693e

Hashing

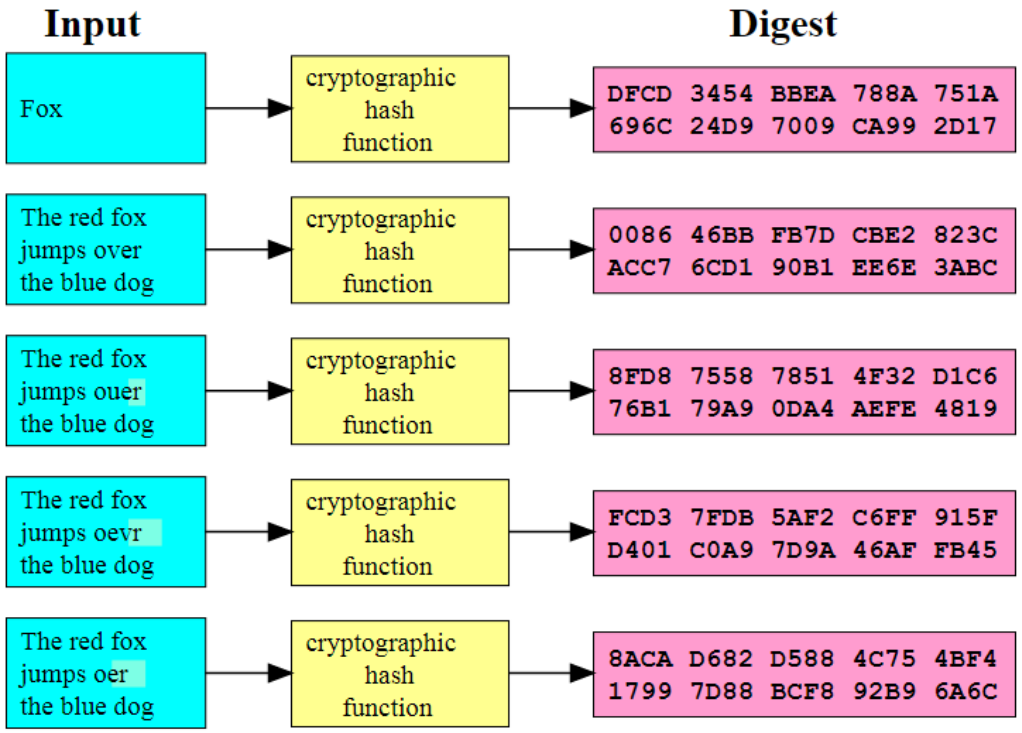
A **hash function** is any function that can be used to map data of arbitrary size to data of fixed size. […]

A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or equivalent alternatives) by knowing the stored hash value.

[…]

This is used for assuring integrity of transmitted data, and is the building block for HMACs, which provide message authentication

Properties of Hash Functions



The ideal cryptographic hash function has five main properties:

- it is **deterministic** so the same message always results in the same hash
- it is **quick** to compute the hash value for any given message
- it is **infeasible** to generate a message from its hash value except by trying all possible messages
- a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
- it is infeasible to find two different messages with the same hash value

Hash Functions

DSA, SHA, SHA-0, SHA-1, SHA-2, SHA-3 …

How SHA works?

```
Initialize variables:

h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0
```

1. Set Initial Hash Values h0-h4

2. Pad the message based on the size (SHA 1 uses 512 Bits)

```
Pre-processing:
append the bit '1' to the message e.g. by adding 0x80 if message length is a multiple of 8 bits.
append 0 ≤ k < 512 bits '0', such that the resulting message length in bits
   is congruent to −64 ≡ 448 (mod 512)
append ml, the original message length, as a 64-bit big-endian integer. Thus, the total length is a multiple of 512 bits.
```

3. Break message in successive 512-bit chunks

How SHA works?

For each chunk

- Break in to sixtee 32-Bit words (0-15)
- Extend sixteen 32-Bit words to 80 – bit words (16-79)
- Do the «scrambling» and update the h0-h4 values (animated:
- https://cyphunk.files.wordpress.com/2006/02/expand_anim.gif?w=237)

```
Initialize hash value for this chunk:
a = h0
b = h1
c = h2
d = h3
e = h4

Main loop:[3][55]
for i from 0 to 79
    if 0 ≤ i ≤ 19 then
        f = (b and c) or ((not b) and d)
        k = 0x5A827999
    else if 20 ≤ i ≤ 39
        f = b xor c xor d
        k = 0x6ED9EBA1
    else if 40 ≤ i ≤ 59
        f = (b and c) or (b and d) or (c and d)
        k = 0x8F1BBCDC
    else if 60 ≤ i ≤ 79
        f = b xor c xor d
        k = 0xCA62C1D6

    temp = (a leftrotate 5) + f + e + k + w[i]
    e = d
    d = c
    c = b leftrotate 30
    b = a
    a = temp

Add this chunk's hash to result so far:
h0 = h0 + a
h1 = h1 + b
h2 = h2 + c
h3 = h3 + d
h4 = h4 + e
```

- And get the final hash value:

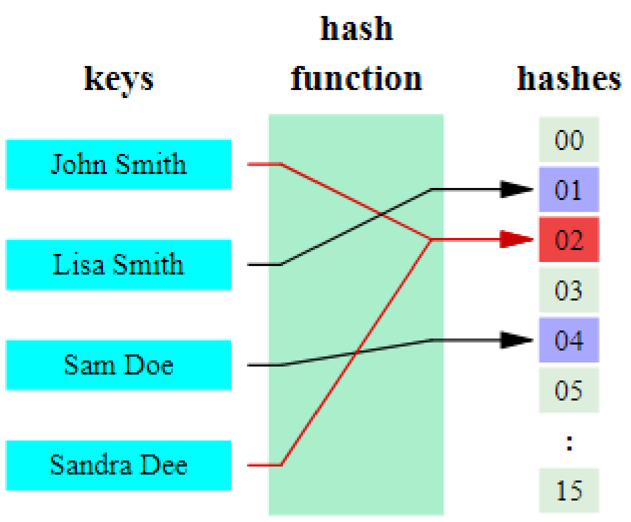
```
Produce the final hash value (big-endian) as a 160-bit number:
hh = (h0 leftshift 128) or (h1 leftshift 96) or (h2 leftshift 64) or (h3 leftshift 32) or h4
```

Further Info: https://www.youtube.com/watch?v=E4FL9Tv-X-k

Applications

- Message Authentication (HMAC)
- Hash Tables
- Caches
- Finding Duplicate Entries
- …

Issues with Hash Functions



By mapping of an arbitrary size of data to fixed size -> **collisions** may appear!

Examples:
- wrong duplicates are detected
- wrong mapping in hash tales
- …

Info:
- a hash function is said to be perfect, if the function is **injective**, i.e. every valid input is mapped to a different hash value.
- There are perfect hash functions: use the data iself (as an integer representation) as the hash value!