Kriptográfia és Információbiztonság
7. előadás

MÁRTON Gyöngyvér

Sapientia Egyetem, Matematika-Informatika Tanszék
Marosvásárhely, Románia
mgyongyi@ms.sapientia.ro

2025

Miről volt szó az elmúlt előadáson?

- az AES blokktitkosító
- hash függvények (hash function)

Miről lesz szó?

- jelszavak biztonsága
- üzenethitelesítő kódok (message authentication code),
- hitelesített titkosítás (authenticated encryption),
- publikus kulcsú rendszerek, alapfogalmak,

Jelszavak biztonsága

- PBKDF2 (password base key derivation function), RFC2898
  - kulcs származtató függvény,
  - egy elsődleges, fő kulcsból egy vagy több másodlagos kulcs létrehozása,
  - egyik szerepe, hogy ha a másodlagos kulcsok kompromitálódnak, akkor az ne veszélyeztesse az elsődleges kulcsot,
  - másik szerep a key streching:
    - megnehezíti a jelszavak feltörését,
    - a jelszóból egy kulcsot származtat,
    - megfelelő paraméterezéssel lehetővé teszi a hashérték kiszámításához szükséges idő meghosszabítását,
- bcrypt
  - salt alkalmazása: lehetővé teszi a jelszavak véletlenszerűsített értékének az eltárolását,
  - a paraméterválasztás lehetővé teszi, hogy lassabban lehessen meghatározni a hashértéket,
- scrypt
  - a salt szerepe hasonló a bcrypthez,
  - a paraméterválasztás lehetővé teszi, hogy nagy memóriaigénnyel lehessen meghatározni a hashértéket.

bcrypt

```
from Crypto.Hash import SHA3_256
from base64 import b64encode, b64decode
from Crypto.Random import get_random_bytes
from Crypto.Protocol.KDF import bcrypt

def createPassword(name, paswd):
    salt = get_random_bytes(16)
    print(salt)
    hashS = SHA3_256.new(paswd.encode())
    hashBcrypt = bcrypt(hashS.digest(), 4, salt)
    keyD = ['name', 'password', 'salt']
    valueD = [name, b64encode(hashBcrypt).decode(), b64encode(salt).decode()]
    person = dict(zip(keyD, valueD))
    return person
```

bcrypt

```
def checkPassword(person, paswd):
    salt = b64decode(person['salt'].encode())
    hashS = SHA3_256.new(paswd.encode())
    hashBcrypt = bcrypt(hashS.digest(), 4, salt)
    print(b64encode(hashBcrypt))
    print(salt)
    if hashBcrypt == b64decode(person['password'].encode()):
        print('Valid!!!', person['name'], 'password: ', paswd)
    else:
        print("Failed!!!", person['name'])


person = createPassword('Szitai Emoke', 'something')
print(person)
checkPassword(person, 'something')
```

Üzenet-hitelesítő kódok (Message Authentication Code)

- üzenetek integritásának a vizsgálatára szélesebb körben elfogadott módszer a MAC-ek alkalmazása
- a fogadó fél le tudja ellenőrizni a küldő fél identitását, illetve az üzeneten végzett szándékos (rosszindulatú) változtatásokat
- a MAC egy titkos információ (titkos kulcs) alapján tetszőleges hosszúságú bitsorozatból fix hosszúságú bitsorozatot állít elő
- nem biztosít titkosítást ezért a MAC függvény inverz függvényének algoritmusát nem kell megadni $\Rightarrow$ kevésbé sérülékeny
- szükséges a MAC-hez használt titkos kulcs előzetes, biztonságos megosztása

Üzenet-hitelesítő kódok (Message Authentication Code)

- szerkesztése:
  - hash függvény alkalmazásával (pl. HMAC, 1996, használják az SSL protokollban, a NIST elismerte standardnak)
  - blokk-titkosító alkalmazásával (pl. CMAC: 3DES vagy AES alkalmazása)
- alkalmazási terület:
  - pl. a windows rendszerállományok esetében annak az eldöntése, hogy módosultak-e vagy sem valamilyen vírus stb. által az állományok
  - egy számítógép-program hitelességének az ellenőrzése sokkal kényelmesebben megoldható MAC-függvények segítségével, a számítógép-program titkosítása helyett

Üzenet-hitelesítő kódok, alkalmazási terület

Miért van szükség MAC-re, miért nem alkalmasak a titkosítók?

- ugyanazt az üzenetet több különböző helyre küldjük: pl. értesíteni kell a felhasználókat, hogy a hálózat nem elérhető, vagy riasztójelzést kell küldeni egy katonai irányítóközpontba. Elég ha az üzenetet, a MAC-el együtt küldjük, nincs szükség titkosításra
- ha nagy adatforgalomat kell kezeljen egy rendszer akkor az összes üzenet visszafejtése csak fölöslegesen terhelné a rendszert
- alkalmazások integritásának az ellenőrzésére is elég a MAC
- jó külön választani az integritásvizsgálatot a titkosítástól: az integritásvizsgálatot az alkalmazások szintjén a titkosítást a szállítási rétegben érdemes implementálni

Üzenet-hitelesítő kódok, matematikai modell

- legyen P az üzenetek és K a kulcsok halmaza
- a kulcs generálás algoritmusa polinom idejű, véletlenszerű: $Gen \rightarrow key$, úgy hogy $key \in K$
- a MAC-érték meghatározásának algoritmusa polinom idejű, véletlenszerű, ahol $m \in P$-re meghatározzuk: $x = MAC_{key}(m)$
- a MAC-érték ellenőrzése determinisztikus algoritmussal történik, ahol az m, key, x értékek alapján először meghatározzuk a $MAC_{key}(m)$ értéket és ellenőrizzük, hogy ez egyenlő-e x-el?

Üzenet-hitelesítő kódok, követelmények

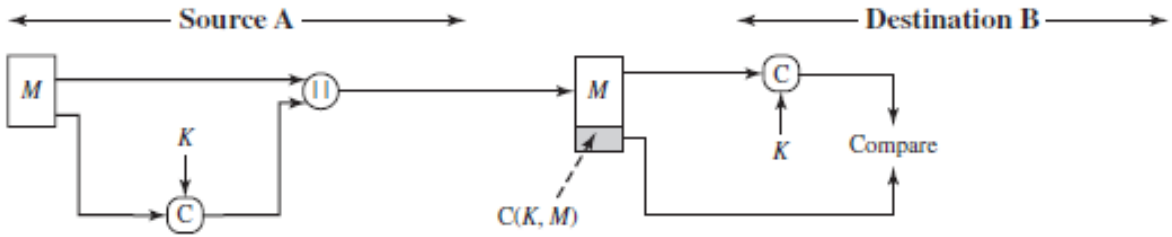
- egy adott MAC-függvény esetében a következő feladatok ne legyenek megoldhatóak polinomiális időben:
  - több m_i és $MAC_{key}(m_i)$ pár ismeretében, határozzuk meg $MAC_{key}(m)$-t, ahol $m \neq m_i$, és ahol a *key* értéke sem ismert
  - több m_i és $MAC_{key}(m_i)$ pár ismeretében határozzuk meg a *key*-t

Megjegyzések:

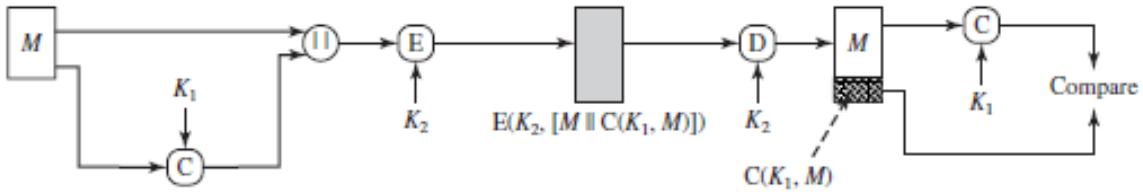
- ha a MAC-függvény eleget tesz az első követelménynek, akkor eleget tesz a másodiknak is. Fordítva azonban nem igaz, mert nem kizárt, hogy egy támadó, egy m üzenethez, a *key* ismerete nélkül is tud szerkeszteni egy érvényes MAC-értéket,
- sokkal több támadási módszer létezik, mint a hash-függvények esetében.

Üzenet-hitelesítő kódok, használat

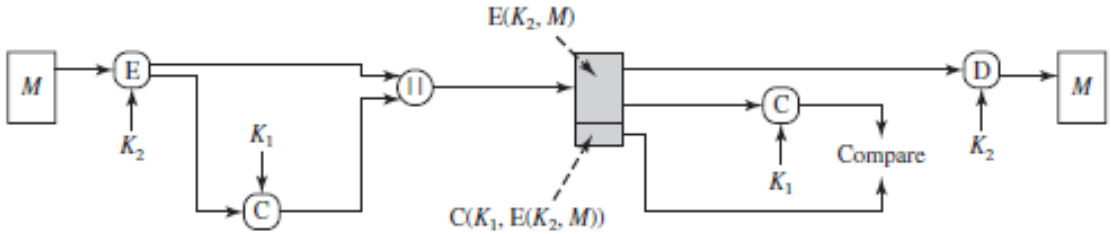
Egy MAC-függvényt háromféleképpen is alkalmazhatunk:



(a) Message authentication



(b) Message authentication and confidentiality; authentication tied to plaintext



(c) Message authentication and confidentiality; authentication tied to ciphertext

Képek forrása: W. Stallings. Cryptography and Network Security. Principles and Practice. Prentice Hall. 2010

Üzenet-hitelesítő kódok, használat

- ha az első ábra szerinti járunk el, akkor nem végzünk rejtjelezést az ebredeti üzeneten, az eljárás azonban így is biztosítja az üzenet hitelességét
- ha a második és harmadik ábrák szerinti járunk el, akkor a MAC-érték meghatározása mellett rejtjelezést is végzünk, ahol a rejtjelezéshez egy K_2 kulcsot, míg a MAC-érték meghatározásához egy $K_1 \neq K_2$ kulcsot kell használni.
- a második két ábra szerinti használat biztosítja az üzenet hitelességét, integritását is
- a gyakorlatban a második ábra szerint szoktak eljárni, azaz a MAC-értéket hozzáfűzik a nyílt-szöveghez és ezen végzik a titkosítást
- a harmadik ábra szerinti először a K_2 kulccsal rejtjelezik az üzenetet, majd erre határozzák meg a MAC-értéket

Üzenet-hitelesítő kódok, a HMAC szerkesztés

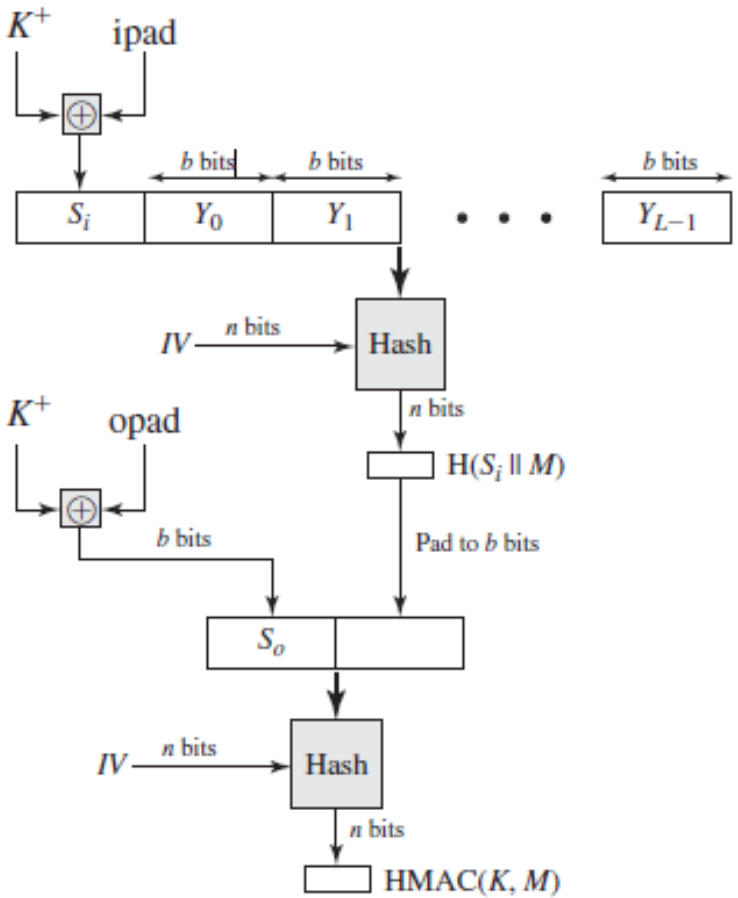
HMAC tervezési szempontok:

- az alkalmazott hash függvény módosítás nélkül legyen használható
- az alkalmazott hash függvény legyen lecserélhető
- őrizze meg az eredeti hash függvény hatékonyságát
- egyszerű legyen a kulcsok használata
- a hash függvény biztonsága alapján következtetni lehessen a MAC-függvény biztonságára
- a következő képlettel adható meg:

$$HMAC_K(M) = H[\ (K^+ \oplus opad) \ ||\ H[(K^+ \oplus ipad) \ ||\ M]\ ]$$

- a HMAC egy fontos tulajdonsága, hogy a külső hash bemenete nem ismert

HMAC



- H - az alkalmazott hash függvény,
- M - a paddingolt üzenet, amelyet az Y_i blokokkra osztottunk, a padding-értékeket az alkalmazott hash függvény határozza meg
- K - a titkos kulcs
- ipad = 0x36, opad = 0x5c, $b/8$-szor ismételve őket
- K^+ - a K kulcs paddingolt értéke: balról nullásokkal egészítjük ki, amíg a kívánt hosszúság b lesz
- L - a blokkok száma
- b - egy blokk bitmérete
- n - a hash függvény kimenetének bitmérete

Üzenet-hitelesítő kódok, a CMAC szerkesztése

- blokk-titkosító segítségével: pl. CMAC (Cipher-based MAC) ahol 3DES vagy AES kerül alkalmazásra
- a szerkesztés a CBC blokktitkosító működésén alapszik, ahol a közbenső titkosított blokkok nem kerülnek eltárolásra, az utolsó blokk titkosítása történik másképp, és a MAC-érték ennek a blokknak a titkosított értéke lesz
- ha az üzenet hossza osztható a blokktitkosító blokkméretével, akkor a K titkos kulcs alapján egy K_1 kerül meghatározásra:

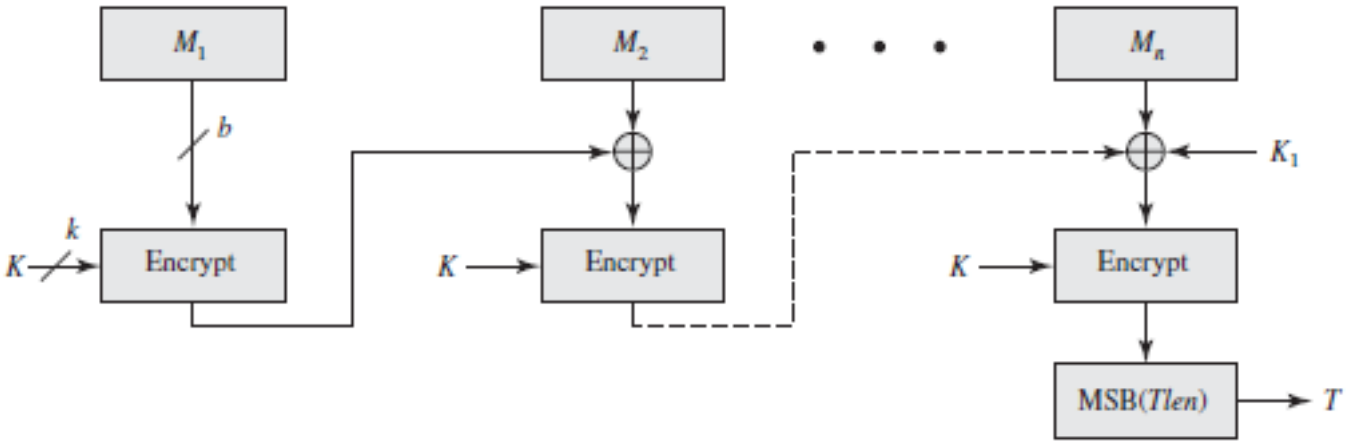
  $$K_1 = E_K(0^b) \bullet y, \text{ ahol } y = \underbrace{0\ldots0}_{b-2 \text{ bit}}\ 10$$

- ha az üzenet hossza nem osztható a blokktitkosító blokkméretével, akkor a K titkos kulcs alapján egy K_2 kerül meghatározásra:

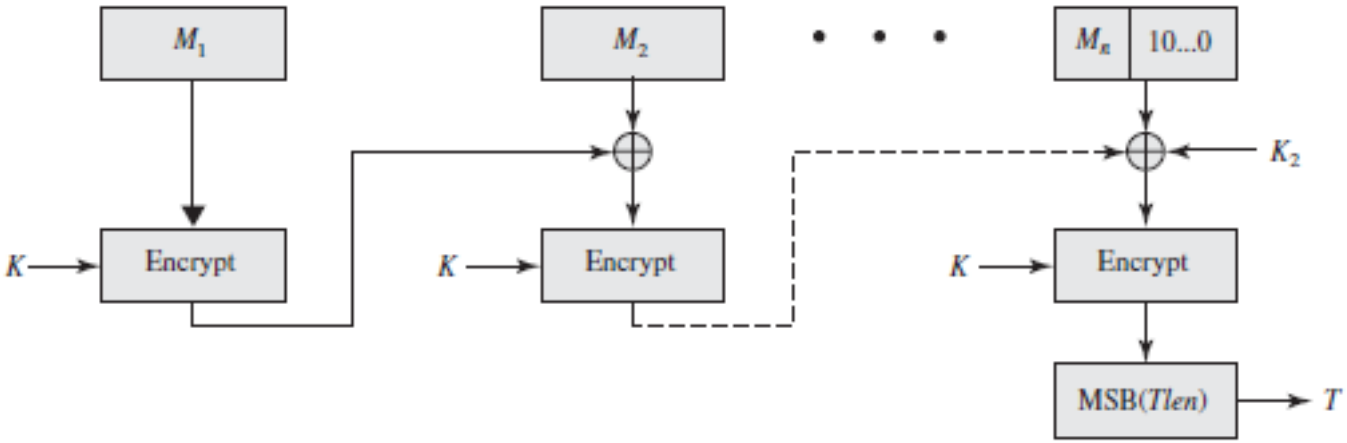
  $$K_2 = E_K(0^b) \bullet y^2 \text{ , ahol } y^2 = \underbrace{0\ldots0}_{b-3 \text{ bit}}\ 100.$$

- a K_1 és K_2 értékeket az utolsó blokk titkosításánál használják
- a $\bullet$ polinomszorzást jelent a $GF(2^b)$ véges testben, megadott irreducibilis polinomok szerint, pl. b = 128 esetében: $x^{128} + x^7 + x^2 + x + 1$ lesz az irreducibilis polinom

CMAC



(a) Message length is integer multiple of block size



(b) Message length is not integer multiple of block size

Hitelesített titkosítás (authenticated encryption)

- Az ESP IP security protokoll (Encapsulating Security Payload): az IP-csomagokat titkosítja és utána számítja ki a MAC-et,
- a TLS (Transport Layer Security) 1.2 protokoll: ha CBC blokk titkosítást alkalmaz, akkor először kiszámítja a MAC-et, majd titkosítja a nyílt szöveget és a MAC-et,
- a titkosítás, majd hitelesítés típusú szerkesztéskor két különálló algoritmusra és két független kulcsra van szükség,
- a TLS 1.3 protokoll csak hitelesített titkosítást támogat, például a GCM módot, amely *egyszerre* végzi a hitelesítést és titkosítást,
- GCM: Galois Counter Mode
  - a CTR egy kiterjesztése, nagy népszerűségnek örvend
  - a GCM mód egyszerre titkosítja a nyílt szöveget és számol hitelesítési tag-et
  - a tag nem csak a nyílt szöveget hitelesíti, hanem további adatokat (AAD -additional authenticated data) is hitelesít
  - GMAC üzenet-hitelesítési kód, amely a GCM egy változata, csak hitelesítési tag-et határoz meg

A GCM (Galois Counter Mode) blokktitkosító mód

- 128 bites blokk titkosítók esetében van definiálva, ezek bármelyikénél lehet használni,
- a CTR mód továbbfejlesztett változata,
- minden titkosításhoz más random IV (nonce) értéket kell használni,
- a hitelesítési tag meghatározása XOR műveletet és polinom-szorzást jelent a $GF(2^b)$ véges testben, pl. b = 128 blokkméret esetében az irreducibilis polinom:

$$x^{128} + x^7 + x^2 + x + 1$$

- a CMAC-hoz képest a bináris szekvenciát little-endian sorrend szerint kezeli: a 128 bites $(b_0 b_1 \dots b_{127})$ blokk a $b_0 + b_1 x + \dots + b_{127} x^{127}$ polinomnak felel meg, és a szorzás művelete is eszerint történik,
- minden titkosításhoz más 96 bites IV értéket kell meghatározni.

A GCM (Galois Counter Mode) blokktitkosító mód

- a 128 bites számláló kezdeti értéke: $ctr = IV||0^{31}||1$, ahol az IV 96 bit.
- az m nyílt szöveget 128 bites blokkokra bontja, ahol az utolsó blokk lehet kisebb is mint 128,
- $m = m_1||m_2||\ldots||m_N$,

$$c_i = E_{key}(ctr + i) \oplus m_i, \text{ minden } i = 1, 2, \ldots N$$
$$c = IV||c_1||c_2||\ldots||c_N$$

legyen $H = E_{key}(0^{128})$ egy 128 bites hash függvény, és $A = AAD$ egy 128 bites *additional authenticated data*:

$$X_1 = A \bullet H$$
$$X_i = (X_{i-1} \oplus c_{i-1}) \bullet H, \text{ minden } i = 2, 3, \ldots N + 1$$
$$X_{N+2} = (X_{N+1} \oplus (len(A)||len(c))) \bullet H,$$
$$t = X_{N+2} \oplus E_{key}(ctr)$$

- a kimenet: (c, t, AAD), ahol t a hitelesítő tag,
- a $\bullet$ művelet szorzást jelent $\pmod{x^{128} + x^7 + x^2 + x + 1}$ szerint a $GF(2^{128})$ véges testben,
- $len(A)$, $len(c)$ 64 biten vannak tárolva.

AES, GCM, str

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

keyS = AES.key_size
print('key sizes: ', keyS)
key = get_random_bytes(keyS[0])

plaintext = 'One of the most widely used security services is Transport
     Layer Security(TSL); the current version is Version 1.3'
nonce = get_random_bytes(32)
print('nonce: ', nonce.hex())
tag, cryptedtext = encryptAES_GCM_str(plaintext, key, nonce)
print('tag: ', tag.hex())
print(cryptedtext.hex())
decryptedtext = decryptAES_GCM_str(cryptedtext, key, tag, nonce)
print(decryptedtext)
```

AES, GCM, str

```
def encryptAES_GCM_str(plaintext, key, nonce):
    header = b'somthing for header'
    cipher = AES.new(key, AES.MODE_GCM, nonce)
    cipher.update(header)
    plaintext = plaintext.encode()
    ciphertext, tag = cipher.encrypt_and_digest(plaintext)
    return tag, ciphertext

def decryptAES_GCM_str(ciphertext, key, tag, nonce):
    header = b'somthing for header'
    cipher = AES.new(key, AES.MODE_GCM, nonce)
    cipher.update(header)
    try:
        decryptedtext = cipher.decrypt_and_verify(ciphertext, tag)
        return decryptedtext
    except:
        print('Failed!!')
```

AES, GCM, file

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

BS = AES.block_size
CS = 2048 * BS

plaintextFile = ...
encryptedFile = ...
decryptedFile = ...
keyS = AES.key_size
print('key sizes: ', keyS)
key = get_random_bytes(keyS[0])
tag, nonce = encryptAES_GCM(plaintextFile, encryptedFile, key)
decryptAES_GCM(encryptedFile, decryptedFile, key, tag, nonce)
```

AES, GCM, file

```
def encryptAES_GCM(inFile, outFile, key):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    header = b'somthing for header'
    cipher = AES.new(key, AES.MODE_GCM)
    cipher.update(header)
    nonce = cipher.nonce # ha nem kap parameterkent akkor general
    with open(inFile, 'rb') as inF, open(outFile, 'wb') as outF:
        for i in range(nrBlock + 1):
            tBlock = inF.read(CS)
            tBlock = cipher.encrypt(tBlock)
            outF.write(tBlock)
    tag = cipher.digest()
    return tag, nonce
```

AES, GCM, file

```
def decryptAES_GCM(inFile, outFile, key, tag, nonce):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    header = b'somthing for header'
    cipher = AES.new(key, AES.MODE_GCM, nonce)
    cipher.update(header)
    with open(inFile, 'rb') as inF, open(outFile,'wb') as outF:
        for i in range(nrBlock + 1):
            tBlock = inF.read(CS)
            tBlock = cipher.decrypt(tBlock)
            outF.write(tBlock)
    try:
        cipher.verify(tag)
    except:
        print('Failed!!')
```

A publikus-kulcsú rendszerek, alapfogalmak

- elméleti alapjait 1976-től kezdve dolgozták/dolgozzák ki:
  - 1976 Diffie-Hellman cikk
  - 1977, Rivest-Shamir-Adleman cikk
  - 1978, Rabin cikk
- nem helyettesíti a szimmetrikus kriptográfiát, de kis méretű (pár kilobájt) információ esetén alkalmas bizalmas információcserére is
- a rendszerek biztonsága matematikai problémákon, feltételezéseken alapszik
- az alapműveletek nem a helyettesítés és permutáció, hanem egész számokkal vagy számpárokkal végzett algebrai műveletek
- a rendszerekben megkülönböztetnek publikus kulcsot és privát kulcsot, ahol a privát kulcsok szigorúan titkosak, a publikus kulcsokat pedig mindenki ismeri

A publikus-kulcsú rendszerek, alapfogalmak

felosztása:

- kulcscserék (key exchange, KE): lehetővé teszik, hogy a kommunikáló felek megeggyezenek egy szimmetrikus titkosító kulcsának az értékében:
  - pl. a küldő és fogadó fél is generál egy-egy random értéket, amelyeket felhasználva megtudnak állapodni egy AES-256 kulcsban
- publikus kulcsú titkosítók (public key encryption, PKC): lehetővé teszi, hogy a kommunikáló felek megosszák egy szimmetrikus titkosító kulcsát
  - pl. a küldő fél generál egy AES-256 kulcsot, amelyet PKC-t alkalmazva elküld a fogadó félnek
- digitális aláírások (digital signature, DS):
  - pl. kulcscserénél, publikus kulcsú titkosítóknál: publikus kulcsok hitelesítése,
  - pl. kriptovaluták esetében a tranzakciók hitelesítése