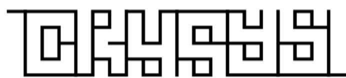
Kriptográfiai kódolás – 2. rész

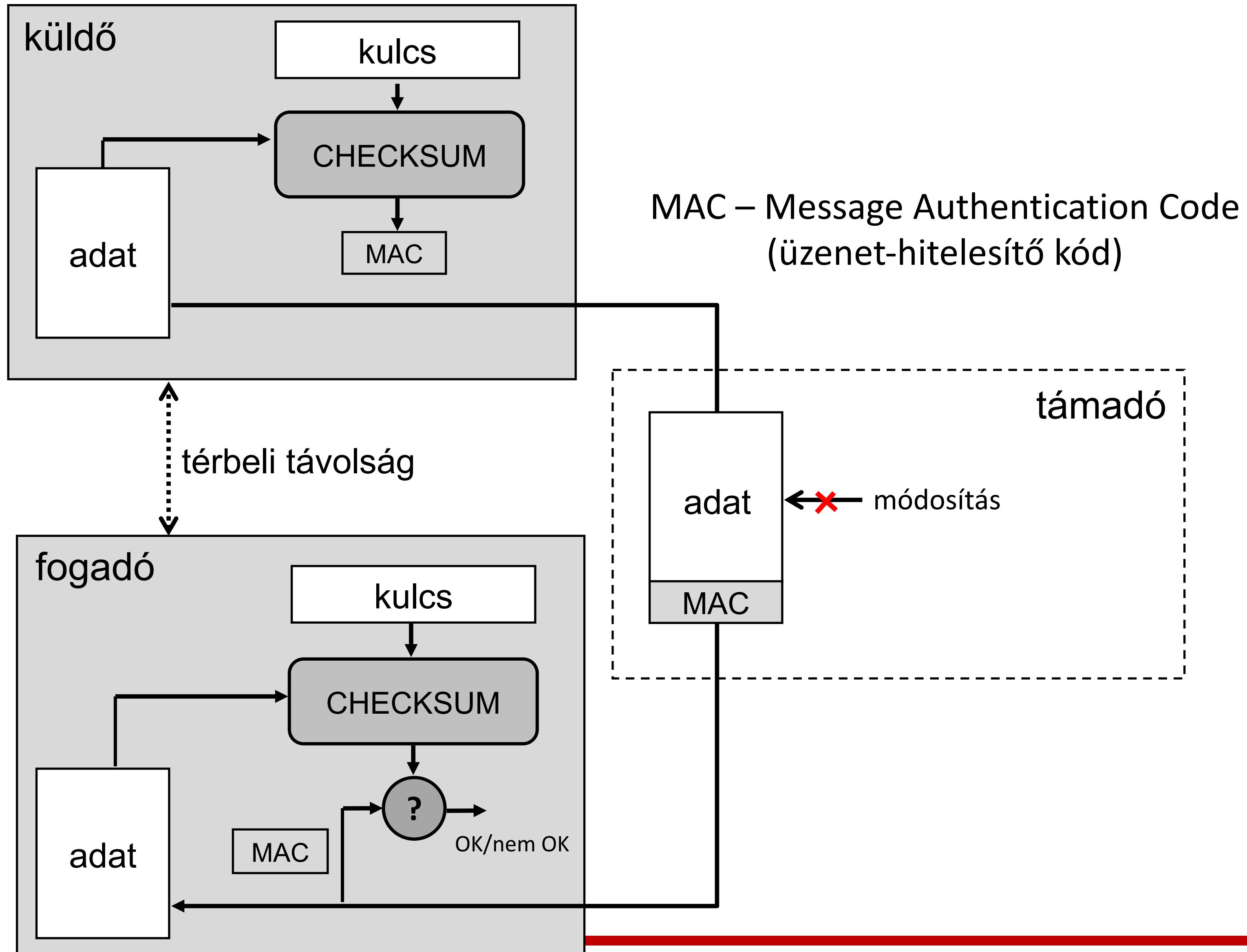
VIHIBB01 – Kódolás és IT biztonság, 2020

Dr. Buttyán Levente

CrySyS Lab, BME
buttyan@crysys.hu



Az üzenet-hitelesítés modellje



Hash függvények

Kriptográfiai hash függvények

- Egy hash függvény tetszőlegesen hosszú bemenetből egy adott fix hosszúságú (n bites) kimenetet állít elő

- Jelölés és terminológia:
  - x – üzenet (bemenet)
  - y = H(x) – hash érték, üzenet lenyomat

- Tipikus alkalmazások:
  - jelszó hash-elés
  - adat hash-elése digitalis aláírás előtt

- Példák:
  - (MD5, SHA-1), SHA-2 (függvény család), SHA-3 (függvény család)

Hash függvények tulajdonságai

- Könnyű (általában gyors) számíthatóság
  – Adott x bemenetre, a H(x) hash érték könnyen számolható
- **Gyenge ütközés ellenállóság** (második őskép ellenállóság)
  – Adott x bemenethez, nehéz olyan x-től különböző x’ bemenetet találni, melyre H(x’) = H(x)
- **Erős ütközés ellenállóság** (ütközés ellenállóság)
  – Nehéz két különböző x és x’ bemenetet találni, melyre H(x) = H(x’)
- **Őskép ellenállóság** (egyirányúság)
  – Adott y hash értékhez (melynek egyetlen őképe sem ismert), nehéz olyan x bemenetet találni, melyre H(x) = y

A hash függvény mint véletlen függvény

- Az ütközés ellenálló hash függvények véletlen függvényként modellezhetők (a blokk rejtjelezőkhöz hasonlóan)
  - Egymáshoz hasonló bemenet hash értéke teljesen különböző
  - Pontosabban, 1 bit változás a bemenetben a kimenet bitjeinek kb. felét változtatja meg (**lavina hatás** - avalanche effect)

- Illusztráció:
  - SHA1("The quick brown fox jumps over the lazy dog")
    --» `2fd4e1c67a2d28fced849ee1bb76e7391b93eb12` (20 byte = 160 bit)
  - SHA1("The quick brown fox jumps over the lazy cog")
    --» `de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3` (20 byte = 160 bit)
  - a második hash érték 81 bitben különbözik az elsőtől

PyCryptodome: Hash függvények

- Crypto.Hash modul
- Támogatott algoritmusok:
  - MD2, MD5, RIPEMD, SHA-1, SHA-2 (család), SHA-3 (család), Blake2

- Használati példa: SHA256 (a SHA2 család egy tagja)
  - Egy új hash objektum a SHA256.new() hívással hozható létre
  - A hash objektum update() függvényével tetszőleges hosszú bemenet feldolgozható
  - Magát a hash értéket a digest() függvény állítja elő
  - Vagy használhatjuk a hexdigest() függvényt is, ami hex formátumban adja vissza a hash értéket

PyCryptodome: Hashelés SHA256-tal

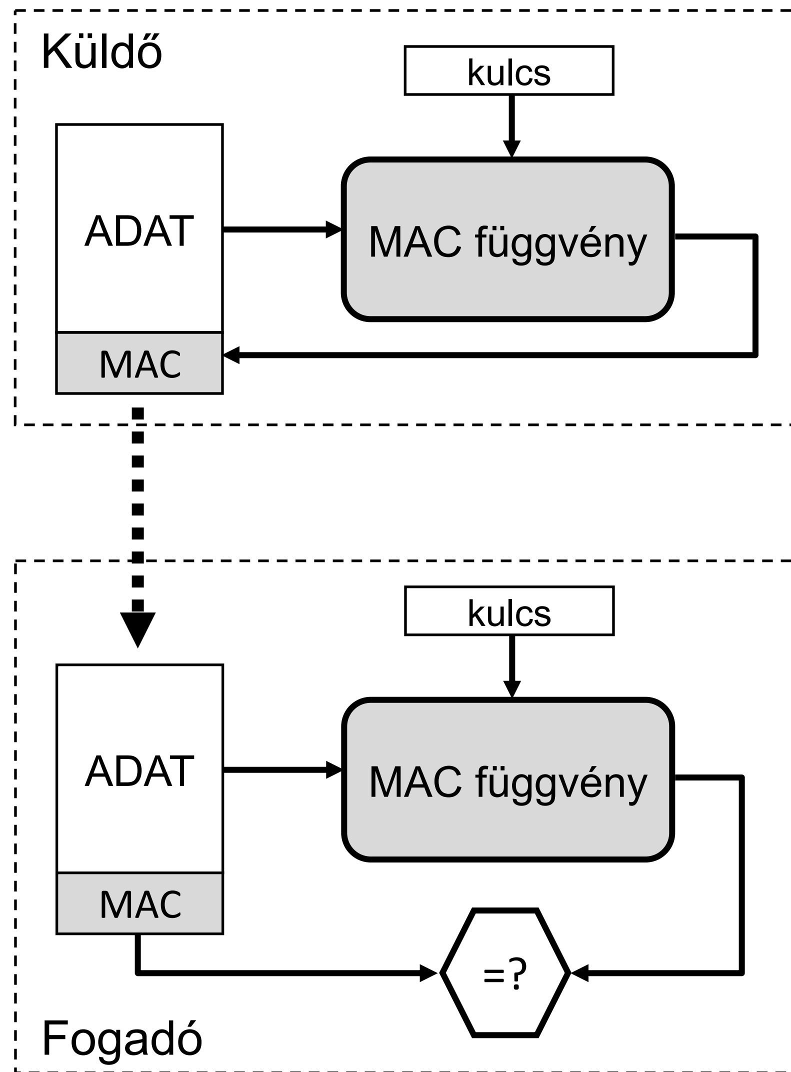
sha256-test.py

```
from Crypto.Hash import SHA256

h = SHA256.new()
text = b'Hello World! SHA256 is a hash function with an output size of 256 bits. '
h.update(text)
text = b'It is member of the SHA-2 family of hash functions.'
h.update(text)

print(h.hexdigest())   # equivalent to print(h.digest().hex())
print(h.digest())
```

MAC függvények

MAC függvények

- Tetszőlegesen hosszú bemenetből és egy fix (k bit) hosszúságú kulcsból, egy fix (n bit) hosszúságú kimenetet állítanak elő
- Úgy lehet rájuk gondolni, mint két bemenettel rendelkező hash függvényekre
- Szolgáltatások:
  - **Üzenet-hitelesítés** és **–integritás-védelem:** a MAC sikeres ellenőrzése után, a fogadó tudja, hogy az üzenetet olyan valaki küldte, aki ismeri a kulcsot (azaz a küldő), és az nem változott meg az átvitel során
- Példák:
  - HMAC, CBC-MAC



MAC függvények biztonsága

- Támadói modellek
  - A támadás célja lehet:
    - » Érvényes MAC érték előállítása egy vagy több üzenethez (MAC hamisítás)
    - » A MAC kulcs megfejtése (a MAC függvény feltörése)
  - A támadó számára rendelkezésére álló információk jellege:
    - » Ismert üzenet-MAC érték párok (átvitel során megfigyelhetők)
    - » (Adatptívan) választott üzenetekhez tartozó MAC értékek (orákulum támadás)

- A MAC függvények elvárt biztonsági tulajdonságai
  - key non-recovery
    - » A K titkos MAC kulcs megfejtése nehéz feladat, még akkor is ha a támadó megfigyel vagy egy orákulumtól megszerez egy vagy több (m_i, M_i) üzenet-MAC párt
  - computation resistance
    - » Tetszőleg m üzenetre, nehéz érvényes M MAC értéket előállítani, még akkor is ha a támadó ismer érvényes (m_i, M_i) üzenet-MAC párokat (és $m \neq m_i$)
    - » A „computation resistance” tulajdonságból következik a „key non-recovery” tulajdonság, de ez fordítva nem feltétlenül igaz

MAC függvények elleni nyers erő támadások

- Egy adott üzenethez véletlen választott MAC érték 2^{-n} valószínűséggel helyes --» a kimerítő próbálkozás átlagos komplexitása 2^{n-1}

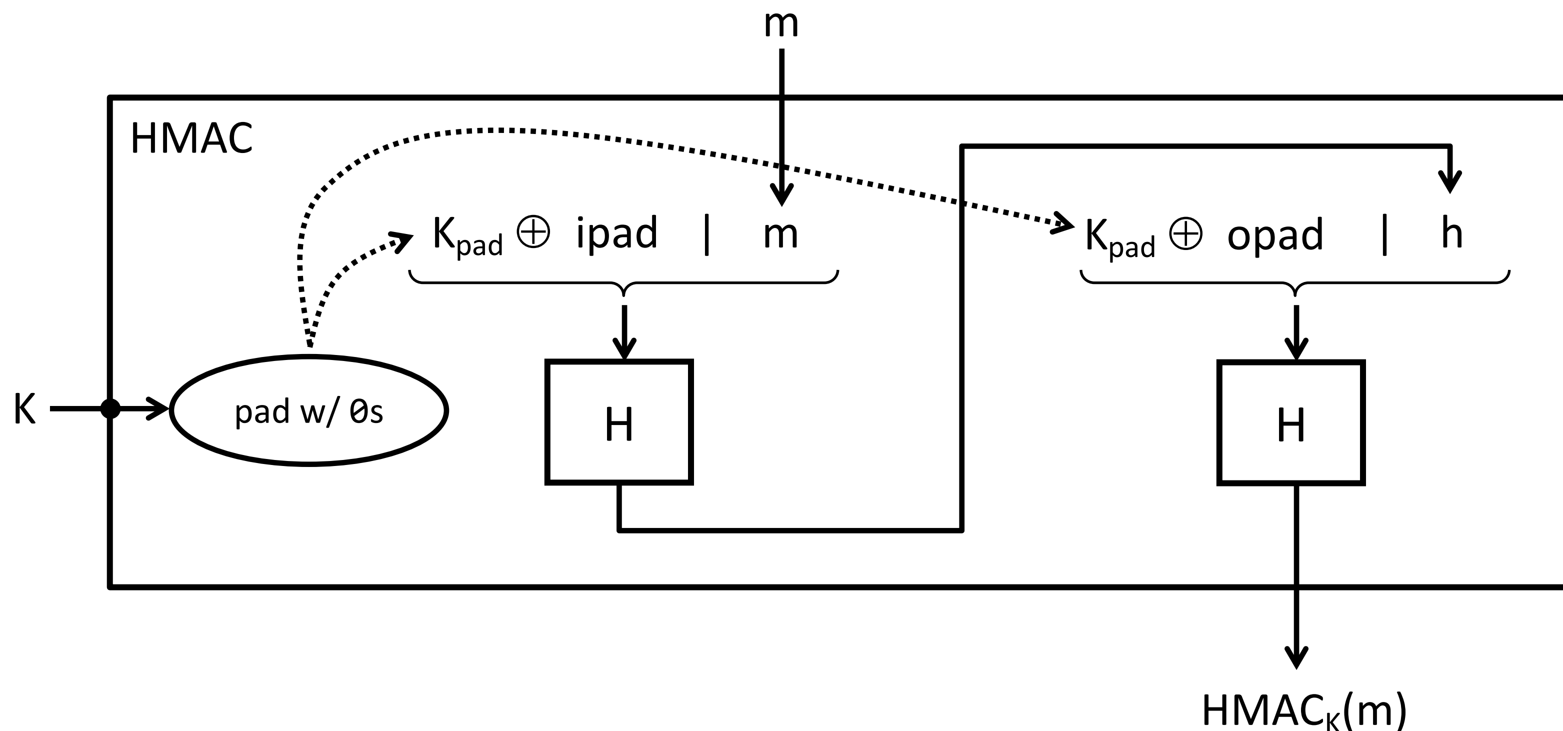
- A kimerítő kulcskeresés komplexitása 2^{k-1}
  - Tegyük fel, hogy van pár megfigyelt (m_i, M_i) üzenet-MAC párunk
  - Minden lehetséges kulccsal kiszámoljuk az m_i MAC értékét és összehasonlítjuk M_i-vel
  - Ha nincs egyezés, akkor új kulccsal próbálkozunk
  - Egyezés esetén a kulcsot teszteljük az összes rendelkezésre álló (m_i, M_i) páron
  - Ha a kulcs mindegyik párra működik, akkor ez lesz a jó kulcs

--» min(2^k, 2^n)-nak elegendően nagynak kell lennie ahhoz, hogy a gyakorlatban egyik nyers erő (brute force) támadás se működjön

HMAC

$HMAC_K(m) = H((K_{pad} \oplus opad) \mid H((K_{pad} \oplus ipad) \mid m))$

- H egy hash függvény, mely
  - » a bemenetét b bites blokkokban dolgozza fel
  - » kimenete n bites
- K_{pad} a K kulcs nullákkal kitöltött változata, melynek hossza b bites
- ipad és opad n bit hosszú konstans értékek



more info: https://tools.ietf.org/html/rfc2104

PyCryptodome: MAC függvények

- Crypto.Hash modul
- Támogatott algoritmusok:
  - HMAC (hash függvényből konstruált MAC függvény)
  - CMAC (blokkrejtjelezőből konstruált MAC függvény)

- Használati példa: HMAC (a SHA256 hash függvénnyel)
  - Új HMAC objektum a HMAC.new() függvénnyel hozható létre, melynek bemenetként megadható a MAC kulcs és a használni kívánt hash függvény
  - A HMAC objektum update() függvényével tetszőleges hosszú bemenet feldolgozható
  - Magát a MAC értéket a digest() függvény állítja elő
  - Vagy használhatjuk a hexdigest() függvényt is, ami hex formátumban adja vissza a MAC értéket

PyCryptodome: MAC számítás HMAC-kel

hmac-test.py

```
from Crypto.Hash import HMAC, SHA256

mackey = b'yoursecretMACkey'
mac = HMAC.new(mackey, digestmod=SHA256)

msg = b'Hello World! HMAC is a keyed hash function. '
mac.update(msg)
msg = b'This example uses the SHA256 hash function. '
mac.update(msg)
msg = b'However, you can change this and use another hash function too.'
mac.update(msg)

print(mac.hexdigest())   # equivalent to print(mac.digest().hex())
print(mac.digest())
```

Digitális aláírás

Digitális aláírás sémák

- Aszimmetrikus kulcsú primitív
- Hasonló a MAC függvényhez, de a digitális aláírás
  - a fogadó által hamisíthatatlan
  - bármely harmadik fél által ellenőrizhető
- Szolgáltatások:
  - **Üzenet-hitelesítés** és **–integritás-védelem:** az aláírás sikeres ellenőrzése után, a fogadó tudja, hogy az üzenetet a küldő küldte, és az nem változott meg az átvitel során
  - **Eredet letagadhatatlanság:** a fogadó bárkinek bizonyítani tudja, hogy az üzenetet a küldő küldte (ezért azt a küldő nem tudja letagadni)
- Példák: RSA, DSA, ECDSA

Függvények és terminológia

– Kulcspár generáló függvény $G() = (K^+, K^-)$

K^+ – publikus kulcs

K^- – privát kulcs

– Aláírás generáló függvény $S(K^-, m) = s$

m – üzenet

s – aláírás

– Aláírás ellenőrző függvény $V(K^+, m, s)$ = OK vagy nem OK

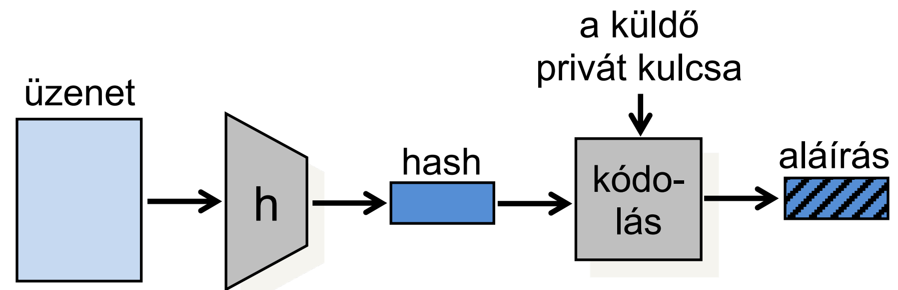
A digitális aláírás biztonsága

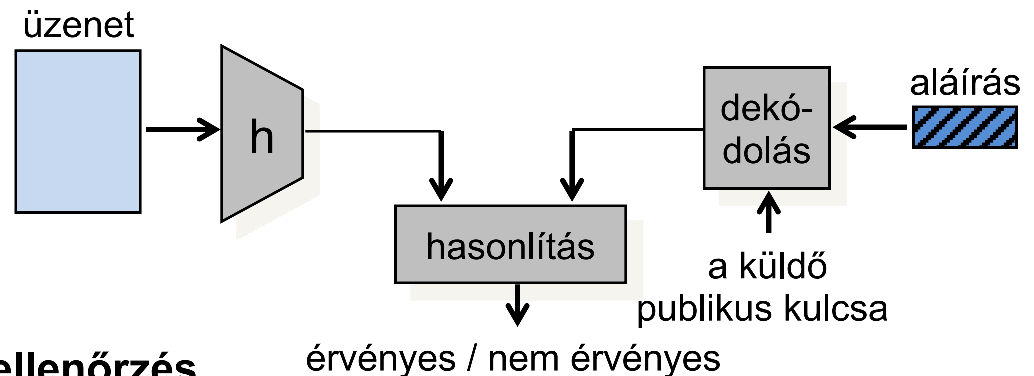
- A publikus kulcsú rejtjelezőkhöz hasonlóan, a biztonság nehéznek vélt matematikai problémákra van visszavezetve

- Támadói modellek
  - A támadó rendelkezésére álló információk jellege:
    - » Csak a publikus aláírás ellenőrző kulcs áll rendelkezésre (key-only attack)
    - » Rendelkezésre állnak üzenet-aláírás párok (known-message attack)
    - » A támadó (adaptívan) választott üzenetekhez megkaphatja az érvényes aláírást egy orákulumtól (chosen-message attack)
  - A támadás lehetséges céljai:
    - » Aláírás hamisítása
      - Érvényes aláírás előállítása olyan üzenetre, amihez eddig nem volt megfigyelt vagy orákulumtól szerzett aláírás
    - » A privát aláíró kulcs megfejtése

Hash-and-sign modell

- Nagy inputon végzett publikus/privát kulcsú műveletek lassúak
- A hatékonyság növelhető, ha nem magát az üzenet írjuk alá, hanem annak a hash értékét
- Ekkor nagyon fontos, hogy a hash függvény ütközés ellenálló legyen (miért?)





PyCryptodome: Digitális aláírás használata

- Crypto.Signature modul
- Támogatott algoritmusok:
  - RSA (w/ PSS – Probabilistic Signature Scheme), DSA, ECDSA
- Használati példa:
  - Aláírás generálás
    - » Privát kulcs importálása
    - » Aláíró objektum létrehozása a new() függvénnyel, melynek átadásra kerül a privát kulcs
    - » Hash objektum létrehozása és az aláírandó üzenet hash-elése
    - » Az aláíró objektum sign() függvényének meghívása a hash objektummal mint bementtel (ez belső állapotában tartalmazza az üzenet hash értékét)
  - Aláírás ellenőrzés
    - » Publikus kulcs importálása
    - » Aláíró objektum létrehozása a new() függvénnyel, melynek átadásra kerül a publikus kulcs
    - » Hash objektum létrehozása és az üzenet hash-elése
    - » Az aláíró objektum verify() függvényének meghívása a hash objektummal mint bementtel (ez belső állapotában tartalmazza az üzenet hash értékét)

PyCryptodome: RSA-PSS aláírás generálás

rsa-sign-test.py

```
from Crypto.Signature import PKCS1_PSS
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

ifile = open('testinput.txt', 'rb')
msg_to_be_signed = ifile.read()
ifile.close()

h = SHA256.new()
h.update(msg_to_be_signed)
# Don't call h.digest() here!!!
# The hash object h will be passed to the signing function,
# and it will complete the hash calculation by calling h.digest()

kfile = open('rsa-test-keypair.pem', 'r')
keypairstr = kfile.read()
kfile.close()

keypair = RSA.import_key(keypairstr, passphrase='x#4K')
signer = PKCS1_PSS.new(keypair)

signature = signer.sign(h)

#print(signature)
print(signature.hex())
```

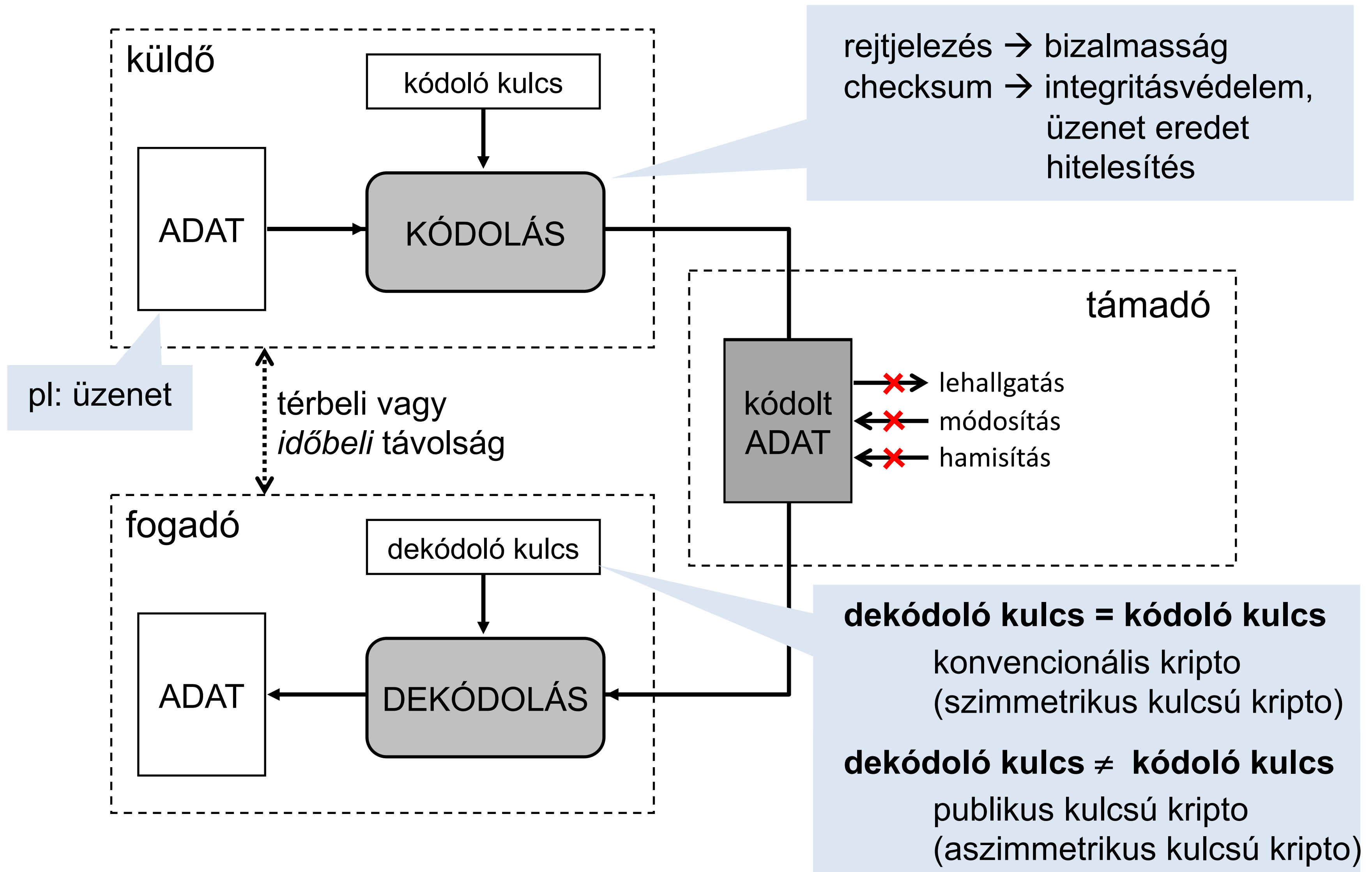
Ez csak egy (rossz) példa!
Soha nem szabad jelszót
alkalmazásba belekódolni!

PyCryptodome: RSA-PSS aláírás ellenőrzés

rsa-sver-test.py

```
from Crypto.Signature import PKCS1_PSS
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

ifile = open('testinput.txt', 'rb')
msg_signed = ifile.read()
ifile.close()

h = SHA256.new()
h.update(msg_signed)
# Don't call h.digest() here!!!
# The hash object h will be passed to the signing function,
# and it will complete the hash calculation by calling h.digest()

kfile = open('rsa-test-pubkey.pem', 'r')
pubkeystr = kfile.read()
kfile.close()

pubkey = RSA.import_key(pubkeystr)
verifier = PKCS1_PSS.new(pubkey)

sfile = open('rsa-sig.out', 'rb')
signature = sfile.read()
sfile.close()

if verifier.verify(h, signature):
        print('Success.')
else:
        print('Failure.')
```

Összefoglalás



Gyakorlatban előforduló problémák

- Kulcs menedzsment problémák
  - pl: gyenge véletlenszám generátor használata kulcs generálásánál
  - pl: jelszó alapú kulcsgenerálás nem megfelelő módon
- Protokoll gyengeségek
  - pl: kripto algoritmusok nem megfelelő módon történő használata
- Implementációs problémák
  - programozási hibák
  - „side channel” támadások (pl: műveletek időigénye, áramfelvétele információt szivárogtat ki az éppen használt kulcsról)
- Emberi butaság
  - pl: saját tervezésű „kripto” algoritmus használata

Kulcsok generálása, tárolása

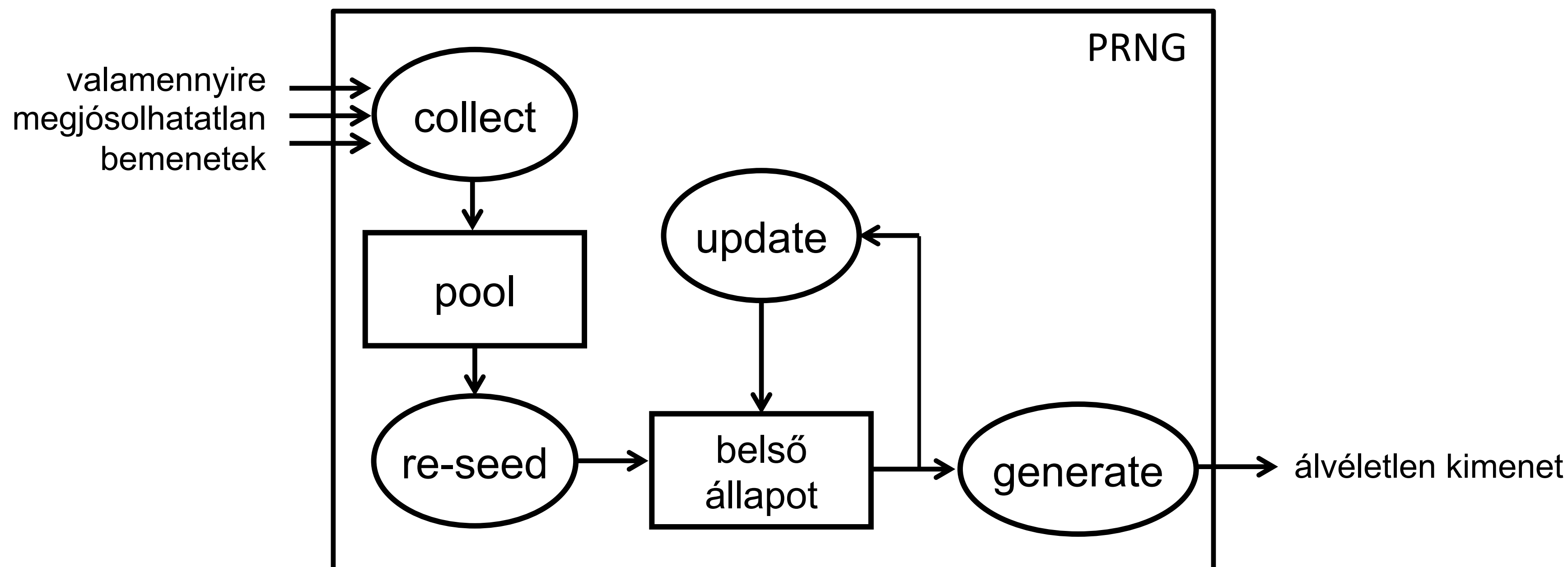
Véletlenszámok

- Mind a szimmetrikus, mind az aszimmetrikus kulcsok generálásához szükség van a támadó számára megjósolhatatlan véletlen számokra (bitekre)
- Sok más kriptográfiai paraméterrel (pl. IV) szemben is követelmény az előre megjósolhatatlanság
- A programnyelvek alapértelmezett véletlenszám generátorai (pl. C rand() függvény) nem alkalmasak kriptográfiai célokra, mert előre megjósolható kimenetet produkálnak

Valódi vs. álvéletlen számok

- Egy valódi véletlenszám generátor kimenete megjósolhatatlan, még akkor is, ha korábbi kimeneteit ismerjük
  - ha pl. a kimenet n bites, akkor akármennyi kimenetet figyeltünk is meg eddig, a következő kimenetet nem tudjuk 2^{-n}-nél nagyobb valószínűséggel helyesen megjósolni
- Sajnos valódi véletlenszámok sokszor nem állnak rendelkezésre megfelelő mennyiségben vagy sebességgel
  - pl. ha egy RSA kulcspárt akarunk generálni, akkor több ezer véletlen bitre van szükségünk, és nem szeretnénk percekig várni ezek generálására
- Az **álvéletlen generátorok** (pseudo-random number generator - PRNG) nem valódi véletlen kimenetet állítanak elő, viszont...
  - gyorsak, nagy mennyiségű kimenetet képesek generálni rövid idő alatt
  - ha megfelelően vannak tervezve és használva, akkor olyan kimenetet produkálnak, ami egy külső megfigyelő számára praktikusan megkülönböztethetetlen egy valódi véletlen generátor kimenetétől

PRNG-k működése



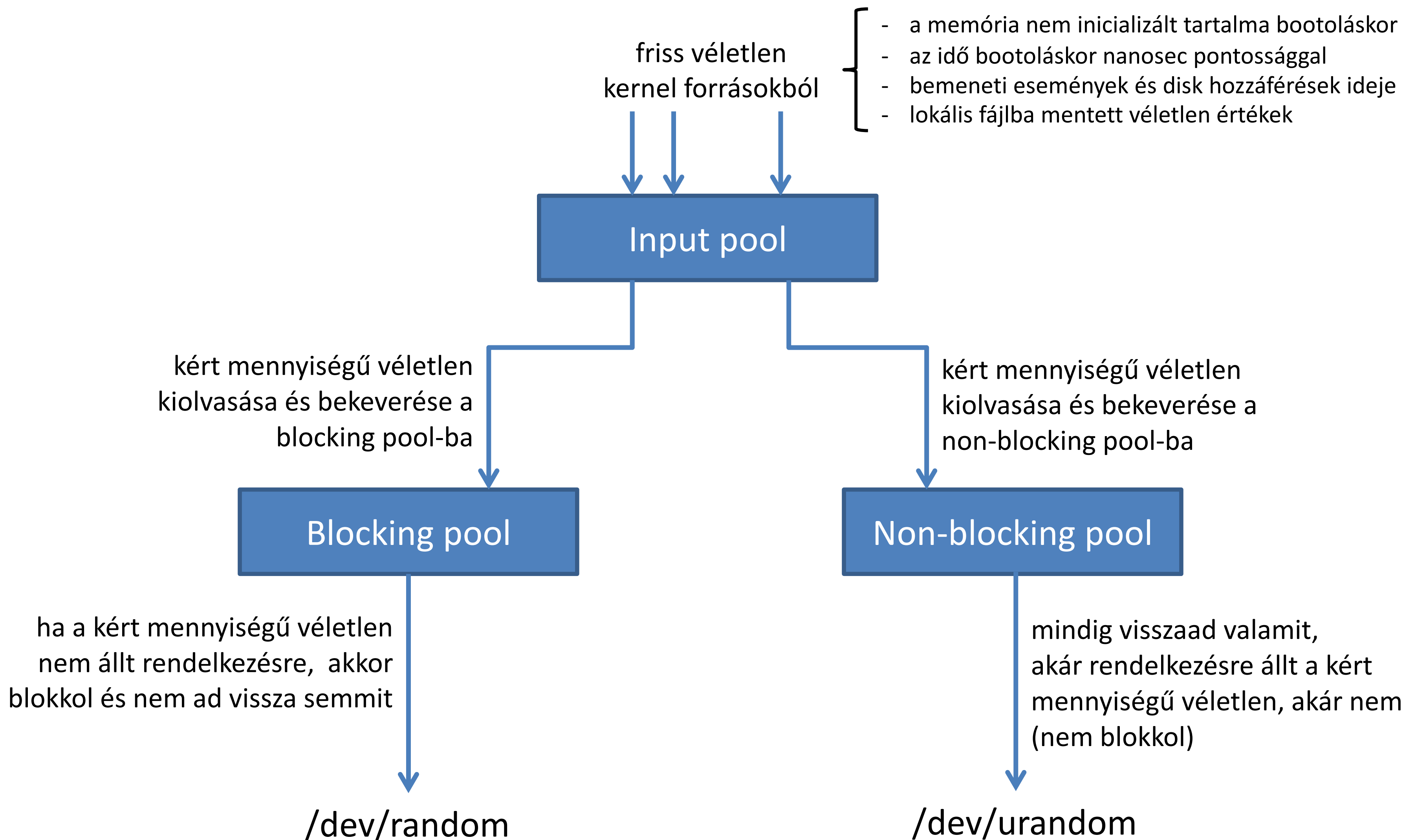
- A belső állapotot...
  - minden kimenet generálás után frissítjük
  - néha (mikor már elég sok valamennyire megjósolhatatlan bemenetet gyűjtöttünk) teljesen újra generáljuk (re-seed)
- A kimenetet a belső állapotból állítjuk elő egy egyirányú függvény segítségével (pl. hash függvény vagy blokkrejtjelező)

Valamennyire megjósolhatatlan bemenet

- Lehetséges források:
  - aktuális idő (óra)
  - billentyű leütések közti idő
  - egér mozgás
  - disk hozzáférési idő
  - kamerakép, mikrofon
  - hardver mellékhatások (pl. ellenállás termikus zaja)

- A bemenet nem teljesen véletlen, a támadó ismerheti vagy befolyásolhatja a bement egy részét
- Ami fontos az az, hogy nem tud mindent megfigyelni vagy befolyásolni, azaz a bemenet valamennyire megjósolhatatlan marad számára

/dev/random és /dev/urandom Linux-on



Kulcsok tárolása

- Az alkalmazásoknak biztonságosan kell tárolniuk a kulcsokat
- Két lehetséges hozzáállás:
  - Bontás ellenálló hardver modul (pl: chip-kártya)
    - » Fizikai védelem és hozzáférések kontrollja
    - » A kulcsnak sosem kell elhagynia a biztonságos modult
    - » Ez azért lehetséges, mert a modul nem csak tárolásra alkalmas, hanem számítási műveletek végzésére is (pl: kripto műveletek tud végezni a kulccsal)
    - » A modul kívülről bemenetként kapja a számításhoz a további adatokat
    - » Minden művelethez engedélyezés (pl: PIN megadása) szükséges
  - Rejtjelezett fájl
    - » A fájlrejtjelező kulcs gyakran jelszóból van előállítva
    - » A felhasználónak minden kulcshasználat előtt be kell írnia ezt a jelszót
    - » Figyelni kell arra, hogy a jelszó potenciálisan lehet gyenge, ami off-line szótáras támadásokhoz vezethet
    - » Speciális módon kell a fájlrejtjelező kulcsot generálni a jelszóból!

Kulcsgenerálás jelszóból

- Két fontos mechanizmus: *stretching* és *salting*
- Stretching
  - a jelszó találgatás költségének növelése azzal, hogy a kulcsot a jelszóból egy sok iterációt igénylő számítással állítjuk elő (pl. többszörös hash-elés vagy rejtjelezés)
  - 10000 iteráció 10000-szeresen lelassítja a kulcs előállítását, de...
    - » a legitim felhasználó számára, aki a jó jelszót írja be, ez szinte észrevehetetlen
    - » a támadónak viszont egy egész szótárral kell próbálkoznia!

Key derivation from passwords

- Salting
  - Salt = véletlen érték, amivel a jelszót módosítjuk a kulcsgenerálás előtt
  - Biztosítja, hogy ugyanabból a jelszóból több különböző kulcs is generálható
    - » Ha a salt mérete elég nagy, akkor a generált kulcsok között elenyésző valószínűséggel lesz azonos
  - Ellehetetleníti az előszámításokra (pre-computation) épülő támadásokat
    - » Ha a támadó előre szeretné kiszámítani és tárolni egy szótár szavaiból előállítható kulcsokat, akkor ezt most minden lehetséges salt értékre meg kell tennie
    - » Ha a salt mérete elég nagy, akkor a szükséges számítási és tárolási kapacitás óriási lehet (pl. 64 bites salt --» 2^{64} számítási és tárolási komplexitás)

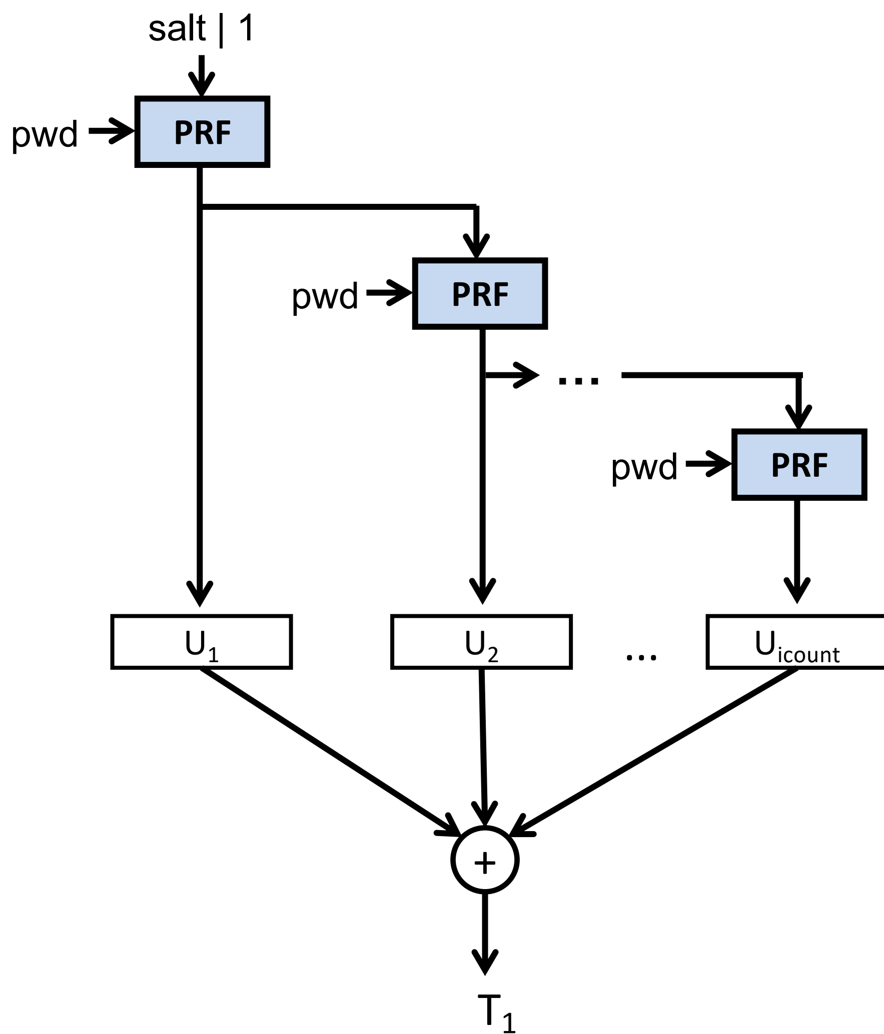
PBKDF2

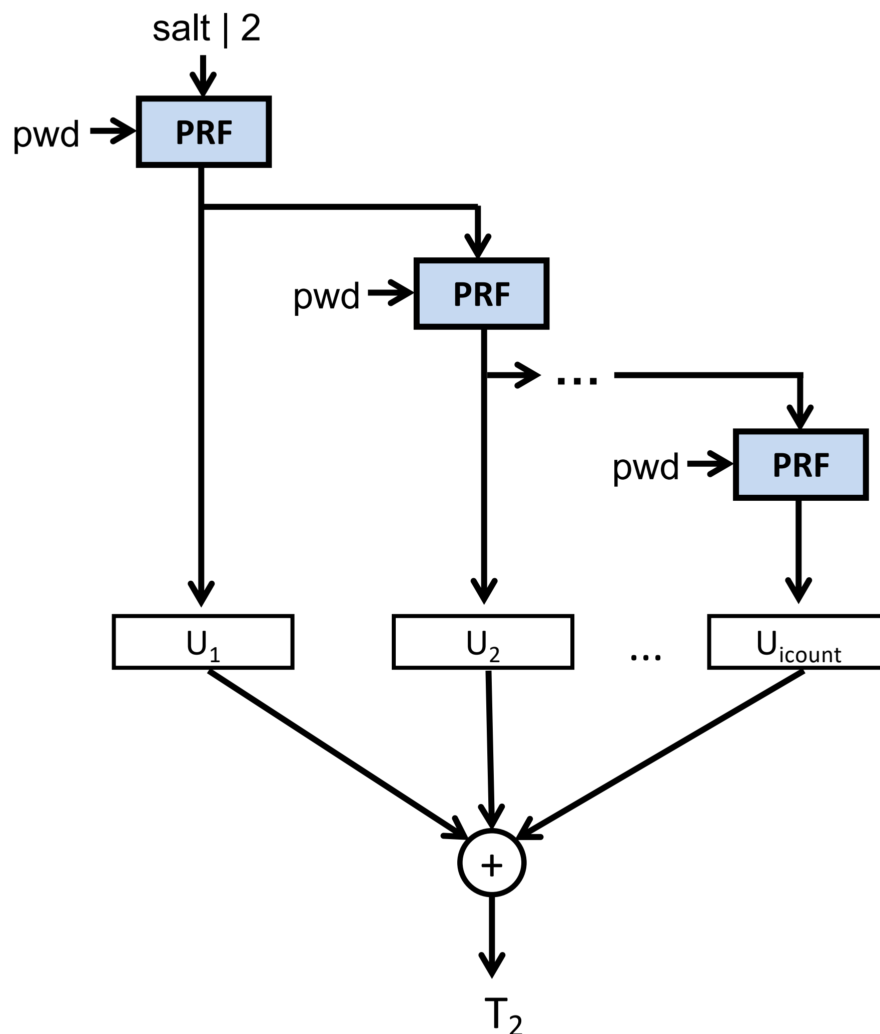
- PBKDF = Password Based Key Derivation Function

- derived key = PBKDF2(PRF, passwd, salt, icount, length)
  - PRF – véletlen függvény két bemenettel (pl: HMAC-SHA256)
  - passwd – jelszó, amiből a kulcsot akarjuk generálni
  - salt – elegendően nagy (pl: 64 bites) véletlen érték
  - icount – iterációk száma (pl: 10000 vagy több)
  - length – a generált kulcs kívánt mérete (pl: 128 vagy 256 bit)

- Számítások:
  - PBKDF2(PRF, passwd, salt, icount, length) = $T_1 \mid T_2 \mid \ldots$
    ahol T_i = F(passwd, salt, icount, i)
  - F(passwd, salt, icount, i) = $U_1 \oplus U_2 \oplus \ldots \oplus U_{icount}$
    ahol U_1 = PRF(passwd, salt|i) és U_k = PRF(passwd, U_{k-1})   (k = 2, 3, ...)

PBKDF2





PyCryptodome: Kulcsgenerálás jelszóból

- Crypto.Protocol.KDF modul
- Támogatott algoritmusok:
  - PBKDF2, scrypt, bcrypt, HKDF
- Használati példa: PBKDF2
  - A jelszó lehet parancssori argumentum vagy be lehet kérni a felhasználótól
  - A salt generáláshoz erős véletlenszám generátort kell használni
  - PBKDF2() függvény meghívása a megfelelő bemeneti értékekkel
  - a függvény kimenete a generált kulcs

PyCryptodome: PBKDF2 használata

kdf-test.py

```
from getpass import getpass
from Crypto.Random import get_random_bytes
from Crypto.Protocol import KDF

passphrase = getpass(prompt='Passphrase: ')
salt = get_random_bytes(16)
aes_key = KDF.PBKDF2(passphrase, salt, count=10000, dkLen=32)   # 32 bytes=256 bits

# use aes_key to initialze an AES cipher object
# use the AES cipher object to encrypt/decrypt data
```

Ellenőrző kérdések

- Mi a kriptográfiai hash függvény?
- Mi a hash függvények 3 fő biztonsági tulajdonsága?
- Hogyan használható egy MAC függvény üzenet-hitelesítésre?
- Milyen támadói modellek léteznek MAC függvények esetén?
- Mik a MAC függvények biztonsággal kapcsolatos tulajdonságai?
- Hogyan működnek a nyers erő támadások a MAC függvények ellen?
- Mi a digitális aláírás? Mik a fő különbségek a digitális aláírás és a MAC függvények között?
- Milyen támadói modellek léteznek digitális aláírás sémák ellen?
- Mi a hash-and-sign modell? Miért használjuk a gyakorlatban? Milyen követelményt támaszt az alkalmazott hash függvénnyel szemben?
- Milyen problémák fordulnak elő a gyakorlatban kriptográfiát használó rendszerekben?

Ellenőrző kérdések

- Mit jelent az, hogy a valódi véletlenszámok megjósolhatatlanok?
- Miben különbözik egy álvéletlen generátor egy valódi véletlenszám generátortól?
- Hogyan épül fel egy PRNG? Honnan kaphat valamennyire megjósolhatatlan bemenetet?
- Hogyan tárolhatunk kulcsokat biztonságosan?
- Mire kell odafigyelni, ha jelszóból generálunk kriptográfiai kulcsot?
- Mi az a salting és mi az a stretching?