Kriptográfia és Információbiztonság
8. előadás

MÁRTON Gyöngyvér

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

2025

Miről volt szó az elmúlt előadáson?

- hash függvények (hash function)
- üzenethitelesítő kódok (message authentication code)
- hitelesített titkosítás (authenticated encryption)

Miről lesz szó?

- publikus kulcsú rendszerek: alapfogalmak
- publikus kulcsú rendszerek: matematika modellek,
- az RSA publikus kulcsú rendszer: RSA-textbook, RSA-OAEP, RSA-PSS

Publikus kulcsú kriptográfia, alapfogalmak

- aszimmetrikus kulcsú kriptográfiának is mondják
- 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 a permutáció, hanem matematikai objektumokkal való algebrai műveletek
- a rendszerekben megkülönböztetnek publikus kulcsot és privát kulcsot, ahol a privát kulcsok szigorúan titkosak, a publikus kulcsokat pedig mindenki ismeri

Publikus kulcsú kriptográfia, alapfogalmak

felosztása:

- kulcscserék (key exchange, KE): lehetővé teszik, hogy a kommunikáló felek egy publikus csatornát használva megeggyezenek egy csak általuk ismert értékben:
  - 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 egy publikus csatornát használva megosszanak egy csak általuk ismert értéket:
  - pl. a küldő fél generál egy AES-256 kulcsot, amelyet publikus kulcsú titkosítást alkalmazva elküld a fogadó félnek
- digitális aláírások (digital signature, DS): lehetővé teszik egy tetszőleges üzenetet hitelesítését, integritását és azt, hogy az aláírást ne lehessen visszavonni (egy későbbi időpontban ne lehessen letagadni az aláírást):
  - pl. kulcscserénél, publikus kulcsú titkosítóknál: publikus kulcsok hitelesítése,
  - pl. kriptovaluták esetében a tranzakciók hitelesítése

Publikus kulcsú kriptográfia

feltételezések:

- számelmélet
  - faktorizáció feltételezés: nincs hatékony algoritmus, amely egy összetett szám prím osztóit meghatározná
  - kvadratikus maradék feltételezés: nincs hatékony algoritmus, amely egy n összetett egész szám esetében megállapítaná egy x számról, hogy az négyzetes maradék vagy sem; egy x szám négyzetes maradék, ha létezik olyan y szám, amelyre fennáll:
$$x \equiv y^2 \pmod{n}$$
  - diszkrét logaritmus feltételezés: nincs hatékony algoritmus, amely egy p prímszám, egy g generátor elem, és egy A szám esetében meghatározná azt az a kitevőt, amelyre teljesül:
$$g^a \equiv A \pmod{p}.$$
- elliptikus görbék matematikája:
  - EC-diszkrét logaritmus feltételezés: nincs hatékony algoritmus, amely egy p prímszám, egy G generátor elem és egy A pont esetében meghatározná azt az a egész számot, amelyre teljesül:
$$\underbrace{G + G + \cdots + G}_{a} = a \cdot G \equiv A \pmod{p},$$
ahol a G, A az elliptikus görbe egy-egy pontja.

A legismertebb publikus kulcsú rendszerek

- Diffie-Hellman kulcscsere, biztonsága a diszkrét logaritmus feltételezésen alapszik
- RSA (Rivest-Shamir-Adleman), biztonsága a faktorizáció feltételezésen alapszik: alkalmas titkosításra és digitális aláírásra
- Rabin, SAEP mindkettő biztonsága a faktorizáció és kvadratikus maradék feltételezésen alapszik, alkalmasak titkosításra, a Rabin digitális aláírásra is
- ElGamal, biztonsága a diszkrét logaritmus feltételezésen alapszik, a Diffie Hellman módosított változata, alkalmas titkosításra és digitális aláírásra
- ECC, elliptikus görbén alapuló kriptográfia, biztonsága az elliptikus görbe diszkrét logaritmus feltételezésen alapszik, alkalmas kulcscserére, titkosításra és digitális aláírásra
- Blum-Goldwasser kriptorendszer, biztonsága a faktorizáció és kvadratikus maradék feltételezésen alapszik, titkosításra alkalmas

A PK kulcscserék matematikai modellje

Jelölése KX, ahol a (K, M) halmazok felett 3 algoritmust értelmezünk:

- *Gen*, a publikus paramétereket generáló algoritmus, polinom idejű:
  $pPar \leftarrow Gen(1^k)$, ahol
  - $pPar \in K$ publikus paraméter,
  - $k \in \mathbb{Z}_{\geq 0}$ a rendszer biztonsági paramétere,
- *Send* a publikus kulcsot küldő algoritmus, polinom idejű, véletlenszerű, amelyet mindkét egység elvégez:

$$pk_i \leftarrow Send(pPar, sk_i), sk_i \xleftarrow{R} M, i \in \{1, 2\},$$

  - publikus kulcsok: pk_1, pk_2,
  - privát kulcsok: sk_1, sk_2,
- a *Calc* a közös kulcsot meghatározó algoritmus, polinom idejű, determinisztikus, amelyet mindkét egység elvégez:

$$key_i \leftarrow Calc(sk_i, pk_j), i \neq j, i, j \in \{1, 2\},$$

- a közös (egyeztetett) kulcs: $key = key_1 = key_2$.
- helyesség:

$$key = Calc(sk_1, pk_2) = Calc(sk_2, pk_1).$$

A PK titkosítók matematikai modellje

Jelölése PK, ahol a (K, M, C) halmaz-hármas felett 3 algoritmust értelmezünk:

- Gen, a kulcs-generáló algoritmus, polinom idejű, véletlenszerű: $(pk, sk) \xleftarrow{R} Gen(1^k)$, ahol $(pk, sk) \in K \times K$, $k \in \mathbb{Z}_{\geq 0}$ a rendszer biztonsági paramétere,
  - publikus kulcs: pk,
  - privát kulcs: sk,
- Enc a rejtjelező algoritmus, polinom idejű, véletlenszerű, amelyet az egyik egység végez: $c \xleftarrow{R} Enc(pk, m)$,
- a Dec a visszafejtő algoritmus, polinom idejű, determinisztikus, amelyet a másik egység végez: $m \leftarrow Dec(sk, c)$,
- a megosztott m csak a kommunikáló felek által ismert titkos információ,
- helyesség: $Dec(sk, (Enc(pk, m)) = m$, minden $m \in M$ esetében.

A digitális aláírások matematikai modellje

Jelölése legyen DS, ahol a (K, M, C) halmaz-hármas felett 3 algoritmust értelmezünk:

- Gen, a kulcs-generáló algoritmus, polinom idejű, véletlenszerű: $(pk, sk) \xleftarrow{R} Gen(1^k)$, ahol $(pk, sk) \in K \times K$, $k \in \mathbb{Z}_{\geq 0}$ a rendszer biztonsági paramétere,
  - publikus kulcs: pk,
  - privát kulcs: sk,
- Aut a hitelesítő algoritmus, polinom idejű, véletlenszerű, amelyet az egyik egység végez: $(m, c) \xleftarrow{R} Aut(sk, m)$, ahol $m \in P$,
- Ver az ellenőrző algoritmus, polinom idejű, determinisztikus, amelyet a másik egység végez: ha $Ver(pk, c) = m$, akkor az aláírás hiteles, ellenkező esetben nem,
- van amikor az m valamely egység publikus kulcsú titkosítójának a publikus kulcsát jelöli, ami nem ugyanaz, mint az ebben a modellben használt pk.

A publikus-kulcsú rendszerek, követelmények

- az arra jogosult fél (eszköz) hatékony algoritmussal tudja
  - kigenerálni a publikus paramétereket,
  - a publikus és privát kulcsokat,
- az arra jogosult fél (eszköz) hatékony algoritmussal tudja
  - meghatározni a rejtjelezett szöveget,
  - visszafejteni a rejtjelezett szöveget,
  - előállítani az aláírást,
  - ellenőrizni az aláírást,
- a publikus kulcs ismeretében ne lehessen hatékonyan meghatározni a privát kulcsot,
- titkosítóknál: a tikosított tartalom és publikus kulcs ismeretében ne lehessen hatékonyan meghatározni a nyílt szöveget,
- digitális aláírásoknál: a publikus kulcs ismeretében ne lehessen hatékonyan aláírást előállítani,
- a kulcscseréknél, a titkosítóknál használt publikus kulcsokat hitelesíteni kell!!
  - ezt digitális aláírás használatával lehet megoldani, nem elég a MAC-használat!!
  - a mai napig is ez a legsérülékenyebb pontja az biztonságnak
- kevés olyan rendszert sikerült kidolgozni, amely eleget tesz a fenti követelményeknek

A publikus-kulcsú rendszerek, követelmények

a rendszerekben alkalmazott matematikai objektumok átlakíthatóak kell legyenek bájtszekvenciákká, pl. fennáll:

$$63 \cdot 256^0 + 169 \cdot 256^1 + 75 \cdot 256^2 + 210 \cdot 256^3 + 184 \cdot 256^4 = 793802156351$$

a 793802156351 little order-ben:

63 169 75 210 184

```
>>> x, xLen = 793802156351, x.bit_length()//8 + 1
>>> list(x.to_bytes(xLen, byteorder='little'))
    [63, 169, 75, 210, 184, 0]
>>> int.from_bytes(bytes([63, 169, 75, 210, 184, 0]), byteorder = 'little')
    793802156351
```

a 793802156351 big order-ben:

0 184 210 75 169 63

```
>>> list(x.to_bytes(xLen, byteorder='big'))
    [0, 184, 210, 75, 169, 63]
>>> int.from_bytes(bytes([0, 184, 210, 75, 169, 63]), byteorder = 'big')
    793802156351
```

Az RSA rendszer

- a legismertebb, a leggyakrabban, a legtöbbet vizsgált, használt publikus kulcsú rendszer
- alkalmazható titkosításra (encryption) és hitelesítésre (digital signature)
- a TLS 1.3 protokoll már nem használja az encryption változatot, csak a digital signature változatot
- textbook RSA, baby RSA,
- RSA Security LLC:
  - az RSA tervezői által létrehozott szervezet,
  - általuk kibocsájtott dokumentumok PKCS (Public Key Cryptography Standrads) névvel jelennek meg, amelyek az különböző algoritmusok/protkollok működését, implementációját írják le,
  - SecurID hitelesítő token,
  - kritika: backdoor
  - RSA Conference
- IETF (Internet Engineering Task Force):
  - internetes szabványok kidolgozója
  - az általuk közreadott dokumentumok RFC (Request For Comments) névvel jelennek meg

RSA változatok

- RSA encryption
  - PKCS#1 v1.5
    - 1993, az RSA titkosító első standard változata,
    - az SSL/TLS-ben használták/használják, nem biztonságos!!!!!,
    - `PKCS#1 v1.5`
  - RSA PKCS#1 v2
    - az RSA-OAEP leírását tartalmazza, az RSA titkosító második standard változata, ezt kell használni!!!!!
    - `PKCS#1 v2`
- RSA digital signatures:
  - a PKCS#1 v1.5:
    - az RSA digitális aláírás első standard változata,
    - habár az algoritmus biztonságos, helytelen implementációk láttak napvilágot, amelyekben lehetőség van aláírást előállítani a privát kulcs ismerete nélkül is
  - PKCS#1 v2:
    - az RSA-PSS (probabilistic signatures scheme) leírása, biztonságos
    - hasonló padding technika kerül alkalmazásra, mint az RSA-OAEP-ben

Az RSA-textbook titkosító rendszer

Feltételezzük, hogy a kommunikációban résztvevő két egység **A** és **B**:

- a *Gen* kulcs generáló algoritmus segítségével meghatározásra kerül a *pk* publikus kulcs és az *sk* privát kulcs, ahol:
  - $n = p \cdot q$, p, q prímszámok, $\phi = (p-1) \cdot (q-1)$
  - $e \xleftarrow{R} \{3, \ldots, \phi\}$, úgy hogy $gcd(e, \phi) = 1$
  - a kiterjesztett euklideszi algoritmussal meghatározzuk d-t, azaz e inverzét (mod ϕ) szerint: $e \cdot d = 1 \pmod{\phi}$,
  - $pk = (n, e)$, és $sk = (n, d)$,
- az **A** egység a $Enc((e, n), m)$ algoritmussal meghatározza, majd elküldi **B**-nek a $c = m^e \pmod{n}$ értéket
- a **B** egység a $Dec((d, n), c)$ algoritmussal meghatározza a $m = c^d \pmod{n}$ értéket
- a megosztásra került, csak **A** és **B** által ismert információ: m,
- a rendszer helyessége a kis-Fermat tétellel bizonyítható, azaz:

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

Az RSA-textbook titkosító rendszer

Bizonyítás:

- tudjuk, hogy: $e \cdot d \equiv 1 \pmod{\phi}$,
- ekkor létezik egy olyan t egész szám, amelyre

$$e \cdot d = 1 + t \cdot \phi$$

- ha $(m, p) = 1$, akkor a kis Fermat-tétel alapján $m^{p-1} \equiv 1 \pmod{p}$ és:

$$m^{e \cdot d} \equiv m^{1+t \cdot \phi} \equiv m^{1+t \cdot (p-1) \cdot (q-1)} \equiv m \cdot (m^{p-1})^{t \cdot (q-1)} \equiv m \pmod{p}$$

- ha $(m, p) \neq 1$ akkor a kongruencia mindkét oldala $0 \pmod{p}$ lesz, ahol $1 < m < n$
- igaz tehát, hogy:

$$m^{e \cdot d} \equiv m \pmod{p},$$

- hasonlóan bizonyítható, hogy

$$m^{e \cdot d} \equiv m \pmod{q},$$

- mivel $n = p \cdot q$ és $p \neq q \Longrightarrow m^{e \cdot d} \equiv m \pmod{n}$

Az RSA-textbook titkosító rendszer, példa

- Kulcsgenerálás
  - Legyen $p = 61$, $q = 97$ a két prímszám.
  - Meghatározzuk:
    - $n = 61 \cdot 97 = 5917$,
    - $\phi = (p-1) \cdot (q-1) = 60 \cdot 96 = 5760$.
  - Legyen $e = 7$, ahol $(7, \phi) = 1$.
  - Meghatározzuk e inverzét $\pmod{\phi}$ szerint, kapjuk: $d = 823$, mert $7 \cdot 823 = 1 \pmod{5760}$.
  - A publikus kulcs : $(7,\ 5917)$.
  - A privát kulcs : $(823,\ 5917)$.
- Titkosítás:
  - Az $m = 2014$ üzenet értéket szeretnék titkosítani/megosztani. Ekkor a titkosított érték: $cM = 2014^{7} \equiv 1526 \pmod{5917}$.
- Visszafejtés:
  - $m = 1526^{823} \equiv 2014 \pmod{5917}$.

RSA, megjegyzések

- e - publikus exponens, d - privát exponens, n - modulus,
- a privát exponens mellett a p, q, ϕ, értékek is szigorúan titkosak; a titkosító és visszafejtő algoritmusok nem használják őket
- ahhoz, hogy a titkosítás egyértelmű legyen m legnagyobb értéke kisebb kell legyen mint n,
- ha $n = p \cdot q$, akkor az Euler függvény: $\phi(n) = (p-1) \cdot (q-1)$,
- a d értékét $\pmod{[p-1, q-1]}$ szerint is meg lehet határozni, ahol $[p-1, q-1]$ a legkisebb közös többszöröst jelöli,
- a gyorsaság miatt kis e-vel kell dolgozni, az $e = 2$ választást a Rabin, illetve SAEP specifikációik tartalmazzák, eltérő műveletvégzést jelent ez
- a biztonság és a hatékonyság miatt az $e = 65537$, amely prímszám, ekkor 17 szorzást kell végezni a titkosítás során:

$$e = 65537 = 2^{16} + 1 = 1000000000000001$$

- a standard előírja, hogy a p és a q minimum 1024 bites prímszámok legyenek, ekkor az n 2048 bites lesz,
- manapság minimum 2048 bites kulcsmérettel ajánlott használni
- ha d is megközelítőleg 2048 bites, akkor p és a q ismerete nélkül, egyelőre nincs algoritmus, amely meghatározná a d-t.

Az RSA gyorsítása

- alkalmazható a kínai maradéktétel a visszafejtés időigényének javítása érdekében,
- alkalmazásával a rendszer nem veszít biztonságából,
- ahelyett hogy az n nagyságrendjével megegyező d hatványkitevővel számolnánk, elvégzünk két kisebb (a p nagyságrendjével megegyező) hatványkitevővel való hatványozást.
- meghatározzuk dp, dq, Mq, Mp értékeket a következő módon:

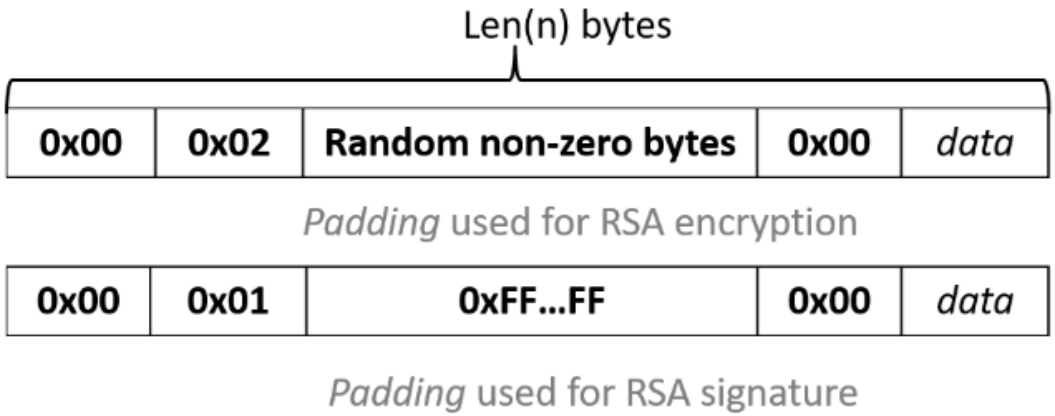
$$\begin{aligned} dp &= d \pmod{p-1} \\ dq &= d \pmod{q-1} \\ q \cdot Mq &= 1 \pmod{p} \\ p \cdot Mp &= 1 \pmod{q}. \end{aligned}$$

- A $c^d \pmod{n}$ értéket megadja az x értéke, ahol

$$\begin{aligned} x &= (Mq \cdot q \cdot xp + Mp \cdot p \cdot xq) \pmod{n} \\ xp &= c^{dp} \pmod{p} \\ xq &= c^{dq} \pmod{q}. \end{aligned}$$

A PKCS#1 v1.5 standard

- az RSA első standard változata, korábbi SSL/TLS verziókban használták/használják titkosításra, digitális aláírásra
- 1998 `Bleichenbacher támadás`: a PKCS#1 v1.5 standard szerinti RSA-titkosító feltörhető, mégis használják!
  - a kulcscsere megállapodás során a szerver leellenőrizte, hogy a visszafejtett érték megfelelő formátumú-e vagy sem, ha nem volt jó a formátum, akkor: `reject` üzenet,
  - fennáll: $r \xleftarrow{R} \mathbb{Z}_n, \hat{c} \leftarrow c \cdot r^e = (m \cdot r)^e$.
- a titkosító esetében random bitekkel egészítették ki az üzenetet, az aláírás változatnál nem:



Az RSA-OAEP titkosító rendszer

- 1995-ben Bellare és Rogaway publikálják,
- helyes paraméterezés esetében az RSA-OAEP megfelelő biztonságot nyújt,
- nem törhető fel a Bleichenbacher támadással
- PKCS#1 v2-ben kap helyet: új padding technika alkalmazása vált szükségessé

Az RSA-OAEP titkosító rendszer

- a *Gen* kulcs generáló algoritmus ugyanaz, mint a textbook RSA-nál
- a rendszer két függvényt alkalmaz egy G álvéletlen bit-generátort és egy H hash függvényt:

$$G : \{0,1\}^{l_H} \to \{0,1\}^{l_G},$$
$$H : \{0,1\}^{l_G} \to \{0,1\}^{l_H}.$$

- a standardban a H függvénynek az SHA újabb verzióját használják (min 160 bites),
- a G függvény szerkesztéséhez szintén az SHA újabb verzióját használják,
  - r véletlen bitsorozat
  - $G(r) = SHA^j(r \mid\mid < 0 >) \mid\mid SHA^j(r \mid\mid < 1 >) \mid\mid \ldots$, ahol
    - az $< i >$ jelölés: az i kettes számrendszerbeli értéke 4 bájton ábrázolva,
    - az SHA^j jelölés: az SHA függvény első j legnagyobb helyértékű bájtjai,
  - az eredmény a kapott bitszekvencia első l_G darab bitje.

Az RSA-OAEP titkosító rendszer

- $M = \{0,1\}^{l_M}$, $C = \{0,1\}^{1+l_H+l_G}$
- tipikus értékek: $k = 256$ bájt (2048 bit), $l_H = 32$ bájt (256 bit), $l_G = 223$ bájt (1784 bit), $j = 32$ bájt (256 bit), $l_R = 32$ bájt.
- ha a nyílt szöveg $l_M = 32$ bájt (256 bit), akkor az $Enc((e,n),m)$ algoritmus a c értékét a következőképpen határozza meg:
  - m-et kiegészíti: $x = 0^{l_G - l_M - 1} \,||\, 0x01 \,||\, m$,
  - $r \xleftarrow{R} \{0,1\}^{l_R}$
  - meghatározza $y = 0x00 \,||\, (x \oplus G(r)) \,||\, (r \oplus H(x \oplus G(r)))$,
  - $c = y^e \pmod n$

Az RSA-OAEP titkosító rendszer

a $Dec((d,n),c)$ algoritmus a következő:

- meghatározza az $y = c^d \pmod{n}$ értéket
- felosztja y-t: $0x00 \,||\, y_1 \,||\, y_2$-re úgy, hogy $|y_1| = l_G$ és $|y_2| = l_H$,
- meghatározza $r = H(y_1) \oplus y_2$,
- meghatározza $x = y_1 \oplus G(r)$,
- ellenőrzi, hogy x első $l_G - l_M - 1$ bitje nulla-e:
  - ha nem, akkor REJECT kimeneti értékkel leáll,
  - ellenkező esetben vissza téríti x utolsó l_M számú bitjét.

Az RSA-titkosító, Python

Kulcsgenerálás, fileba írás:

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes

def keyWrite(password, puKeyFile, prKeyFile):
    key = RSA.generate(2048)
    with open(puKeyFile, 'wb') as f:
        f.write(key.public_key().export_key())

    with open(prKeyFile, 'wb') as f:
        f.write(key.export_key(passphrase=password, pkcs=8,
            protection='PBKDF2WithHMAC-SHA512AndAES256-CBC'))

password = 'myPass000'
puKeyFile = 'publickeyRSA.pem'
prKeyFile = 'privatekeyRSA.pem'
keyWrite(password, puKeyFile, prKeyFile)
```

Az RSA-textbook titkosító, Python

```
def RSA_textbook(mess, password, puKeyFile = 'publickeyRSA.pem',
                                 prKeyFile = 'privatekeyRSA.pem'):
    temp = open(puKeyFile, 'rb').read()
    key = RSA.import_key(temp)
    n = key.n
    e = key.e
    nrMess = int.from_bytes(mess)
    cnrMess = pow(nrMess, e, n)
    messEncr = cnrMess.to_bytes(n.bit_length() // 8 + 1)
    print('encryption: ', messEncr.hex())

    temp = open(prKeyFile, 'rb').read()
    key = RSA.import_key(temp, password)
    d = key.d
    n = key.n
    messDecr = pow(cnrMess, d, n)
    print('decryption: ', messDecr.to_bytes(32).hex())
```

Az RSA-OAEP-titkosító, Python

```
def RSA_OAEP(mess, password, puKeyFile = 'publickeyRSA.pem',
                             prKeyFile = 'privatekeyRSA.pem'):
    temp = open(puKeyFile, 'rb').read()
    key = RSA.import_key(temp)
    cipher = PKCS1_OAEP.new(key)
    messEncr = cipher.encrypt(mess)
    print('encryption: ', messEncr.hex())

    temp = open(prKeyFile, 'rb').read()
    key = RSA.import_key(temp, password)
    decipher = PKCS1_OAEP.new(key)
    messDecr = decipher.decrypt(messEncr)
    print('mess decryption: ', messDecr.hex())
```

Az RSA-OAEP-titkosító, Python

Ugyanazt az üzenetet titkosítjuk RSA-textbook-kal, utána pedig RSA-OAEP-vel, mindkét esetben kétszer:

```
mess = get_random_bytes(32)
print('message: ', mess.hex())
password = 'myPass000'

RSA_textbook(mess, password)
RSA_textbook(mess, password)

RSA_OAEP(mess, password)
RSA_OAEP(mess, password)
```

Mit veszünk észre, ha megfigyeljük a titkosított tartalmakat?

Az RSA digitális aláírás, a PKCS#1 v1.5-ben

Feltételezzük, hogy a kommunikációban résztvevő két egység **A** és **B**:

- a *Gen* kulcs generáló algoritmus ugyanaz, mint a textbook RSA-nál
- az **A** egység az $Aut((d,n),m)$ algoritmussal meghatározza, majd elküldi **B**-nek a (b,c,m) értéket: $b \xleftarrow{R} \{0,1\}$ $c \leftarrow h^d \pmod{n}$, ahol $h \leftarrow H(b,m)$,
- a **B** egység a $Ver((e,n),(b,c,m))$ algoritmussal a következőket ellenőrzi le:
  - $h_1 \leftarrow c^e \pmod{n}$,
  - $h_2 \leftarrow H(b,m)$,
  - ha $h_1 = h_2$, akkor az aláírás hiteles, ellenkező esetben nem.

Az RSA digitális aláírás a PKCS#1 v2-ben

- Bellare és Rogaway publikálták 1998-ban
- az RSA-PSS az RSA Security szerint egyike a legbiztonságosabb sémáknak
- egy H hash függvényt és egy G álvéletlen bit-generátort használ, amely hasonló, mint amit az RSA-OAEP-nél is alkalmaztak
- a *Gen* kulcs generáló algoritmus ugyanaz, mint a textbook RSA-nál
- feltételezzük, hogy a kommunikációban résztvevő két egység **A** és **B**
- az **A** egység a $Aut((d,n),m)$ algoritmussal meghatározza, majd elküldi **B**-nek a (c,m) értékét:

$$
\begin{aligned}
&salt \xleftarrow{R} \{0,1\}^{l_{salt}},\\
&x = 0^{64} \;||\; H(m) \;||\; salt,\\
&r = 0^{l_y - l_{salt} - l_H - 2} \;||\; salt,\\
&y = (r \oplus G(H(x))) \;||\; H(x) \;||\; 0xbc,\\
&c = y^d \pmod{n}
\end{aligned}
$$

Az RSA-PSS digitális aláírás

A **B** egység $Ver((e,n),(c,m))$ algoritmusa a következő:

- meghatározza az $y = c^e \pmod{n}$ értéket
- ellenőrzi az y bithosszát, illetve hogy a vége egyenlő-e a 0xbc bájttal, ezt a bájtot levágja
- felosztja a levágott bitszekvenciát $y_1 \,||\, y_2$-re úgy, hogy $|y_1| = l_G$ és $|y_2| = l_H$,
- meghatározza $r = y_1 \oplus G(y_2)$,
- ellenőrzi az r hosszát, tartalmát: a felső helyértékű bitek nullások kell legyenek
- meghatározza a *salt* értékét az r alapján, ez az alsó helyértékű l_{salt} darab bit kell legyen
- létrehozza $x = 0^{64} \,||\, H(m) \,||\, salt$
- ellenőrzi, hogy fennáll-e $y_2 = H(x)$

Az RSA-PSS digitális aláírás, Python

```
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA3_256
from Crypto.Signature import pss

def messSignPSS(mess, password, prKeyFile):
    key = RSA.import_key(open(prKeyFile, 'rb', ).read(), password)
    hashV = SHA3_256.new(mess)
    signer = pss.new(key)
    signature = signer.sign(hashV)
    print(signature.hex())
    with open('signatureRSA_PSS.hex', 'wt') as f:
        f.write(signature.hex())
```

Az RSA-PSS digitális aláírás, Python

```
def messVerifyPSS(mess, puKeyFile):
    key = RSA.import_key(open(puKeyFile, 'rb').read())
    signature = open('signatureRSA_PSS.hex').read()
    signature = bytes.fromhex(signature)
    verifier = pss.new(key)
    try:
        hashV = SHA3_256.new(mess)
        verifier.verify(hashV, signature)
        print('ervenyes az alairas')
    except ValueError:
        print('ervenytelen az alairas')

password = 'myPass000'
keyWrite(password,'publickeyRSA_signPSS.pem', 'privatekeyRSA_signPSS.pem')
mess = b'myPublic rsakey for key encryption'
messSignPSS(mess, password, 'privatekeyRSA_signPSS.pem')
messVerifyPSS(mess, 'publickeyRSA_signPSS.pem')
```

RSA, biztonsági problémák

- az n faktorizálásához kapcsolódó problémák:
  - Fermat féle faktorizáció: a p és a q ne legyenek túl közel egymáshoz
  - Pollard ρ féle faktorizáció: a p, vagy a q kicsi,
  - Pollard $p-1$ féle faktorizáció: $p-1$, illetve $p+1$-nek legyen legalább egy "nagy" prímosztója,
  - ...
- az RSA kulcsok generálása során adódó hibák:
  - a p és a q értékek egyforma nagyságrendűek, egymástól függetlenek, véletlenszerűen generáltak kell legyenek, szorzatuk pedig legalább 2048 bites szám legyen
  - ugyanazt a p, vagy q értéket nem lehet egy másik modulushoz felhasználni
  - ugyanazt a modulust nem lehet különböző e értékhez felhasználni
  - e kicsi kell legyen, d azonban az n nagyságrendjével kell egyenlő legyen
  - minden egyes titkosításhoz/aláíráshoz más r random értéket kell generálni
- legjobb módszer: ha véletlenszerűen választjuk meg a prímeket!!

RSA, biztonsági problémák

a textbook RSA esetében az $e = 3$-as eset lehetővé teszi a rendszer feltörését, meghatározható az eredeti m érték, a privát kulcs ismerete nélkül.

- a kínai maradék tétel és egy köbgyököt meghatározó eljárást kell alkalmazni
- tegyük fel, hogy a m értéket háromszor titkosítottuk a $(3, n_1)$, $(3, n_2)$, $(3, n_3)$ publikus kulcsokkal, és rendre a cM_1, cM_2, cM_3 értékeket kaptuk
- a következő lineáris kongruencia rendszer adható meg:

$$\begin{aligned} x \equiv m^3 \equiv cM_1 \equiv \pmod{n_1} \\ x \equiv m^3 \equiv cM_2 \equiv \pmod{n_2} \\ x \equiv m^3 \equiv cM_3 \equiv \pmod{n_3}, \end{aligned}$$

- ezt a kínai maradéktételt alkalmazva könnyedén meg tudjuk oldani
- a kongruencia rendszer eredményeként kapott x értékből köbgyököt kell vonni:

```
>>> c = 677394484934938164599918310783958190345576648
>>> decimal.getcontext().prec = 100
>>> m = math.ceil(pow(c, 1/decimal.Decimal(3)))
```

RSA, biztonsági problémák

textbook RSA, egyforma modulusok probléma:

- ha a m értéket kétszer rejtjelezzük egyszer az (e, n), másodszor az (f, n) publikus kulcsokkal, és
- ha fennáll, hogy $(e, f) = 1$,
- akkor meghatározható a m értéke a privát kulcs ismerete nélkül

legyen cM_1, cM_2 a két rejtjelzett értéke a m-nek:

- mivel $(e, f) = 1$ kiterjesztett euklideszi algoritmussal meghatározhatjuk azokat az x, y-t, amelyekre: $e \cdot x + f \cdot y = 1$.
- a keresett m a $cM_1^x \cdot cM_2^y \pmod{n}$ érték lesz:

$$cM_1^x \cdot cM_2^y = (m^e)^x \cdot (m^f)^y = m^{e \cdot x + f \cdot y} \equiv m \pmod{n}.$$

- a kiterjesztett euklideszi algoritmus negatív értéket is számolhat x, vagy y-ban,
- ha $x < 0$, akkor szükséges meghatározni a cM_1 inverzét $\pmod{n}$ szerint, és a $cM_1^x \pmod{n}$ hatványérték helyett az $inv^{-x} \pmod{n}$ hatványértékkel kell dolgozni, ahol inv-vel a cM_1 inverzét jelöltük
- hasonlóan kell eljárni, ha $y < 0$.

RSA, biztonsági problémák

A textbook RSA, egyforma modulusok, példa:

- legyen $p = 37, q = 127, e = 11, f = 17$. Ekkor $n = 4699$ és rejtjelezzük a $m = 446$ értéket. Kapjuk:

$$cM_1 = 446^{11} \pmod{4699} = 3158$$
$$cM_2 = 446^{17} \pmod{4699} = 1757$$

- a kiterjesztett euklideszi algoritmus meghatározza a $x = -3, y = 2$ értékeket
- mivel $x < 0$ szükséges meghatároni 3158 inverzét, ez 2906 lesz
- $m : 2906^3 \cdot 1757^2 \pmod{4699} = 446$.
- Python: `pow` függvény automatikusan az alap inverzével fogja a hatványértéket számolni:

```
>>> p, q, e, f = 37, 127, 11, 17
>>> n = p * q
>>> cM1, cM2 = 3158, 1757
>>> x, y = -3, 2
>>> (pow(cM1, x, n) * pow(cM2, y, n)) % n
    446
```

Le is ellenőrizhetjük:

```
>>> (pow(2906, -x, n) * pow(cM2, y, n)) % n
    446
```

RSA, biztonsági problémák

Implementációhoz kapcsolódó problémák:

- timing attack, Koecher és tsai 1997: le lehet mérni, hogy mennyi időbe telik $c^d \pmod n$ meghatározása
- power attack, Koecher és tsai 1999: le lehet mérni, hogy mekkora energiafogyasztás szükséges $c^d \pmod n$ meghatározásához

```
def myPow(a, x, n):
    res = 1
    while x != 0:
        if x & 1 == 1:
            res = (res * a) % n
        a = (a * a) % n
        x = x >> 1
    return res
```

```
def myPow1(a, x, n):
   res = (a * a) % n
   binX = bin(x)[2:]
   for b in binX[1:] :
       if b == '0':
           res = (res * a) % n;
           a = (a * a) % n
       else:
           a = (res * a) % n;
           res = (res * res) % n
   return a
```

RSA, biztonsági problémák

Implementációhoz kapcsolódó problémák:

- faults attack, Boneh és tsai 1997: a $c^d \pmod n$ meghatározásakor fellépő számítási hibák vizsgálata:
  - ha a $xq = c^{dq} \pmod q$ meghatározásánál hiba adódik, míg az $xp = c^{dp} \pmod p$-nél nem, akkor legnagyobb közös osztót számolva lehetségessé válik az egyik prim meghatározása:

$$x \equiv (Mq \cdot q \cdot xp + Mp \cdot p \cdot xq) \pmod n$$
$$x^e \equiv c \pmod p, x^e \not\equiv c \pmod q.$$

  - tehát $x^e - c$ osztható p-vel, de nem osztható q-vel: $(x^e - c, n) = p$.

RSA, biztonsági problémák

faults attack, Boneh és tsai 1997, példa:

$$p = 13, q = 17, n = p \cdot q = 221, phi = (p-1) \cdot (q-1) = 192,$$
$$e = 5, d = 77, e \cdot d = 1 \pmod{phi}$$
$$m = 207, c = 207^5 \equiv 90 \pmod{221}$$

- visszafejtéshez a következő számítások szükségesek:

$$dp \equiv d \pmod{p-1} \equiv 5$$
$$dq \equiv d \pmod{q-1} \equiv 13$$
$$Mq = 10 = pow(p, -1, q) : 17 \cdot 10 \equiv 1 \pmod{13}$$
$$Mp = 4 = pow(q, -1, p) : 13 \cdot 4 \equiv 1 \pmod{17}$$
$$xp \equiv c^{dp} \pmod{p} = 90^5 \equiv 12 \pmod{13}$$
$$xq \equiv c^{dq} \pmod{q} = 90^{13} \equiv 3 \pmod{17}$$

- legyen a hibás érték xq meghatározásakor 7, ezért x a következő lesz:

$$x \equiv 10 \cdot 17 \cdot 12 + 4 \cdot 13 \cdot 7 \equiv 129$$

- legnagyobb közös osztót számolva meg lehet határozni a p-t:

$$129^5 \pmod{221} - 90 = 52$$
$$(52, 221) = 13$$