Kriptográfia és Információbiztonság
4. előadás

MÁRTON Gyöngyvér

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

2025

Miről volt szó az elmúlt előadáson?

- tökéletes biztonság
- számítási biztonság
- az OTP titkosítási rendszer
- folyamtitkosító rendszerek: jellemzők, biztonsági problémák
- RC4, LFSR (linear feedback shift register)

Miről lesz szó?

- folyamtitkosító rendszerek: A5/1, Salsa20, ChaCha20,
- a Poly1305 üzenet hitelesítő mód

Támadási módok

- differential cryptoanalysis (diferenciális kriptoanalízis)
  - sok eredeti/rejtjelezett pár ismeretében gyakran az összes kulcs kipróbálásának módszerénél hatékonyabb feltörés végezhető,
- side-channel típusú támadás
  - az implementáció adta gyengeségeket használják ki,
  - meghatározzák a titkosítás/visszafejtés időigényét,
  - meghatározzák az áramfogyasztást nagyságát.
- számos egyéb támadási forma létezik, amelyek kivédésére a legjobb módszer, ha nem saját implementációkat használunk, hanem a nyílt forrás-kódú könyvtárakban található implementációkkal dolgozunk, pl.
  - `Python/PyCryptodome`
  - `OpenSSL`
  - `Crypto++`
  - `NaCl: Networking and Cryptography library`,

LFSR, visszafejtés

```
def lfsr_output(key, cVect, d):
    for k in range(d):
        bit = 0
        for i in range(d):
            if cVect[i]: bit ^= key >> ((d - 1) - i)
        bit = bit & 1
        key = (key << 1) | bit
    mask = (1 << d) - 1
    return key & mask

def crypt(message, key, cVect, d):
    size = d // 8
    cMessage = bytearray()
    for i in range(0, len(message), size):
        nrMess = int.from_bytes(message[i : i + size])
        cNr = nrMess ^ key
        cMessage += cNr.to_bytes(size)
        key = lfsr_output(key, cVect, d)
    return cMessage
```

LFSR, visszafejtés

```
import base64
d = 16
cVect = [1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
key = b'hzU='
key = base64.b64decode(key)
key = int.from_bytes(key)

cryptStr = 'd35d2d91cc4acae7c9313a9b0915eb8e67fa38f0e56985d45b3b\
            9639b5aba7cee852522ff95ee9a6706722e8b3fb0373a0056781\
            f4c47e747d5203d0d0480ceea83b5d3acef7'
cryptStr = bytes.fromhex(cryptStr)

message1 = crypt(cryptStr, key, cVect, d)
print('decrypted text: ', message1.decode())
```

Az A5/1

- a GSM technológiában használják, ismert az A5/2 változat is, amely négy léptető regiszterrel dolgozik, de biztonsága gyengébb
- hardware szinten szokták implementálni,
- három léptető regiszterrel dolgozik, X, Y, Z-vel amelyek összesen 64 bitet kezelnek
  - az X 19 bites: $(x_0, x_1, \ldots, x_{18})$,
  - az Y 22 bites: $(y_0, y_1, \ldots, y_{21})$,
  - a Z 23 bites: $(z_0, z_1, \ldots, z_{22})$,
- a kulcs 64 bites, amellyel feltöltik a három regisztert
- a következő belső "majoráló/növelő" függvényt számoljuk ki minden óraimpulzusban:

$$m = maj(x_8, y_{10}, z_{10}) = \begin{cases} 0, & \text{ha a három bit között több a 0-ás} \\ 1, & \text{ha a három bit között több az 1-es} \end{cases}$$

- ha $x_8 = m$, akkor X-et léptetjük
- ha $y_{10} = m$, akkor az Y-t léptetjük
- ha $z_{10} = m$, akkor az Z-t léptetjük

Az A5/1

A kulcsfolyam előállítása a következőképpen történik:

- X léptetése:

$$\begin{aligned} t &= x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18} \\ x_i &= x_{i-1}, i = 18, 17, \ldots 1 - re \\ x_0 &= t \end{aligned}$$

- Y léptetése:

$$\begin{aligned} t &= y_{20} \oplus y_{21} \\ y_i &= y_{i-1}, i = 21, 20, \ldots, 1 - re \\ y_0 &= t \end{aligned}$$

- Z léptetése:

$$\begin{aligned} t &= z_7 \oplus z_{20} \oplus z_{21} \oplus z_{22} \\ z_i &= z_{i-1}, i = 22, 21, \ldots, 1 - re \\ z_0 &= t \end{aligned}$$

- a kulcsfolyam aktuális s bitje: $s = x_{18} \oplus y_{21} \oplus z_{22}$

A Salsa20

- a 2008-ban lezáruló eSTREAM projekthez beküldött, egyike a négy kiválasztott folyamtitkosítónak,
- az eSTREAM nem standardokat kibocsájtó intézmény, hanem bátorítani akarja a kriptográfusokat titkosítók tervezésére, titkosítók kriptoanalizisére,
- tervezője Daniel Bernstein, munkái teljesen nyilvánosak, a következő címen érhetők el:

  `https://cr.yp.to/djb.html`

- úgy hardver, mint szoftver implementációja ismert,
- a számítások párhuzamosíthatók, és a több magos processzorok esetében a titkosítás hatékonysága nagymértékben javítható,
- feltörési módszerek: az eddigi leghatásosabb módszer az összes kulcs kipróbálásának módszere,
- hatékonysága: 643 Mb/sec, $\sim$ 6-szor gyorsabb mint az RC4,
- a ChaCha20 a Salsa20 egy módosított változata,
- egy pszeudorandom függvényt alkalmaz CTR módban.

A ChaCha20 folyamtitkosító

- a Salsa20-nak egy módosított változata, tervezője szintén Bernstein,
- az alkalmazott körfüggvény nagyobb diffúziót biztosít, mint a Salsa20-é,
- az algoritmusnak elérhető úgy hardveres, mint softveres implementációja,
- viszonylag könnyen implementálható, ahol a számítások párhuzamosíthatók,
- széles körben elterjedt, használja az Open SSH, az SSL/TLS, és a Noise,
- a Google 2013-ban szabványosította, Android eszközökön is lehet használni,
- a nyílt szöveg hosszával megegyező kulcsfolyamot állít elő, amelyet $\oplus$ műveletet alkalmazva adnak hozzá a nyílt szöveghez.
- használata:
  - kommunikáló felek közti bizalmas információcsere
  - állomámyok titkosítása
  - IoT biztonság
  - Web biztonság
- változatai: ChaCha20/20 (20 kör), ChaCha20/12 (12 kör), and ChaCha20/8 (8 kör).

A ChaCha20 folyamtitkosító

- a Chacha20-Poly1305 a társított adatokkal való hitelesített titkosítást (authenticated encryption with associated data (AEAD)) tesz lehetővé:
  - a titkosítás mellett biztosítja a hitelesítését, és az integritást,
  - a rejtjelezett szöveg mérete nagyobb lesz, mert a hitelesítéshez szükséges tag-eket is tartalmazza
  - a társított adatok a nyílt szöveghez kapcsolódó metadadatok,
- a kulcsfolyam generálásához a következőképpen járnak el:
  - alkalmazásra kerül egy H pszeudorandom függvény, amelynek bemenete egy 256 bites k kulcs, és egy 96 bites r nonce (number use at once) érték:

$$\begin{aligned} ChaCha20 &: \{0,1\}^{256} \times \{0,1\}^{96} \to \{0,1\}^n \\ ChaCha20(k,r) &= H(k \,||\, 0 \,||\, r)) \,||\, H(k \,||\, 1 \,||\, r) \,||\, \ldots. \end{aligned}$$

  - a nonce érték egyedi kell legyen, de nem szükséges, hogy megjósolhatatlan (unpredictable) legyen,
  - a H függvény kimenete 512 bit.

A ChaCha20 folyamtitkosító

A H függvény bemenete:

- egy 256 bites kulcs:

$$k = k_0 \,||\, k_1 \,||\, k_2 \,||\, k_3 \,||\, k_4 \,||\, k_5 \,||\, k_6 \,||\, k_7$$

- egy 96 bites $r = r_0 \,||\, r_1 \,||\, r_2$ nonce ,
- egy nullától induló 32 biten tárolt i számláló érték.
- a $k \,||\, i \,||\, r$ bemenet egy 4×4-es mátrix alakba kerül, a kimenet is egy 4×4-es mátrix:

$$\begin{pmatrix} t_0 & t_1 & t_2 & t_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ i & r_0 & r_1 & r_2 \end{pmatrix} \longrightarrow \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix},$$

- minden mátrix elem egy 32 bites értéket jelöl (words),
- a t_0, t_1, t_2, t_3 értékek konstansok.

A ChaCha20 folyamtitkosító

A t_0, t_1, t_2, t_3 értékek meghatározása:

- az `expand 32-byte k` karakterlánc 4-esével vett ASCII kódjaiból átakaított egész szám hexadecimális értékeit jelölik:

$$t_0 = 61707865 \quad t_1 = 3320646E,$$
$$t_2 = 79622D32 \quad t_3 = 6B206574.$$

```
constChaCha = b'expand 32-byte k'
for i in range(0, 16, 4):
    t = int.from_bytes(constChaCha[i:i+4], byteorder = 'little')
    print(hex(t), end = ' ')

t = [0x61707865, 0x3320646e, 0x79622d32, 0x6b206574]
for i in range(4):
    s1 = t[i].to_bytes(8, byteorder = 'little')
    for j in range(8):
        print(chr(s1[j]), end = '')
```

A ChaCha20 folyamtitkosító

- A H függvény kimenetének meghatározásához 10-szer kerül alkalmazásra a $h : \{0,1\}^{128} \to \{0,1\}^{128}$ quarterround függvény,
- jelöljük a H bemenetét S-el, akkor a kimenet meghatározásához a következő lépeseket kell végrehajtani:

$S = t||k||i||r$
$S' = S$
for i in range(10):

$$h(x_0, x_4, x_8, x_{12})$$
$$h(x_1, x_5, x_9, x_{13})$$
$$h(x_2, x_6, x_{10}, x_{14})$$
$$h(x_3, x_7, x_{11}, x_{15})$$
$$h(x_0, x_5, x_{10}, x_{15})$$
$$h(x_1, x_6, x_{11}, x_{12})$$
$$h(x_2, x_7, x_8, x_{13})$$
$$h(x_3, x_4, x_9, x_{14})$$

$S = S \odot S'$
Szerializacio(S)

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix}$$

A ChaCha20 folyamtitkosító

Megjegyzések:

- az első négy meghívásban a mátrix első négy oszlopát dolgozza fel a h függvény,
- a következő négy meghívásban az átlókon levő elemeket,
- a mátrix minden szava kétszer is fel lesz dolgozva,
- a körök száma: $(10 \cdot 8)/4 = 20$
- a h minden hívása után átlagosan 12.5 kimeneti bit értéke lesz megváltoztatva,
- a $\odot$ jelölés $\pmod{2^{32}}$ szerinti összeadást jelent,
- a szerializáció a little endian sorrend meghatározását jelenti, pl. ha az első két szó: `0x792E4D56` és `0xF4E7A120`, akkor a szerializáció eredménye:
  `0x56, 0x4D, 0x2E, 0x79, 0x20, 0xA1, 0xE7, 0xF4`

A ChaCha20 folyamtitkosító

A *h* XOR-t, (mod 32) szerinti összeadást, és a *rot* függvényt alkalmazza:

```
BITS_INT32 = 32
def rot(b, i):
    b = (b << i) | (b >> (BITS_INT32 - i))
    return b & 0xFFFFFFFF

def h(a, b, c, d):
    a = (a + b) & 0xFFFFFFFF
    d = rot(d ^ a, 16)

    c = (c + d) & 0xFFFFFFFF
    b = rot(b ^ c, 12)

    a = (a + b) & 0xFFFFFFFF
    d = rot(d ^ a, 8)

    c = (c + d) & 0xFFFFFFFF
    b = rot(b ^ c, 7)
    return (a, b, c, d)
```

A ChaCha20 folyamtitkosító

```
from base64 import b64encode, b64decode
from Crypto.Cipher import ChaCha20
from Crypto.Random import get_random_bytes

def chacha20(plaintext, myKey, myNonce):
    cipher = ChaCha20.new(key=myKey, nonce=myNonce)
    ciphertext = cipher.encrypt(plaintext)
    print('ciphertextHex: ', ciphertext.hex())
    nonceB64 = b64encode(cipher.nonce).decode('utf-8')
    ciphertextB64 = b64encode(ciphertext).decode('utf-8')
    print('ciphertextB64:', ciphertextB64)

    try:
        nonce = b64decode(nonceB64)
        ciphertext = b64decode(ciphertextB64)
        cipher = ChaCha20.new(key=myKey, nonce=myNonce)
        plaintext = cipher.decrypt(ciphertext)
        print("The message was: " + plaintext.decode())
    except (ValueError, KeyError):
        print("Incorrect decryption")

plaintext1 = b'<!DOCTYPE html>\n<html lang="en-US"'
myKey = get_random_bytes(32)
myNonce = get_random_bytes(12)
chacha20(plaintext1, myKey, myNonce)
```

A ChaCha20 folyamtitkosító

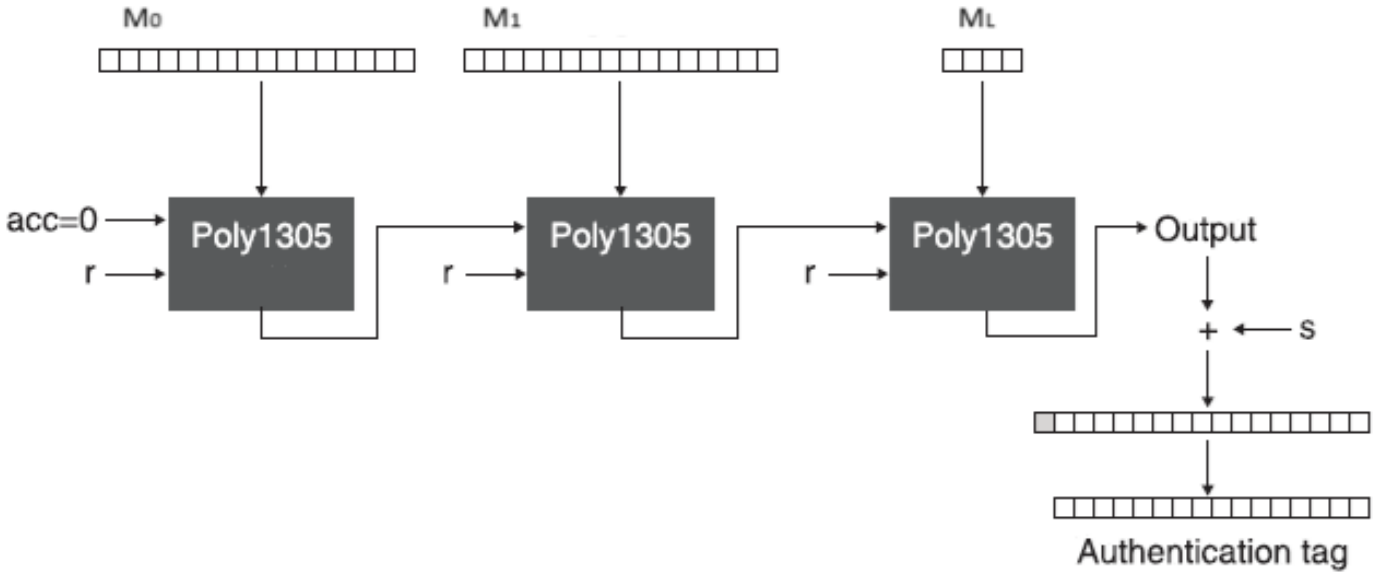
```
plaintext1 = b'<!DOCTYPE html>\n<html lang="en-US"'
ciphertextB64 = 'QsURbuDE5tANgJrcLFugQ3+6UUqRHkQoxOj8gkjGscaxqg=='
ciphertext1 = b64decode(ciphertextB64)

keyStreamRec = bytearray()
for p1, c1 in zip(plaintext1, ciphertext1):
    keyStreamRec.append(p1 ^ c1)
print('keyStreamRec: ', keyStreamRec.hex())

ciphertextB64 = 'HZYsUdf/2PIp0JrRYVTxPDGhQAfbHkQoyK/zkB+dvKnL'
ciphertext2 = b64decode(ciphertextB64)
plaintextRec = bytearray()
for p1, c1 in zip(keyStreamRec, ciphertext2):
    plaintextRec.append(p1^c1)
print('plaintext2: ', plaintextRec.decode())
```

A Poly1305 üzenethitelesítő kód

- leggyakrabban a ChaCha20 hitelesítő komponense,
- támogatja a TLS és az OpenSSH,
- a Carter-Wegman szerkesztési séma szerint hozták létre:



kép forrása: D. Wong. Real-World Cryptography. Manning. 2021.

A Poly1305 üzenethitelesítő kód

- az m üzenet hitelesítő komponensének a meghatározásához következő polinom r pontban való helyettesítési értékét kell kiszámolni:

$$Poly1305 = (c_1 r^q + c_2 r^{q-1} + \cdots + c_q r^1) \pmod{2^{130} - 5},$$

  - a $Poly1305$ bemenete egy tetszőleges hosszúságú m üzenet és egy 16 bájtos r titkos érték, kimenete pedig egy 16 bájtos érték,
  - a polinom együtthatói a c_i-k az m üzenet alapján kerülnek meghatározásra,
  - az r egy 16 bájtos érték,
  - $2^{130} - 5$ prímszám segíti úgy a szoftveres, mint hardveres implementáció hatékonyságát
- hitelesítő tag-et egy szigorúan titkos 16 bájtos s érték alapján számolják ki, ahol s különbözik r-től, és minden üzenet esetében más kell legyen:

$$t = (Poly1305 + s) \pmod{2^{128}}$$

A Poly1305 üzenethitelesítő kód

Az s és az r a 32 bájtos titkos kulcs alapján kerül meghatározásra a következőképpen:

- a titkos 32 bájtos (256 bit) kulcsot két részre osztják:
  - az első 16 bájtot *key*-el jelölik,
  - a következő 16 bájtot $r_0, r_1, \ldots, r_{15}$-el jelölik
- a *key* és egy egyszer használatos random `Nonce` értéket felhasználva az *Enc* titkosítóval számolják ki az s értékét:

$$s = Enc_{key}(\text{Nonce})$$

- az r értékét a következőképpen számolják ki:

$$r = r_0 + 2^8 r_1 + \ldots 2^{120} r_{15},$$

ahol az $r_0, r_1, \ldots, r_{15}$ bájtokra fenn kell álljon: $r_3, r_7, r_{11}, r_{15} \in \{0, 1, \ldots 15\}$ és $r_4, r_8, r_{12} \in \{0, 4, 8, \ldots, 252\}$

A Poly1305 üzenethitelesítő kód

Példa a c_i-k meghatározására:

- álljon az üzenet $k = 50$ bájtból,
- jelöljük az m üzenetet bájtjait: $m = m[0], m[1], \ldots, m[47], m[48], m[49]$,
- $q = \lceil k/16 \rceil = 4$ blokkot határozunk meg, ahol az első három blokk mindegyikében 16 bájtot, a csonka blokkban $k \equiv 2 \pmod{16}$ bájtot dolgozunk fel,
- $i = 1, 2, 3$-ra meghatározzuk a $c_i \in \{1, 2, 3, \ldots, 2^{129}\}$ értékeket:

$$c_1 = m[0] + 2^8 m[1] + 2^{16} m[2] + \cdots + 2^{120} m[15] + 2^{128}$$

$$c_2 = m[16] + 2^8 m[17] + 2^{16} m[18] + \cdots + 2^{120} m[31] + 2^{128}$$

$$c_3 = m[32] + 2^8 m[33] + 2^{34} m[2] + \cdots + 2^{120} m[47] + 2^{128}$$

- a csonka blokkra, $q = 4$-re meghatározzuk:

$$c_4 = m[48] + 2^8 m[49] + 2^{16}$$

A Poly1305 üzenethitelesítő kód

A c_i-k meghatározása:

- az $m = m[0], m[1], \ldots, m[k-1]$ üzenetet felosztjuk 16 bájtos blokkokra, ahol jelöljük k-val a bájtok számát, és legyen $q = \lceil k/16 \rceil$,
- az $i = 1, 2, \ldots q$-ra meghatározzák a $c_i \in \{1, 2, 3, \ldots, 2^{129}\}$ értékeket:
  - ha k 16 többszöröse:

$$c_i = m[16i-16] + 2^8 m[16i-15] + 2^{16} m[16i-14] + \cdots + 2^{120} m[16i-1] + 2^{128}$$

  - ha k nem 16 többszöröse, akkor még meghatározásra kerül:

$$c_q = m[16q-16] + 2^8 m[16q-15] + \cdots + 2^{8(k \bmod 16)-8} m[k-1] + 2^{8(k \bmod 16)}$$

- lényegében minden blokkból egy egész számot képezünk, amelyet kiegészítünk az 1-es értékű bájttal, 17 bájtot kapunk,
- a csonka blokkot is kiegészítjuk az 1-es értékű bájttal, majd nullás bájtokkal padingoljuk, míg 17 bájtot nem kapunk.

A ChaCha20-Poly1305 hitelesített titkosítás

- egy 256 bites *Key* kulcsot és egy 96 bites `Nonce` értéket használ a titkosításhoz
- hasonló szerkezetű, mint az AES-GCM:
  - a ChaCha20 titkosítót CTR módban alkalmazza, hogy meghatározza a kulcsfolyamot (keystream),
  - a kulcsfolyamot $\oplus$ művelettel adja hozzá a nyíltszöveghez
  - az első blokkot a hitelesítő tag létrehozására használja
- hatékonyabb, mint az AES-GCM, mert a blokkmérete 64 bájt szemeben a 16 bájtos AES-el