Kriptográfia és Információbiztonság
6. előadás

MÁRTON Gyöngyvér

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

2025

Miről volt szó az elmúlt előadáson?

- blokktitkosító rendszerek
- blokktitkosító módok: ECB, CBC, CTR
- DES, 3DES, TEA, Blowfish

Miről lesz szó?

- az AES blokktitkosító
- hash függvények (hash function)

AES (Advanced Encryption Standard)

- Daemen és Rijmen, belga kriptográfusok tervezték 1997-ben
- az 1990-ben meghírdetett nyilvános pályázat győztese
- valószínűleg nincs benne "backdoor", mert a győztes pályázat elbírálása teljesen nyilvánosan történt
- 2001-ben fogadta el a NIST standard blokktitkosítónak
- kulcsmérete: 128, 192, 256 bit, blokkmérete: 128 bit,
- szakvélemények szerint a 256 bites kulcsméret biztonsága *örök* időkre szól
- nem használja a Feistel-sémát, viszont hasonlóan a DES-hez, iterációs szerkezetű, több körből áll a titkosítás, amelyekhez körkulcsokat generál:
  - 128 bites kulcs esetén a körök száma 10
  - 192 bites kulcs esetén a körök száma 12
  - 256 bites kulcs esetén a körök száma 14
- minden körben a kulcsokon egy keverést alkalmaz, és a bemeneten, helyettesítést, permutációt és lineáris transzformációt is végrehajt
- nem találtak sikeres támadást ellen, a mai napig az implementációkból adódó gyengeségek okozzák a biztonsági problémákat

AES (Advanced Encryption Standard)

- a 128 bites, azaz a 16 bájtos $S_1 S_2 \dots S_{16}$ bemenetet egy 4×4-es mátrix alakban dolgozza fel:

$$\begin{matrix} S_1 & S_5 & S_9 & S_{13} \\ S_2 & S_6 & S_{10} & S_{14} \\ S_3 & S_7 & S_{11} & S_{15} \\ S_4 & S_8 & s_{12} & S_{16} \end{matrix}$$

- minden bájt a $GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$ test eleme:
  - minden S_i bájt megfeleltethető egy $n = 7$ fokszámú polinomnak, ahol az együtthatók értéke 0 vagy 1,
  - a $b_7 b_6 \dots b_1 b_0$ bájt megfelel a

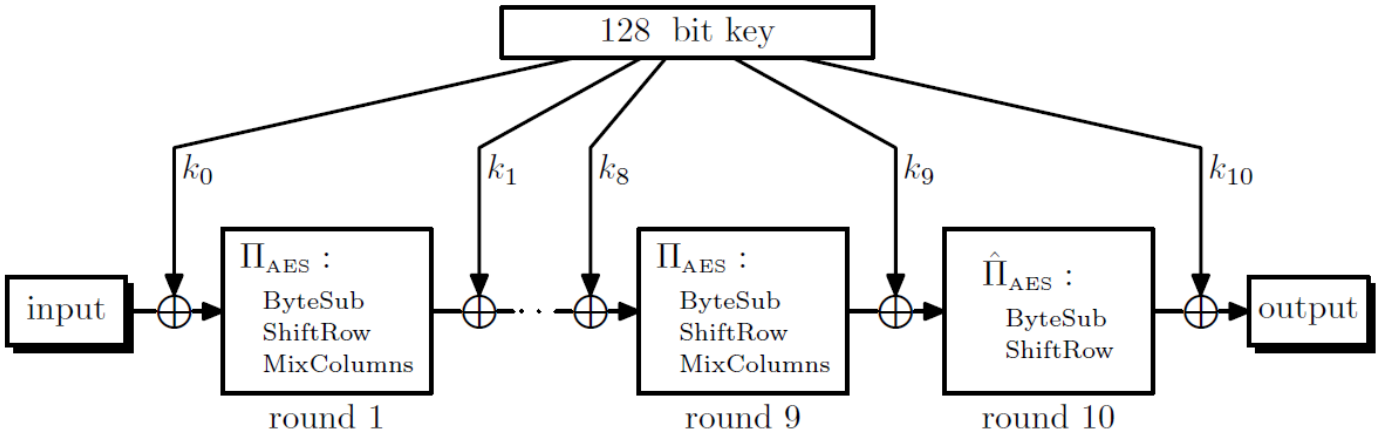
    $b_7 \cdot x^7 + \dots + b_1 \cdot x + b_0$ polinomnak, ahol $b_1, ..., b_7 \in \{0, 1\}$
  - a műveletvégzések után az eredménynek meghatározzák az $m(x) = x^8 + x^4 + x^3 + x + 1$ polinom szerinti osztási maradékát,
  - $m(x)$ egy 8-ad fokú irreducibilis polinom: olyan polinom, amely nem bontható fel két 8-nál kisebb fokú polinom szorzatára.
- Példa:

  $x^6 + x^3 + x$-nek megfelelő bináris alak: 0100 1010.

AES (Advanced Encryption Standard)

- a körkulcsok meghatározásához a $128 = 16 \cdot 8$ bites kulcsot oszloponként szintén egy 4×4-es kétdimenziós mátrix formába írják, ahol a mátrixelemeken ciklikus eltolást, szorzást, multiplikatív inverzet számolnak a $GF(2^8)$ feletti véges test szabályai szerint,
- az összeadás, kivonás, szorzás, osztás tehát a $GF(2^8)$ feletti véges testben történik:
  - a műveleteket a szabályos polinomok feletti műveleti szabályok szerint kell végezni, ahol az együtthatók esetében $\pmod 2$ kell számolni,
  - ha egy művelet elvégzése után nagyobb kitevőt kapunk mint x^7, akkor meg kell határozni az eredmény $m(x) = x^8 + x^4 + x^3 + x + 1$ szerinti osztási maradékát, ahol $m(x)$ egy 8-ad fokú irreducibilis polinom.

AES-128 titkosítás



kép forrása: *Boneh and Shoup/DES*

- ByteSub: egy fix permutáció, nem lineáris művelet, a bemeneti mátrix bájtjain alkalmazza az S-boxot,
- ShiftRow: ciklikus biteltolást, lineáris műveletet végez,
- MixColumns: a mátrix elemein szorzásokat, lineáris átalakításokat végez a $GF(2^8)$ véges test szerint,
- a lineáris és affin átalakítások lavina effektust eredményeznek.

AES-128 titkosítás, S-box

- az S-box elemei előre ki vannak számítva, és egy táblázatban vannak tárolva,
- az S-box egy bájtos bemenetének bitjeit egy polinom együtthatóinak tekintik:
  - meghatározzák a polinom multiplikatív inverzét a $GF(2^8)$ véges testben,
  - majd egy affin transzformációval megkapják az S-box kimeneti bitjeit
- Például a 0x4a bemenetre 0xd6 a kimenet, mert:
  - 0x4a= $(0100\ 1010) = x^6 + x^3 + x$ multiplikatív inverze:
    $x^7 + x^5 + x^3 + x + 1 = (1010\ 1011)$ lesz, fennáll

    $$(x^6 + x^3 + x) \cdot (x^7 + x^5 + x^3 + x + 1) = 1 \pmod{x^8 + x^4 + x^3 + x + 1}$$

  - az affin transzformácó után kapjuk:
    $nb = (1101\ 0110) =$ 0xd6, mert:

$$
\begin{aligned}
nb_0 &= b_0 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7 \oplus c_0 = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 0 \\
nb_1 &= b_1 \oplus b_5 \oplus b_6 \oplus b_7 \oplus b_0 \oplus c_1 = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1 \\
nb_2 &= b_2 \oplus b_6 \oplus b_7 \oplus b_0 \oplus b_1 \oplus c_2 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1 \\
nb_3 &= b_3 \oplus b_7 \oplus b_0 \oplus b_1 \oplus b_2 \oplus c_3 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 0 \\
nb_4 &= b_4 \oplus b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus c_4 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 = 1 \\
nb_5 &= b_5 \oplus b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus c_5 = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 0 \\
nb_6 &= b_6 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus c_6 = 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1 \\
nb_7 &= b_7 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6 \oplus c_7 = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 1
\end{aligned}
$$

AES-128 titkosítás, S-box

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

AES visszafejtés

- a blokktitkosítókra jellemzően a körkulcsokat fordított sorrendbe alkalmazzák,
- a titkosításnál alkalmazott sorrendhez képest az alkalmazott függvényeket fordított sorrendben veszik ,
- a ByteSub, ShiftRows, MixColumns, S-box függvények inverzeit használják $\rightarrow$ a titkosítás nem ugyanaz, mint a visszafejtés
  - titkosítási sorrend: ByteSub, ShiftRows, MixColumns,
  - visszafejtési sorrend: InvMixColumns, InvShiftRows, InvByteSub
- meg kell külön írni a titkosító és visszafejtő függvényt $\rightarrow$ hátrány, de lehet ezt optimalizálni

AES, bináris file titkosítása, CBC mód

```
import os
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

BS = AES.block_size
CS = 128 * BS

def encryptAES_CBC(inFile, outFile, key, iv):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    with open(inFile, 'rb') as inF, open(outFile, 'wb') as outF:
        cipher = AES.new(key, AES.MODE_CBC, iv)
        for i in range(nrBlock):
            tBlock = inF.read(CS)
            tBlock = cipher.encrypt(tBlock)
            outF.write(tBlock)
        tBlock = inF.read(CS)
        tBlock = cipher.encrypt(pad(tBlock, BS))
        outF.write(tBlock)
```

AES, bináris file titkosítása, CBC mód

```
def decryptAES_CBC(inFile, outFile, key, iv):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    with open(inFile, 'rb') as inF, open(outFile,'wb') as outF:
        cipher = AES.new(key, AES.MODE_CBC, iv)
        for i in range(nrBlock):
            tBlock = inF.read(CS)
            tBlock = cipher.decrypt(tBlock)
            outF.write(tBlock)
        tBlock = inF.read(CS)
        tBlock = cipher.decrypt(tBlock)
        tBlock = unpad(tBlock, BS)
        outF.write(tBlock)
```

AES, bináris file titkosítása, CBC mód

```
iv = get_random_bytes(BS)
keyS = AES.key_size
key = get_random_bytes(keyS[2])
print('blocksize: ', BS, '\nkey sizes: ', keyS, '\nkey: ', key.hex())

plaintextFile = ...
encryptedFile = ...
decryptedFile = ...

encryptAES_CBC(plaintextFile, encryptedFile, key, iv)
decryptAES_CBC(encryptedFile, decryptedFile, key, iv)
with open(plaintextFile, 'rb') as inF1, open(decryptedFile, 'rb') as inF2:
        assert( inF1.read() == inF2.read() )
```

AES, string titkosítása, CTR mód

```
from Crypto.Util import Counter
nonce = get_random_bytes(BS//2)
print('nonce: ', nonce.hex())
keyS = AES.key_size
print('key sizes: ', keyS)
key = get_random_bytes(keyS[2])
print('key: ', key.hex())

plaintext = 'One of the most widely used security services is Transport
        Layer Security(TLS); the current version is Version 1.3'
plaintext = plaintext.encode()
ctr = Counter.new(BS * 8 // 2, prefix=nonce, initial_value  = 10)
cipher = AES.new(key, AES.MODE_CTR, counter=ctr)
ciphertext = cipher.encrypt(plaintext)
print('ciphertext: ', ciphertext.hex())

ctr = Counter.new(BS * 8 // 2, prefix=nonce, initial_value  = 10)
decipher = AES.new(key, AES.MODE_CTR, counter=ctr)
decryptedtext = decipher.decrypt(ciphertext)
print('decrypted text: ', decryptedtext.decode())
assert(decryptedtext == plaintext)
```

Hash függvények

- a H hash függvény a bemeneti tetszőleges hosszúságú M adatszekvenciára egy fix méretű $h = H(M)$ hash értéket állít elő
- a "jó" hash függvény egyenletes eloszlású és látszólag véletlenszerű kimenetet eredményez
- a biztonsági alkalmazásokhoz szükséges hash függvényt kriptográfiai hash függvénynek nevezzük
- a kriptográfiai hash függvény több biztonsági követelménynek is eleget kell tegyen
- egyike a legsokoldalúbb kriptográfiai primitívnek (algoritmusnak): számos biztonsági alkalmazásban, internetes protokollban használják
- elsődleges alkalmazási területük az adatintegritás, de használják:
  - jelszavak tárolásakor
  - titkosító rendszerekben
  - kulcscsere protokollokban
  - digitális aláírásokhoz
  - blokkláncok létrehozásakor

Hash függvények

jelszavak tárolása esetében:

- biztonsági okokból nem közvetlenül a jelszavaknak megfelelő bájtszekvenciát tárolják, hanem egy hashfüggvény segítségével egy másik bájt szekvenciát képeznek és ezt tárolják el
- a hashfüggvény alkalmazása előtt a jelszón egy "normalizálást" is elvégeznek: azaz egy kulcsképző függvényt (key derivation function) és egy "toldalék"-bájtszekvenciát (salt-ot) felhasználva, egy újabb bájtszekvenciát állítanak elő
- a hash függvény kimeneti értéke mellett, eltárolásra kerül a salt értéke is

Hash függvények

blokkláncok esetében:

- a blokklánc összekapcsolt blokkok sorozata, ahol minden blokk tartalmazza az előző blokk hash értékét
- Pl. egy blokkláncban hatékonyan lehet pénzügyi tranzakciókat tárolni, mert a blokkban lévő tranzakciók nem módosíthatók, csak ha a tranzakciót megelőző összes hash-értéket megváltoztatjuk

bitTorrent esetében:

- állományok cseréje érdekében az állományokat feldarabolják csonkokra
- minden egyes csonknak meghatározzák a hash értékét, amelyeket megosztanak
- a megosztott hash értékek alapján lehet a biztonságos letöltést megoldani

Hash függvények, tulajdonságok

Az alábbi problémák ne legyenek megoldhatóak polinomiális időben:

- 1. probléma, egyirányú (preimage) tulajdonság:
  Bemenet: h, és $y \in Y$
  Kimenet: határozzuk meg x-t,úgy hogy: $h(x) = y$
- 2. probléma, gyengén ütközésmentes (second preimage) tulajdonság:
  Bemenet: h, és $x \in X$
  Kimenet: határozzuk meg x_1-t,úgy hogy: $x_1 \neq x$ és $h(x) = h(x_1)$
- 3. probléma, ütközésmentes (collision) tulajdonság:
  Bemenet: h
  Kimenet: határozzuk meg x, x_1-t,úgy hogy: $x_1 \neq x$ és $h(x) = h(x_1)$

Gyakran összekavarják a gyengén ütközésmentes tulajdonságot, az ütközésmentes tulajdonsággal!

Hash függvények, tulajdonságok

- kulcs nélküli függvények, de létezik kulcsos változatuk is,
- egy $h : X \to Y$, hash függvény esetében az X jóval nagyobb elemszámú, mint az Y, ez azt is jelenti, hogy ütközés fog fennállni: lesz két különböző bemenet, amelynek ugyanaz lesz a hash értéke,
- hatékonyan, determinisztikusan számítja ki a hash értékét,
- ugyanarra a bemenetre mindig ugyanazt a kimeneti értéket határozza meg,
- általában 16-os számrendszerben, vagy base64-es alakban szokták a kimeneti értéket tárolni,
- lavina effektus: a bemenet egyetlen bitjének a megváltoztatása a teljes kimenet változásával kell járjon,
- a hash kimenete legalább annyi bit kell legyen, hogy ne lehessen brute-force támadást alkalmazni

Hash függvények, születésnap paradoxon

- egy $h : X \to Y$ hash függvény esetében a X jóval nagyobb elemszámú, mint az Y, ez azt is jelenti, hogy ütközések mindig fennállnak
- a születésnapi paradoxon megmutatja, hogy az ütközések meglepően gyakran előállhatnak
- születésnapi paradoxon: 23 véletlenszerűen kiválasztott ember esetében, legalább 50%-os az esélye annak, hogy legalább ketten lesznek olyanok, akiknek ugyanazon a napon van a születésnapjuk
- bebizonyítható, hogy ha $M = 365$, akkor elég

$$1.17 \cdot \sqrt{M} = 1.17 \cdot \sqrt{365} = 1.17 \cdot 19.10 = 22.35$$

napot kiválasztani ahhoz, hogy 50%-nál nagyobb eséllyel találjunk ütközést, azaz, hogy lesz két egyforma nap, amit kiválasztottunk.

Hash függvények, születésnap paradoxon

- legyen M a lehetséges születésnapok száma, és Q a kiválasztásra kerülő napok száma, akkor annak a valószínűsége, hogy ne legyen ugyanazon a napon két születésnap a következő:

$$\left(\frac{M-1}{M}\right)\cdot\left(\frac{M-2}{M}\right)\dots\left(\frac{M-Q+1}{M}\right)$$

- felhasználva a 2. sorban $1-x\approx e^{-x}$, majd a 3. sorban ε-al jelölve a valószínűséget, és a 6. sorban a $\varepsilon = 0.5$ helyettesítést alkalmazva kapjuk

1. $\left(1-\frac{1}{M}\right)\cdot\left(1-\frac{2}{M}\right)\dots\left(1-\frac{Q-1}{M}\right) =$
2. $\left(e^{\frac{-1}{M}}\right)\cdot\left(e^{\frac{-2}{M}}\right)\dots\left(e^{\frac{Q-1}{M}}\right) = e^{-(\frac{1}{M}+\frac{2}{m}+\dots+\frac{Q-1}{M})} = e^{\frac{-Q(Q-1)}{2M}}$
3. $e^{\frac{-Q(Q-1)}{2M}} \approx 1-\varepsilon$
4. $\frac{-Q(Q-1)}{2M} \approx ln(1-\varepsilon)$
5. $Q^2 - Q \approx 2M\cdot ln\frac{1}{(1-\varepsilon)}$
6. $Q \approx \sqrt{2M\cdot ln\frac{1}{(1-\varepsilon)}} \approx 1.17\sqrt{M}$

Hash függvények, születésnap paradoxon

- Pythonban ellenőrizve:

```
p = 1
for i in range(22, 0, -1): p *= (1 - i/365)
print(p)
```

- két ugyanazzal a születésnappal rendelkező ember kiválasztása ugyanazt jelenti, mint ütközést találni egy hash függvény esetében
- pl. egy 40 bites hash nem nyújt biztonságot, mert ekkor $M = 2^{40}$ és $\sqrt{2^{40}} = 2^{20} \simeq 10^6$ hash értéket valós időben meg lehet határozni ($1.17 \cdot 2^{20} = 1226833.92$), ami pedig 50%-nál nagyobb valószínűséggel eredményez ütközést,
- a jelenlegi ajánlások minimum 256 bites kimenetű hash függvények használatát írják elő.

Hash függvények

- MD2, MD4, MD5, MD6:
  - az MD4-t Ron Rivest publikálta 1990-ben
  - az MD5 az MD4 egy továbbfejlesztett változata, 1992-ben
  - 2004-ben komoly biztonsági problémák merültek fel az MD5-el kapcsolatosan
- SHA, SHA-1, SHA-2:
  - Merkle-Damgård szerkezetű
  - SHA-t a NIST publikálta először 1993-ban
  - SHA-1 az SHA kisebb módosításokkal, 160 bites a kimenete, 2017-ben találtak rá ütközést
  - SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512): hasonló szerkesztésű, mint az SHA-1, de a belső állapot mérete 256 bit
  - a 256, 384, 512 bites változatok 2002 óta standardok
  - a 224-es változat 2004 óta standard

Hash függvények

- SHA-3
  - 2007-ben a a NIST pályázatot írt ki, 2012-ben a *Keccak* nyert: biztonság, flexibilitás, egyszerűség a jellemzői
  - szerkesztése eltér a korábbi módszerektől: nem Merkle-Damgård szerkezetű
  - SHA3-224, SHA3-256, SHA3-384, SHA3-512 változatok
- RIPEMD-128-160-256-320
  - először Belgiumban publikálták, 1996-ban
  - szabad felhasználási lehetőség, nincs levédve
- WHIRLPOOL
  - 2000-ben publikálták,
  - egyik szerkesztője Rijmen,
  - az ISO által elfogadott nemzetközi standard.

Hash függvények, általános szerkesztés

A legismertebb, legelterjetteb szerkesztési mód a Merkle-Damgård szerkesztés:

- legyen $h : X \to Y$ a hash függvény
- ha az x bitsorozat hash értékét akarjuk megszerkeszteni:
  - felosztjuk x-et l-bit hosszúságú bit-sorozatokra: $x_1, x_2, \ldots x_k$,
  - szükség esetén kiegészítjük x-et további bitekkel
  - definiálunk egy f tömörítő függvényt, amelyet rekurzívan alkalmazunk
  - meghatározunk egy kezdeti IV értéket, amely betöltheti a kulcs szerepét is, általában azonban előredefiniált konstans értéket jelent
  - az általános szerkesztési lépéssorozat:

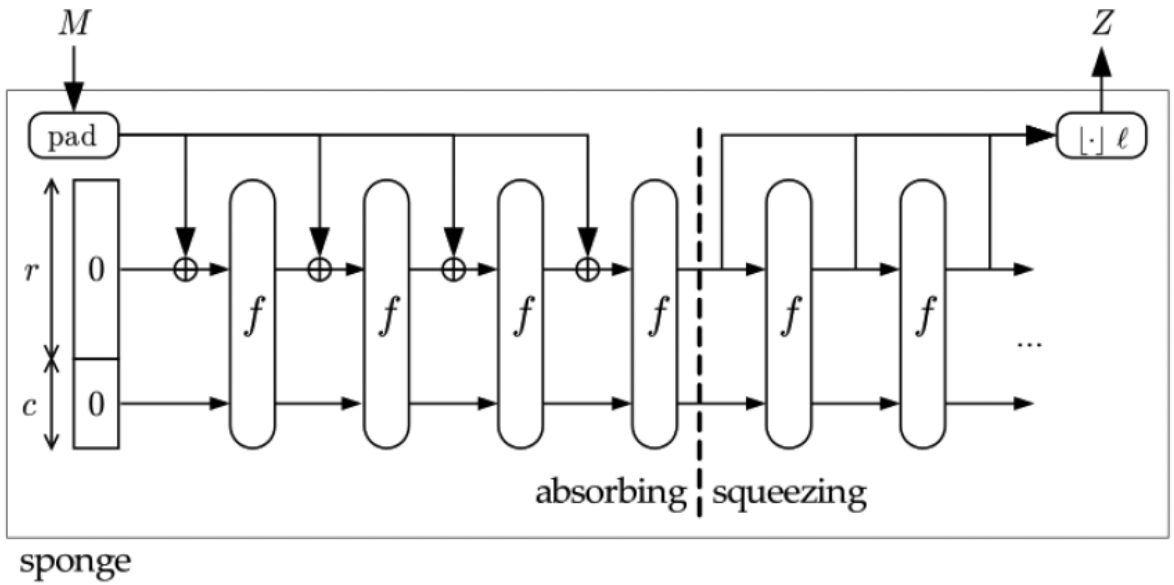
$$\begin{aligned} H_0 &= IV \\ H_i &= f(H_{i-1}, x_i), i = 1, 2, ..., k \\ h(x) &= H_k \end{aligned}$$

  - az f függvény és a IV határozzák meg a hash függvény biztonságát

Hash függvények, általános szerkesztés

A másik, újabban használt szerkesztési mód a szivacs (sponge) szerkesztés:

- Bertoni, Daemen, Peeters és Van Assche fejlesztették
- a tömörítő függvény helyett alkalmazott f függvény (ez általában bijektív) fix hosszúságú bitsorozatból azonos hosszúságú bitsorozatot határoz meg
- a sponge szerkezet lehetővé teszi a tetszőleges hosszúságú kimenetet
- $f : \{0,1\}^b \to \{0,1\}^b$, esetében a $b = r + c$ egész számot szélességnek (width), az r-et bitmértéknek (bitrate), és a c-t kapacitásnak (capacity) hívjuk
- r bitenként dolgozzuk fel az üzenetet, ezért az r meghatározza a sponge függvény hatékonyságát, a c biztonságáért felelős,
- az eljárás eredményeként l bit kerül meghatározásra.



Hash függvények, általános szerkesztés

a szivacs (sponge) szerkesztés két részre osztható: elnyelő fázis (absorbing phase) és összenyomó fázis (squeezing phase)

- az absorbing phase kimenete az $x_k||y_k$ bitsorozat, amelynek hossza $r + c$:
  - $M = m_1||\ldots m_k$, ahol $m_1, \ldots, m_k \in \{0,1\}^r$,
  - szükség esetén kiegészítjük M-et további bitekkel, azért hogy osztható legyen r-el
  - legyen $x_0 = \underbrace{00\ldots0}_{r}$ és $y_0 = \underbrace{00\ldots0}_{c}$ a kezdeti állapotérték
  - az aktuális állapotérték: $x_i||y_i$, ahol:

$$x_i||y_i \quad = \quad f_{i-1}(x_{i-1} \oplus m_i||y_{i-1}), i = 1, 2, ..., k$$

- squeezing phase:
  - ha $l \leq r$, akkor a hash függvény kimenete az x_k első l bitje lesz
  - ha $l > r$, akkor mégegyszer alkalmazzuk az f függvényt az aktuális állapotértéken és meghatározunk r kimeneti bitet, a folyamatot addig ismételjük, amíg el nem érjük a kívánt l hosszúságot

Az SHA3 (Secure Hash Algorithm) hash függvény

- a Keccac alapján szerkesztették
- az állapotérték (state) az $b = r + c$ egy $5 \times 5 \times 64 = 1600$ bites érték:
- az $f : \{0,1\}^{1600} \rightarrow \{0,1\}^{1600}$ függvény bijektív:
  - kijelenthető, hogy hasonló a random permutációhoz
  - hasonló az AES-ben alkalmazott függvényhez, ez nem véletlen mert az egyik tervező részt vett az AES tervezésében (Daemen)
  - XOR, AND, NOT bitműveleteket használ, ezért gyors úgy a szoftveres, mint hardveres implementáció

Hash-kimenet	r (rate)	c (capacity)	r + c (state/állapotérték)
224	1152	448	1600
256	1088	512	1600
384	832	768	1600
512	575	1024	1600

pycryptodome, SHA3

```
from base64 import b64encode, b64decode
from Crypto.Hash import SHA3_256

def createPassword1(name, paswd):
    paswd = paswd.encode()
    hashS = SHA3_256.new()
    hashS.update(paswd)
    keyD = ['name', 'password']
    valueD = [name, b64encode(hashS.digest())]
    person = dict(zip(keyD, valueD))
    return person
```

pycryptodome, SHA3

```
def checkPassword1(person, paswd):
    paswd = paswd.encode()
    hashS = SHA3_256.new()
    hashS = hashS.update(paswd)
    if hashS.digest() == b64decode(person['password']):
        print('hash ok')
    else: print('somthing wrong')

person = createPassword1('Kiss Attila', 'm64Password')
print(person)
checkPassword1(person, 'my64Password')

person = createPassword1('Kiss Tibor', 'my64Password')
print(person)
checkPassword1(person, 'my64Password')
```

hashlib, sha256

```
from base64 import b64encode, b64decode
import hashlib
from Crypto.Random import get_random_bytes

def createPassword2(name, paswd):
    paswd = paswd.encode()
    salt = get_random_bytes(32)
    hashS = hashlib.pbkdf2_hmac('sha256', paswd, salt, 1000)
    keyD = ['name', 'password', 'salt']
    valueD = [name, b64encode(hashS), b64encode(salt)]
    person = dict(zip(keyD, valueD))
    return person
```

hashlib, sha256

```
def checkPassword2(person, paswd):
    paswd = paswd.encode()
    salt = b64decode(person['salt'])
    hashS = hashlib.pbkdf2_hmac('sha256', paswd, salt, 1000)
    print(b64encode(hashS))
    if hashS == b64decode(person['password']):
        print('hash ok')
    else: print('somthing wrong')

person = createPassword2('Kiss Attila', 'my64Password')
print(person)
checkPassword2(person, 'my64Password')

person = createPassword2('Kiss Tibor', 'my64Password')
print(person)
checkPassword2(person, 'my64Password')
```

scrypt

```
from base64 import b64encode, b64decode
import scrypt
from Crypto.Random import get_random_bytes

def createPassword3(name, paswd):
    paswd = paswd.encode()
    salt = get_random_bytes(32)
    hashS = scrypt.hash(paswd, salt, 1048576, 8, 1, 32)
    keyD = ['name', 'password', 'salt']
    valueD = [name, b64encode(hashS), b64encode(salt)]
    person = dict(zip(keyD, valueD))
    return person
```

scrypt

```
def checkPassword3(person, paswd):
    paswd = paswd.encode()
    salt = b64decode(person['salt'])
    hashS = scrypt.hash(paswd, salt, 1048576, 8, 1, 32)
    if hashS == b64decode(person['password']):
        print('hash ok')
    else: print('somthing wrong')

person = createPassword3('Kiss Attila', 'my64Password')
print(person)
checkPassword3(person, 'my64Password')

person = createPassword3('Kiss Tibor', 'my64Password')
print(person)
checkPassword3(person, 'my64Password')
```

scrypt, pbkdf2 - különböző package-ek

```
from Crypto.Random import get_random_bytes
from Crypto.Protocol.KDF import PBKDF2, scrypt
from Crypto.Hash import SHA3_256
import hashlib
import scrypt as sc

paswd = 'mypassword'.encode()
salt = get_random_bytes(16)

print('password base key derivation function:')
hashS1 = PBKDF2(paswd, salt, 32, 100000, hmac_hash_module=SHA3_256)
hashS2 = hashlib.pbkdf2_hmac('sha3_256', paswd, salt, 100000)
print(hashS1.hex())
print(hashS2.hex())

print('scrypt:')
hashS1 = sc.hash(paswd, salt, 2 ** 20, 8, 1, 32)
hashS2 = scrypt(paswd, salt, 32, N=2 ** 20, r=8, p=1)
print(hashS1.hex())
print(hashS2.hex())
```