Kryptografické hašovací funkce - MAC, MDC kódy

často stačí pouze autentizace původu či obsahu zprávy, případně ověření integrity
ne vždy stačí k zajištění těchto požadavků vlastní šifrovací algoritmus

Použití

- symetrické systémy - slouží k rozpoznání pravosti dešifrované zprávy
- asymetrické systémy - rychlejší autentizace: spočítá se hašovací funkce nad danou zprávou a elektronicky se podepíše až výsledný hashkód
- ochrana hesel a passphrases

Závisí-li výpočet hašovací funkce na tajném klíči, označujeme tuto funkci jako MAC (message authentization code). Pokud takový klíč použit není, jde o MDC (manipulation detection code).

Semi-formální definice

MAC je funkce h splňující

1. zatímco argument X může být libovolné délky, výsledek $h(K, X)$ má pevnou velikost n ($n \geq 32 \dots 64$) 128 ..
2. pro dané h a X je těžké určit $h(K, X)$ s pravděpodobností úspěchu výrazněji zvyšující -1.21548(c)0.130407(h)-4.59096 4054. Td [(p)-4.97907(á)0.105548(r)333]TJ /

není známa žádná dokazatelně bezpečná hašovací funkce, pouze známe funkce ekvivalentní s NP-úplnými problémy

Návrhy MDC funkcí

prakticky všechny známé hašovací funkce pracují nad vstupem pevné délky
iterované hašovací funkce (iterated hash f.) - delší vstup X je zpracováván opakováním výpočtu

$$H_0 = IV$$

$$H_i = \mathbf{f}\left(X_i, H_{i-1}\right) \qquad i = 1,\ 2,\ \ldots\ ,\ \mathrm{t}$$

$$\mathbf{h}(X) = H_t$$

IV - inicializační hodnota

často délka vstupu není násobkem délky vstupního bloku → nutné zarovnání (padding)
zarovnání by mělo být jednoznačné (unambiguous) - nesmí dvě různé zprávy doplnit na stejnou, je vhodné, aby na konec zprávy zapisovalo délku

Tvrzení
Pokud zarovnání obsahuje délku vstupní zprávy a tato je dlouhá alespoň 2 bloky, potom nalezení kolidujícího vzoru **h** při pevném IV vyžaduje 2^n operací právě když nalezení kolidujícího vzoru **f** při libovolném H_{i-1} vyžaduje 2^n operací.

Tvrzení
Předpokládejme jednoznačné zarovnání obsahující délku vstupní zprávy. Je-li **f** odolná vůči kolizím (collision resistant), je **h** CRHF.

Hašovací funkce založené na blokových šifrách

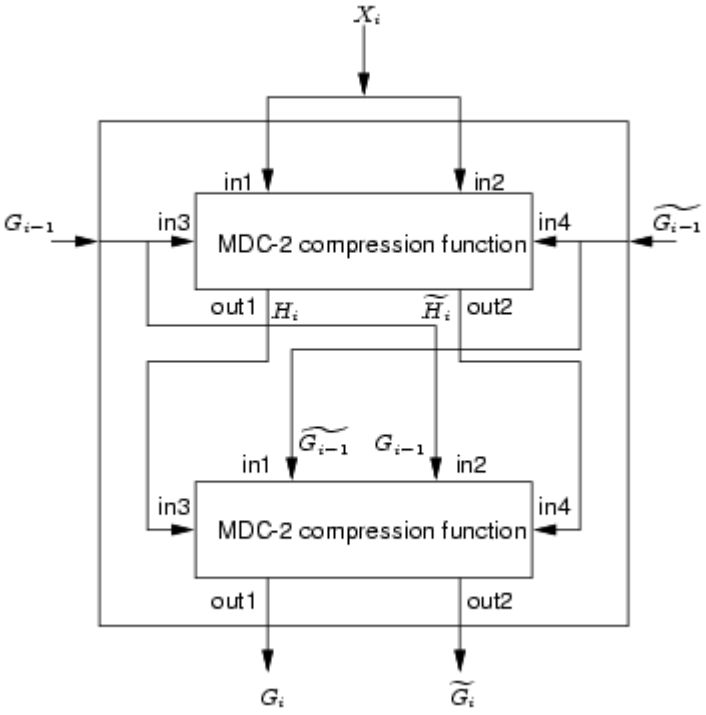
výhodné z hlediska návrhu a implementace
většina schémat ekvivalentní s jedním z následujících

Matyas, Meyer, Oseas

$$\mathbf{f} = E^{\oplus}\left(s\left(H_{i-1}\right), X_i\right)$$

délka hashkódu odpovídá délce vstupu, $E^{\oplus}(K, X) = E(K, X) \oplus X$.
ISO/IEC 10118 p. 2

jeden krok se skládá ze dvou iterací MDC-2, jako vstup druhé iterace se použije $H2_{i-1}$ a $H1_{i-1}$.



Funkci dnes považujeme za bezpečnou při použití bezpečného šifrovacího algoritmu.

Hašovací funkce založené na modulární aritmetice

nejlepší schémata založena na umocňování na druhou modulo *n*

$$\mathbf{f} = \left(X_i \oplus H_{i-1} \right)^2 \mathbf{mod}\, N \oplus X_i$$

použitím většího množství operací lze ještě zvýšit bezpečnost

$$\mathbf{f} = \left(H_{i-1} \oplus \left(X_i \right)^2 \right)^2 \mathbf{mod}\, N$$

Hašovací funkce založené na problému batohu

zatím není jasné, zda problém batohu je těžký pouze v nejhorším případě, nebo zda je těžký v průměrném případě
prakticky všechny systémy založené na problému batohu byly prolomeny

Speciální MDC funkce

algoritmy navrhované od počátku jako výpočet hašovacích funkcí a potažmo MDC obvykle výkonnější

MD4, MD5

MD4 navržen Rivestem, produkuje 128 bitový výstup, iterace ve třech cyklech
publikován útok proti prvním dvoum cyklům algoritmu
autor provedl vylepšení → MD5

MD5 pracuje nad vstupními bloky délky 512 bitů, 128 bitový výstup, pracuje ve čtyřech cyklech
algoritmus patří do stejné „rodiny“ hashovacích funkcí, jako SHA1 a má i obdobnou strukturu
v současné době publikovány útoky na kompresní funkci MD5, které sice neznamenají aktuální nebezpečí pro všechny implementace využívající MD5, nicméně svědčí o slabosti algoritmu
dle posledních výsledků lze oba algoritmy považovat za prolomené, což platí obecně pro všechny algoritmy s délkou „digestu“ do 160 bitů

SHA-1

message digest funkce transformující zprávu libovolné délky na 160-bitový digest. Pracuje ve třech krocích:

- inicializace – zarovnání vstupu, příprava interních datových struktur

- iterace kompresní funkce – iterativní aplikace kompresní funkce na bloky zprávy, akumulace historie výpočtu
- dokončení – konstrukce finálního výsledku z akumulovaných vnitřních stavů

Inicializace

nejprve se doplní zpráva do násobku 512: doplní se 1, potom tolik nul, aby zpráva byla o 64 kratší než nejbližší vyšší násobek 512 a na konec do 64 bitů zapsaná původní délka zprávy

Inicializace řetězících proměnných:

H_1 = 0x67452301
H_2 = 0xefcdab89
H_3= 0x98bacdf
H_4 = 0x10325476
H_5= 0xc3d2e1f0

Iterace

Kompresní funkce užívá následující interní funkce:
$\boldsymbol{f}(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$
$\boldsymbol{g}(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$
$\boldsymbol{h}(X, Y, Z) = (X \otimes Y \otimes Z)$

Zde $\wedge$ znamená bitové AND, $\vee$ je bitové OR a $\otimes$ bitové XOR.

Každý 512-bitový blok X zprávy (16 32-bitových bloků plaintextu) je zpracován následujícím algoritmem:

$X_j \leftarrow p_j$ pro $j \in 0, \ldots, 15$
for j= 16; j ≤ 79; j++)
$X_j \leftarrow (X_{j-3} \otimes X_{j-8} \otimes X_{j-14} \otimes X_{j-16}) <<< 1;$
$(A, B, C, D, E) \leftarrow (H_1, H_2, H_3, H_4, H_5);$
for (j= 0; j ≤ 19; j++) {
    $t \leftarrow (A <<< 5) + \boldsymbol{f}(B, C, D) + E + X_j + y_J);$
    $(A, B, C, D, E) \leftarrow (t, A, B <<< 30, C, D);$
}
for (j=20; j ≤ 39; j++) {

$t \leftarrow (A \lll 5) + \boldsymbol{h}(B, C, D) + E + X_j + y_J);$
$(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D);$
}
for (j=40; j ≤ 59; j++)\{
$t \leftarrow (A \lll 5) + \boldsymbol{g}(B, C, D) + E + X_j + y_J);$
$(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D);$
}
for (j=60; j ≤ 79; j++)\{
$t \leftarrow (A \lll 5) + \boldsymbol{f}(B, C, D) + E + X_j + y_J);$
$(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D);$
}
$(H_1, H_2, H_3, H_4, H_5) \leftarrow (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E);$

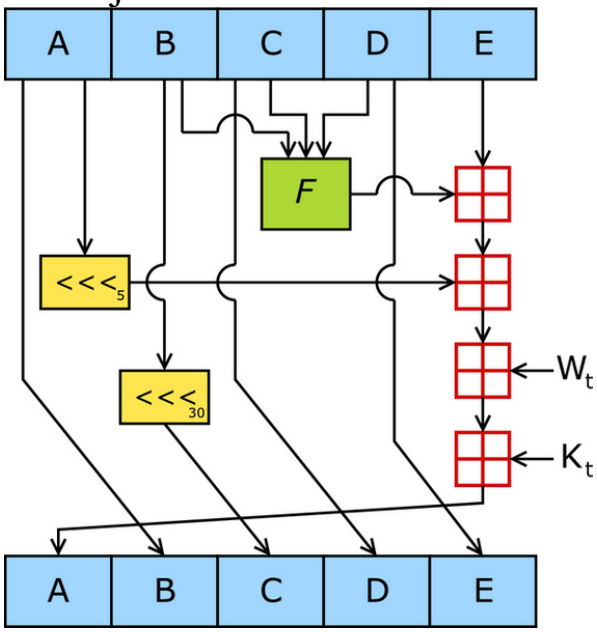
Zde x <<< y znamená rotaci x o y bitů vlevo, y_J je konstanta definovaná pro dané kolo.

Dokončení

Po zpracování posledního bloku SHA-1 vyprodukuje *digest*:

digest = $H_1 \mid H_2 \mid H_3 \mid H_4 \mid H_5$

Totéž jako obrázek:



(zde y_J odpovídá K_t, tj. elementům konstantního pole 80 položek o délce 32 bitů, W_t je vstupní blok plaintextu)

Na konci roku 2005 byl oznámen útok spočívající v nalezení **kolize** se složitostí 2^{63} operací (Wang, Yao, Yao)
Z praktického hlediska toto neznamená bezprostřední nutnost přestat SHA-1 používat, ale v nových implementacích by měl být upřednostněn některý z novějších algoritmů

SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512)

principiálně stejné algoritmy, jako SHA-1
jednotliví členové rodiny SHA-x se od sebe liší délkou slova (32, nebo 64 bitů) a počtem kol 64 u SHA-256, 80 u SHA-512
jde o nově navržené algoritmy, které by měly pokrývat dosud známe slabiny předchozí generace

Příprava konstant

h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

k(0..63) ←
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85, 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

Příprava vstupních dat

na konec dat připojit bit "1"
doplnit bity "0" do celkové délky ≡ 448 ≡ -64 (mod 512)
doplnit původní délku dat v bitech jako 64-bit big-endian integer

Zpracování zprávy po 512 bitových blocích

každý blok rozložit do šestnácti 32-bitových big-endian wordů $w(i)$, $0 \leq i \leq 15$

expandovat šestnáct 32-bit wordů do šedesátičtyř 32-bit wordů:
for i from 16 to 63 do
    $s0 \leftarrow (w(i\text{-}15) >>> 7) \oplus (w(i\text{-}15) >>> 18) \oplus (w(i\text{-}15) >> 3)$
    $s1 \leftarrow (w(i\text{-}2) >>> 17) \oplus (w(i\text{-}2) >>> 19) \oplus (w(i\text{-}2) >> 10)$
    $w(\text{i}) \leftarrow w(i\text{-}16) + s_0 + w(i\text{-}7) + s_1$
enddo

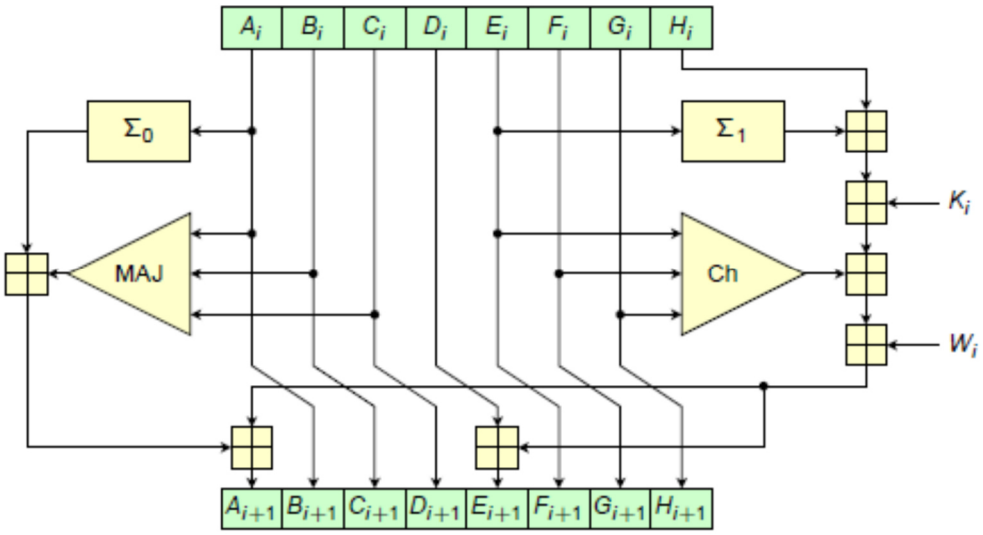
Inicializace
$a \leftarrow h_0$; $b \leftarrow h_1$; $c \leftarrow h_2$; $d \leftarrow h_3$; $e \leftarrow h_4$; $f \leftarrow h_5$; $g \leftarrow h_6$; $h \leftarrow h_7$

Hlavní smyčka
for *i* from 0 to 63 do
    $s_0 \leftarrow (a >>> 2) \oplus (a >>> 13) \oplus (a >>> 22)$
    $maj \leftarrow (a \wedge b) \oplus (b \wedge c) \oplus (c \wedge a)$
    $t_0 \leftarrow s_0 + maj$
    $s_1 \leftarrow (e >>> 6) \oplus (e >>> 11) \oplus (e >>> 25)$
    $ch \leftarrow (e \wedge f) \oplus ((\neg e) \wedge g)$
    $t_1 \leftarrow h + s_1 + ch + k(\text{i}) + w(\text{i})$
enddo

$h \leftarrow g$; $g \leftarrow f$; $f \leftarrow e$; $e \leftarrow d + t_1$; $d \leftarrow c$; $c \leftarrow b$; $b \leftarrow a$; $a \leftarrow t_0 + t_1$

Přidání výsledku bloku do stavu výpočtu
$h_0 \leftarrow h_0 + a$
$h_1 \leftarrow h_1 + b$
$h_2 \leftarrow h_2 + c$
$h_3 \leftarrow h_3 + d$
$h_4 \leftarrow h_4 + e$
$h_5 \leftarrow h_5 + f$
$h_6 \leftarrow h_6 + g$
$h_7 \leftarrow h_7 + h$

Konstrukce výsledku:
$digest \leftarrow h_0 \mid h_1 \mid h_2 \mid h_3 \mid h_4 \mid h_5 \mid h_6 \mid h_7$

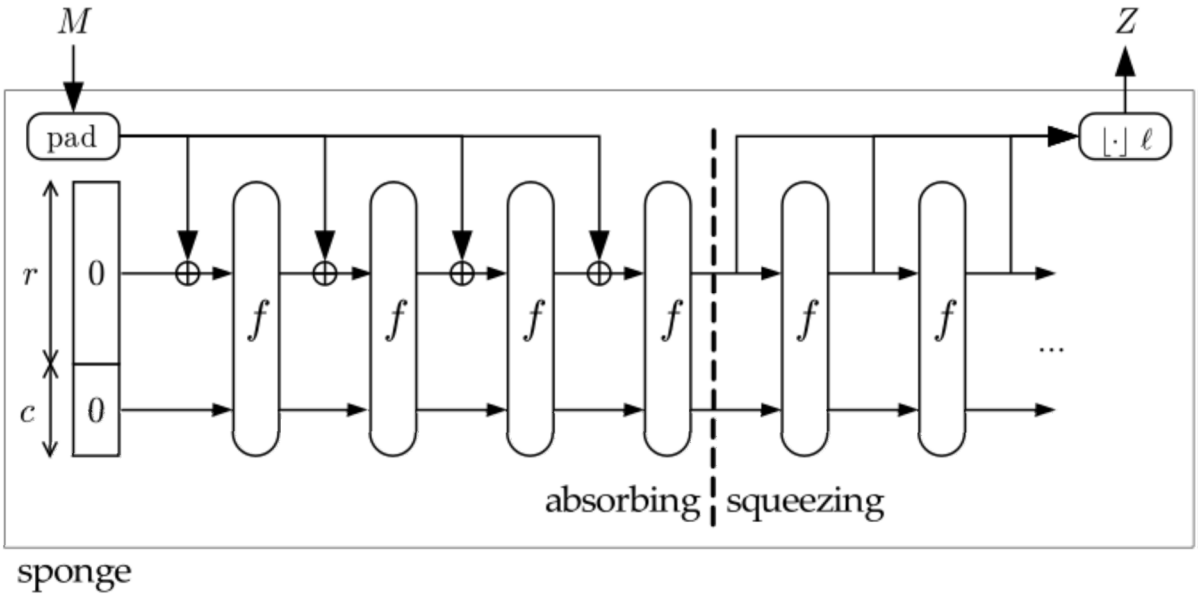


Keccak (SHA-3)

hashovací funkce založená na SPONGE struktuře
hlavní funkcí je permutace využívající bitové operace XOR, AND, NOT a rotace
vítěz SHA-3 výzvy, standardizován 2012

Sponge struktura

umožňuje akumulovat vstup a následně na základě stavu generovat výstup
stav je bitové pole o délce $r + c$
r – bitrate
c - kapacita

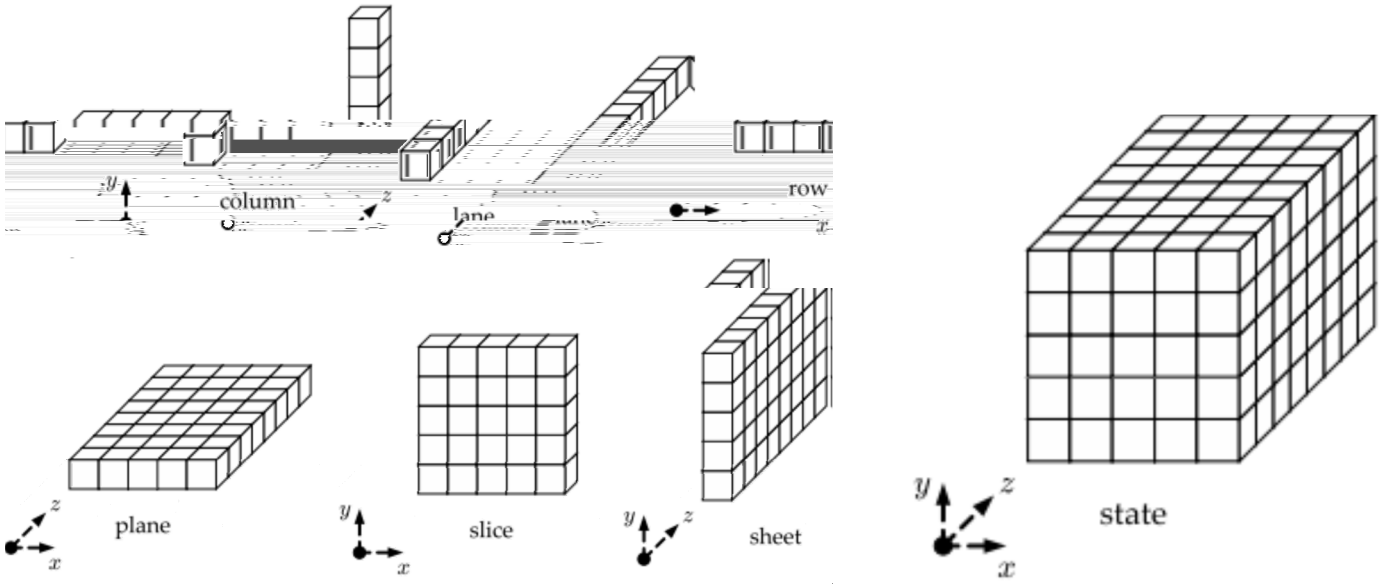


sponge

absorbce – *r*-bitový vstup je XORován s vnitřním stavem, následuje *f* transformace
mačkání – ze sturktury je načteno *r* bitů výstupu, následuje *f* transformace

Keccak$_f$

stav *S* definován jako 5 x 5 x 2^l bitů



f transformace:
// -step difuze na úrovni bitů
for {0,..,4}: (1)
C[x] = A[x,0] **xor** A[x,1] **xor** A[x,2] **xor** A[x,3] **xor** A[x,4]

for {0,..,4}: (2)
D[x] = C[x-1] **xor rot**(C[x+1], 1)

for (,) {{0,..,4} {0,..,4}}: (3)
A[x,y] = A[x,y] **xor** D[x]

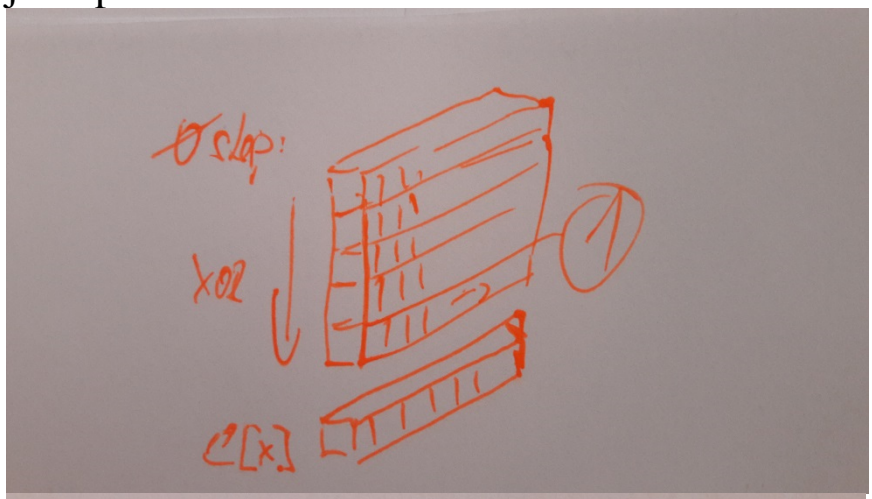
// -step difuze mezi řezy stavu
// -step zrušení *x,y* zarovnání pomocí přeházení linek
for (,) {{0,..,4} {0,..,4}}:
B[y, 2x + 3y] = **rot**(A[x,y], r[x,y]) (rotace linky a přesun jinam)
… zde r[x,y] je konstantní tabulka přesunů

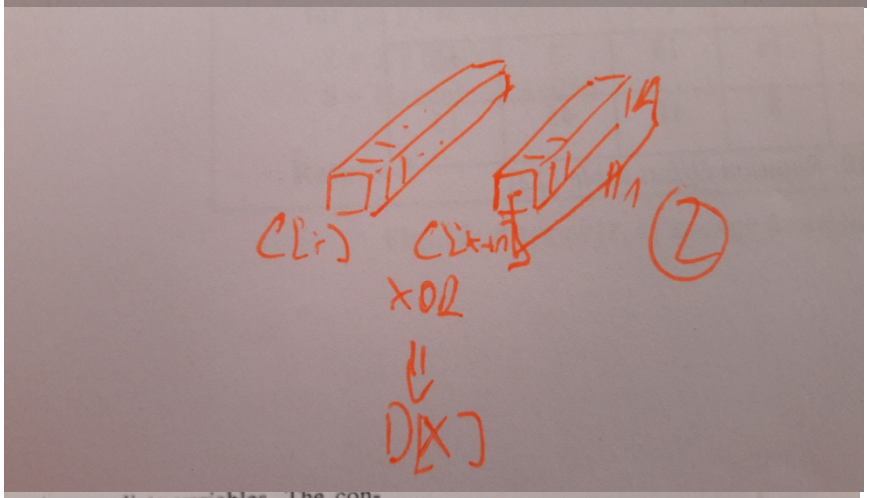
// -step nelineární mapování
for (,) {{0,..,4} {0,..,4}}: (míchání se sousedy v plane)

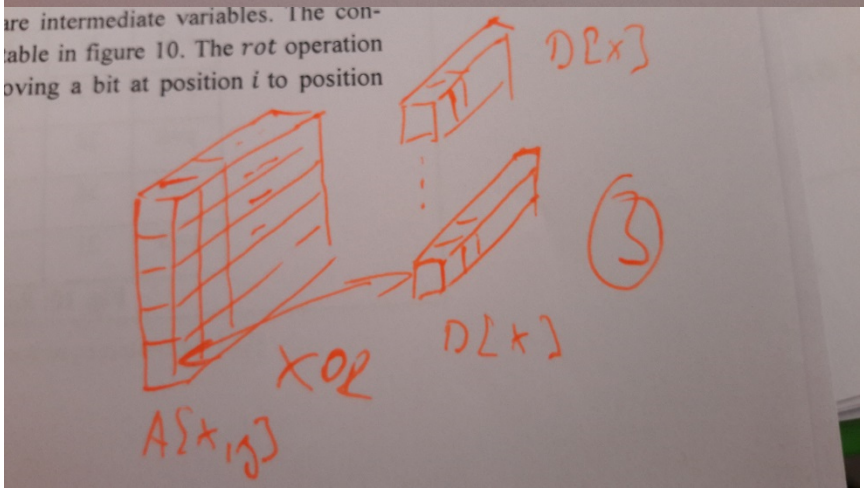
A[x,y] = B[x,y] **xor** ((**not** B[x+1,y]) **and** B[x+2,y])

// -step zrušení symetrie
A[0,0] = A[0,0] **xor** RC

ještě pár obrázků







Návrhy MAC funkcí

obecná struktura stejná jako MDC, funkce **f** a případně i *IV* závisí navíc na klíči *K*
existuje velmi malé množství algoritmů

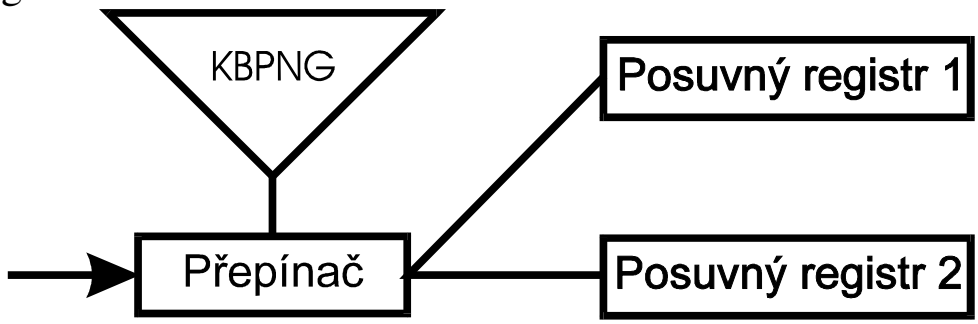
nejčastěji používanou metodou je počítání AES nebo podobného algoritmu v modu CBC, resp CFB, konkrétní schemata se liší volbou případně kombinací zmíněných módů, použitím různých zarovnání …

Jinou možností je výpočet

$$\mathbf{f} = E\left(K, X_i \oplus H_{i-1}\right) \oplus X_i$$

Stream MAC

potřebujeme kryptograficky bezpečný generátor pseudonáhodných sekvencí
podle výsledku generátoru je vstupní bit přesunut do prvního nebo druhého posuvného registru se zpětnou vazbou, výsledek určen konečným obsahem posuvných registrů



HMAC

vlastně obecný návod jako zkonstruovat MAC na základě jakékoliv hashovací (MDC) funkce:

$$\mathrm{HMAC}_K(m) = h\left((K \oplus opad) \| h\left((K \oplus ipad) \| m\right)\right),$$

kde oba bloky paddingu jsou definovány jako konstanty *opad*=0x5c5c5c...5c, *ipad*=0x363636...36

Rozbor nabízených a požadovaných vlastností hašovacích funkcí pro různé aplikace:

Vlastnosti	preimage	2nd preimage	odolnost proti kolizi
MDC+ asym. podpis	ano	ano	ano

MDC+ autentický kanál		ano	ano
Uložení hesel (MDC)	ano		
MAC (neznámý klíč)	ano	ano	ano

Generátory pseudonáhodných sekvencí

všechny tyto generátory jsou **deterministickými** p-time algoritmy, které na vstupu přijmou náhodný řetězec a expandují jej do (obvykle) mnohem delší posloupnosti

Definice:
Náhodná posloupnost je taková posloupnost, kterou nelze generovat programem kratším, než je ona sama.

Definice:
Pseudonáhodnou posloupností budeme rozumět posloupnost délky n, o které žádný deterministický p-time algoritmus není schopen s pravděpodobností větší než $(1/n)+\varepsilon$ rozhodnout, zda se jedná o náhodnou posloupnost, či nikoliv.

Kongruenční generátor

Lineární $$X_i = \left(aX_{i-1} + b\right) \mathbf{mod}\, m$$

při vhodné volbě *a, b, m* (např. *m* prvočíslo) generuje neopakující se posloupnost délky *m* - *generátor s maximální délkou*

kvadratický $$X_i = \left(aX_{i-1}^2 + bX_{i-1} + c\right) \mathbf{mod}\, m$$

kubický $$X_i = \left(aX_{i-1}^3 + bX_{i-1}^2 + cX_{i-1} + d\right) \mathbf{mod}\, m$$

kongruenční generátory jsou velmi rychlé, avšak predikovatelné, byla vypracována analýza

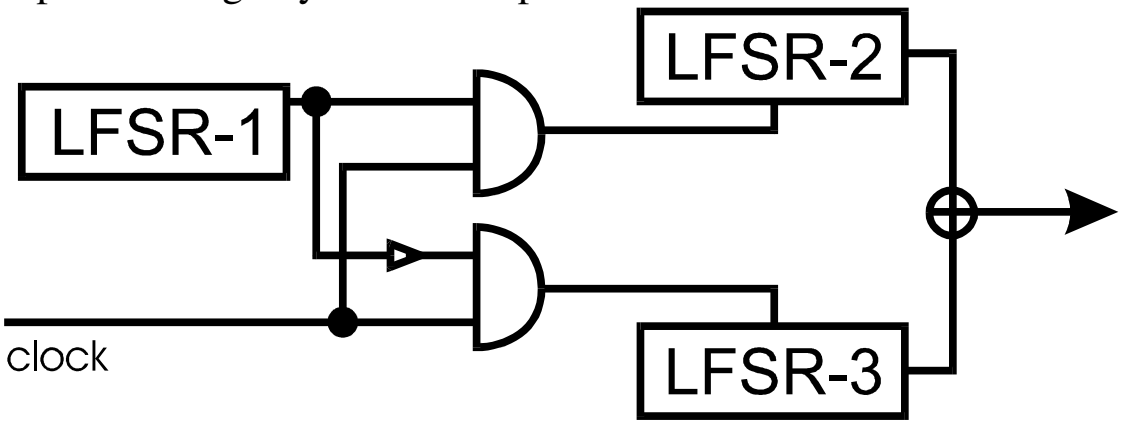
Posuvné registry s lineární zpětnou vazbou

(linear feedback shift registers)
sestávají z posuvného registru a vypouštěcí sekvence (tap sequence), což je polynom stupně max. délky posuvného registru

v každém kroku je obsah registru posunut o bit doprava, výstupem je nejpravější bit, registr se zleva doplní o XOR těch bitů v registru, které odpovídají koeficientům vypouštěcího polynomu
pro generátor s maximální délkou nutno polynom volit primitivní polynom stupně n - ireducibilní polynom dělící $x^{2^n-1}+1$, který nedělí $x^d + 1$ pro lib d dělící $2^n - 1$

Střídající stop-and-go generátor

používá tři posuvné registry s lineární zpětnou vazbou



impuls hodin jsou v závislosti na hodnotě LFSR-1 přiveden na LFSR-2 nebo LFSR-3, výstup těchto registrů je XORován, čímž vzniká výsledná hodnota.
gnerátor má velmi dlouhou periodu

Blum-Micali

$$X_i = a^{X_{i-1}} \bmod p$$

a, p prvočísla, bezpečnost plyne z obtížnosti výpočtu diskrétního logaritmu

RSA generátor

$$X_i = X_{i-1}^e \bmod n$$

výsledkem nejnižší bit X_i. Bezpečnost ekvivalentní s bezpečností RSA

Blum Blum Shub (BBS)

kde n je Blumovo číslo - součin dvou velkých prvočísel kongruentních s 3 mod 4. Generátor se inicializuje hodnotou $X_i = \left(X_{i-1}^2\right) \bmod n$ $X_0 = \left(X^2\right) \bmod n$, kde X je náhodné číslo nesoudělné s n.

Výsledkem je nejnižší bit X_i . Bezpečnost vyplývá z obtížnosti faktorizace velkých čísel. Generátor je nepredikovatelný vlevo i vpravo, ale je možné se znalostí faktorů *p, q* čísla *n* spočítat přímo libv. prvek generované posloupnosti!

$$X_i = X_0^{\left(2^i\right) \bmod \left((p-1)(q-1)\right)} \bmod n$$

Dosud není známa metoda zlomení tohoto generátoru. Nevýhodou značná pomalost. Je možné dávat na výstup log log (*n*) bitů stavu při zachování bezpečnosti.

Generátor Yarrow

uvádím jako příklad komplexní konstrukce náhodného generátoru
cílem útočníka je získat vnitřní stav generátoru, pokud uspěje, může predikovat výstup generátoru, nebo rekonstruovat dřívější části výstupní posloupnosti
kvalita generátoru závisí na kvalitě vstupu, častým problémem nesprávný odhad míri entropie zdrojů
generátor se skládá ze čtyř hlavních komponent:

- akumulátor entropie (entropy accumulator)
- mechanismus změny klíče (reseed mechanism)
- mechanismus generování (generation mechanism)
- řízení změny klíče (reseed control)

cílem je kvalitní akumulace entropie ze všech zdrojů, zajištění dostatečně častých (nealgoritmických !) změn klíče a občasná změna klíče, která bude pro útočníka nepřekonatelná i v případě, že u některého zdroje došlo k nadhodnocení obsahu entropie

Akumulátor entropie

používají se dva pooly – rychlý a pomalý
každý pool je realizován jako kontext hashovací funkce (zde SHA-1), do kterého se přimíchávají vstupy „tak jak přicházejí“

Mechanismus změny klíče

na pokyn řízení změny klíče počítá novou hodnotu klíče *K*
změna klíče na základě rychlého poolu využívá

- stávající klíč
- hash všech vstupů v rychlém poolu od poslední změny klíče

změna klíče na základě pomalého poolu využívá

- stávající klíč
- hash všech vstupů v rychlém poolu od poslední změny klíče
- hash všech vstupů v pomalém poolu od poslední změny klíče

po změně klíče jsou vynulovány čítače entropie všech dotčených poolů

Řízení změny klíče

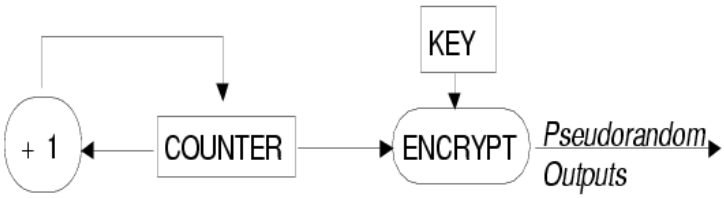
s každým zdrojem entropie je v poolu udržován čítač nashromážděných náhodných bitů

- změna klíče z rychlého poolu se provede pokud aspoň jeden čítač přesáhne práh
- změna klíče z pomalého poolu se provede pokud aspoň k čítačů přesáhne práh

práh pro rychlý pool bývá 100, pro pomalý 160, k se v závislosti na prostředí volí 2 ÷ 3

Mechanismus generování

využívá se symetrická šifra (3DES) běžící v tzv. counter módu
vstupem pro šifru je obsah čítače C a klíč K:
$C \leftarrow (C + 1) \bmod 2^n$
výstup $\leftarrow E_K\{C\}$
mechanismus generování nejpozději po $1 \leq P_g \leq 2^{n/3}$ bitech interně vygeneruje $|K|$ bitů a použije je jako nový klíč – jde o tzv. generator gate, která zabraňuje útočníkovi počítat „do minulosti"



uvedená konstrukce je možná i s použitím jiných kryptografických primitiv

Generátory náhodných posloupností

... produkují skutečně náhodné posloupnosti
… odvozené od sledování nějakého nepredikovatelného rysu skutečného děje
nutná průběžná kontrola funkčnosti generátoru (tzn. „náhodnosti" výstupu)

Generátory založené na speciálním hardware

odvíjejí svoji činnost od sledování nějakého fyzikálního jevu, který je ve své podstatě nepredikovatelný, zhusta dosti sofistikovaně „nevhodným" způsobem:

- měření záření kousku radioaktivního materiálu
- termální šum polovodiče

- měření stavu volně běžícího oscilátoru
- vnitřní šum zesilovače
- samovolné vybíjení kondenzátoru
- ...

Intel generátor

dvojice rezistorů slouží jako zdroj tepelného šumu (měření napětí)
výsledný signál zesílen a použit pro modulaci pomalu běžícího oscilátoru
výstup pomalého oscilátoru použit pro řízení samplování aktuálního stavu volně běžícího rychlého oscilátoru
následně korekce výstupu na základě dvojic výstupů měření:
01 => 1, 10 => 0, ostatní dvojice se ignorují
odhadovaná kvalita výstupu je kolem 0,999 bitů entropie na bit výstupu (za korektorem), rozumné počítat 0,5 bitu entropie na bit

na HW část generátoru zpravidla navazuje SW driver:
512 bitů stavu (16 slov)
aktualizace:

- spočítá SHA-1 hash původního stavu (5 slov, poslední slovo jde na výstup)
- přidá 32 bitů vstupu z HW (1 slovo)
- nakonec připojí úvodních 10 slov předchozího stavu

zlepšuje výsledné kryptografické vlastnosti výstupu

Generátory softwarové

jsou založeny na pozorování jevů v počítači, které jsou z hlediska programu náhodné:

- doba odezvy diskového systému
- čas mezi stisky kláves uživatelem
- pohyb myši
- systémový časovač
- stav vnitřních tabulek OS
- statistiky síťové komunikace
- ...

zásadním problémem odhad množství a rozložení entropie v generovaných datech

Testy náhodných generátorů

Monobit test

zda počet „1“ a „0“ je přibližně ‚správný’

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

zde n_0 je počet „0“, n_1 je počet „1“ ve vzorku délky n.

Serial test

zda počet výskytů „00“, „01“, „10“ a „11“ je přibližně stejný, jednotlivé podřetězce se mohou překrývat.

$$X_2 = \frac{4}{n-1}\left(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2\right) - \frac{2}{n}\left(n_0^2 + n_1^2\right) + 1$$

zde n_{xx} výskytů řetězce „xx“ ; $n_{00} + n_{01} + n_{10} + n_{11} = n - 1$

Poker test

zkoumá, zda posloupnost obsahuje zhruba stejný počet všech možných podposloupností délky m

buď m tž. $\left\lfloor \frac{n}{m} \right\rfloor \geq 5\left(2^m\right)$ a položme $k = \left\lfloor \frac{n}{m} \right\rfloor$

vstupní posloupnost rozdělíme na k nepřekrývajících se podposloupností délky m, označíme n_i počet výskytů i-tého typu podposloupnosti

$$X_3 = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k$$

Runs test

zda vstupní posloupnost obsahuje správný počet běhů, tj. posloupností samých nul (díry) a jedniček (bloky); očekávaný počet běhů délky i je $e_i = (n - i + 3)/2^{i+2}$

nechť k je nejvyšší hodnota i tž. $e_i \geq 5$.

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

Zde B_i je počet bloků, B_i počet děr délky i.

Autocorellation test

cílem je srovnání korelace mezi vstupní posloupností s a její posunutou verzí

buď d tž. $1 \le d \le \lfloor n/2 \rfloor$, počet bitů, ve kterých se s liší od posunuté verze označíme $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$

$$X_5 = 2\left(A(d) - \frac{n-d}{2} \right) \Big/ \sqrt{n-d}$$

na základě uvedených kriterií je následně možno definovat sady statistických testů, např. **FIPS140-1 test náhodnosti**:
vstupem testu je výstupní posloupnost generátoru o délce 20 000 bitů, která musí splnit všechna čtyři kriteria

1. monobit test: s požadavkem $9654 < X_1 < 10346$
2. poker test: X_3 počítané pro $m = 4$ musí vyhovovat $1{,}03 < X_4 < 57{,}4$
3. runs test: počítají se běhy délky 1 až 6 (všechny delší běhy se počítají jako by měly délku 6), pro všech 12 druhů běhů musí platit

délka	min	max
1	2267	2733
2	1079	1421
3	502	748
4	223	402
5	90	223
6	90	223

4. long run test: žádný běh delší než 34

tato sada testů musí být aplikována při každém spuštění generátoru