Funkcje jednokierunkowe (one-way functions), funkcje zapadkowe (trapdoor functions) – w poszukiwaniu nowych pomysłów na kryptografię asymetryczną i funkcje skrótu

Hubert Rydz
Krzysztof Uszko

SPIS TREŚCI

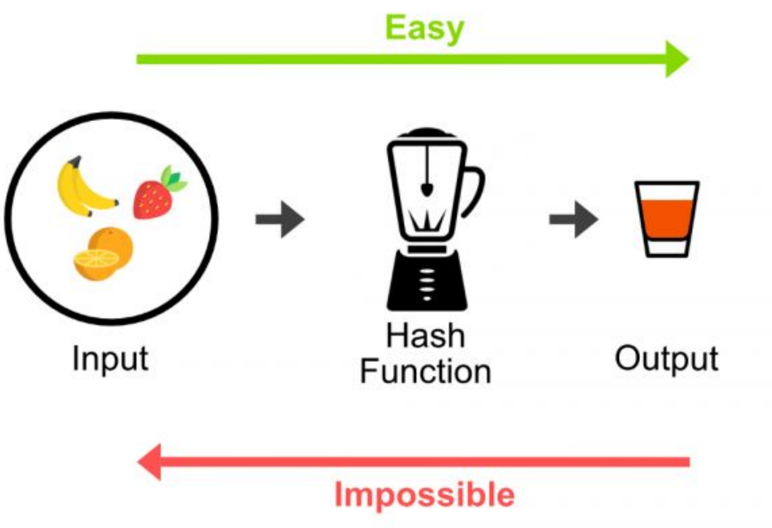
1. Definicja funkcji jednokierunkowej (one-way function)

Funkcja jednokierunkowa - funkcja działająca na dziedzinie rzeczywistej (na wejście może przyjąć dowolny argument), która jest łatwa do obliczenia, natomiast ciężka jest operacja odwrotna. Trudność w odwróceniu funkcji polega na tym że nie istnieje żaden algorytm probabilistyczny, którego złożoność obliczeniowa jest równa lub mniejsza algorytmowi wielomianowemu.

Wyróżnia się trzy główne zasady tworzenia funkcji jednokierunkowej:
- opis funkcji jednokierunkowej f jest publicznie znany, i do jej działania nie jest potrzebna żadna sekretna informacja
- dla danego x, łatwo jest wyliczyć f(x) (złożoność wielomianowa)
- dla danego y, trudno jest znaleźć takie x, że y = f(x)

Istnienie funkcji teoretycznie jednokierunkowych nie jest udowodnione. Jeżeli byłoby to prawdą, wnioskowało by to o $P \neq NP$ (jeden z głównych problemów informatyki). Z tego powodu możemy jedynie przypuszczać, że nie istnieje funkcja odwrotna.

<u>Podstawowe funkcje matematyczne, na których opierają się algorytmy:</u>
- operacja modulo i kwadrat liczby naturalnej (Funkcja Rabina) - dla liczby całkowitej x i liczb pierwszych p i q mamy $x^2 mod(pq) = y$ (więcej w punkcie: Algorytm Rabina)
- logarytmowanie dyskretne - dla liczby pierwszej p oraz liczby naturalnej x z przedziału (0,p-1) mamy $2^x mod(p) = y$ -> opiera się na tym kryptosystem ElGamal (więcej w punkcie: “Problem logarytmu dyskretnego”)
- mnożenie - głównie chodzi tu o trudność w rozkładzie dużych liczb na czynniki pierwsze. Przy pomnożeniu bardzo dużych liczb pierwszych, trudne jest wywnioskowanie ich na podstawie wyniku działania (ciekawostka: za pomocą algorytmu faktoryzacji GNFS rozłożono 193 cyfrową liczbę, co trwało 5 miesięcy)



(https://computersciencewiki.org/index.php/One-way_function)

1.0. Rodzaje:

1. Funkcje skrótu (hash) - funkcje polegająca na przekształceniu dowolnie dużego wejścia na ciąg o zawsze stałym, krótkim rozmiarze. Wyjście ma być niespecyficzne (odporne na ataki typu Known Plaintext)

2. Funkcje zapadkowe (trapdoor functions) - więcej na ich temat w dalszej części projektu

3. Generatory liczb pseudolosowych - program generujący ciąg bitów, który jest nieodróżnialny od ciągu wejściowego. Pseudolosowość jest wystarczająca w niektórych algorytmach probabilistycznych, czy np. grach komputerowych -> dokładność pseudolosowości zazwyczaj decyduje o dokładności obliczeń. W kryptografii istotne jest aby znajomość poprzednio wygenerowanych bitów nie wpływała na odgadnięcie kolejno generowanego elementu.

1.1. Historia:

(od lewej: Witfield Diffie, Martin Hellman, Michael Oser. Rabin)

Funkcje jednokierunkowe są ściśle związane z kryptografią. Z tego też powodu pierwsza potrzeba użycia nich wynikała z pracy Witfielda Diffiego oraz Martina Hellmana, W 1976 roku wydali publikację naukową dotyczącą budowania schematu podpisu cyfrowego, gdzie zawarli pierwszy opis funkcji skrótu - polegający na skróceniu dużej ilości danych do wyrażenia o stałej długości.

> *A cryptosystem which is secure against a known plaintext attack can be used to produce a one-way function.*

(źródło: New Directions in Cryptography Diffie, Hellman r. 1976)

W styczniu 1979 roku Michael Oser. Rabin zaproponował algorytm polegający na faktoryzacji.
Pierwsze definicje, analizę i konstrukcję dla funkcji jednokierunkowej można znaleźć w pracy Michael Oser. Rabin, G.Yuval, Ralph Merkle w późnych latach 70.
W 1989 Naor i Yung zdefiniowali wariant funkcji odpornych na kolizje o nazwie Universal One Way Hash Function (UOWHFs).

Następnie w 1990 Ron Rivest zaprojektował funkcje skrótu MD4, która została złamana w 1995.W 1992 Ron Rivest opracował następce MD4 zwaną MD5. W niektórych przypadkach jest ona dalej stosowana, lecz nie zaleca się używać jej w zastosowaniach wymagających odporność na kolizje, gdyż w 2004 znaleziono sposób na generowanie tych kolizji.
W 1992 roku powstał również algorytm HAVAL-256-3, który w 2004 roku również został “złamany”. W 1993 roku powstał SHA-0 lecz bardzo krótko był na rynku

ponieważ w 1995 roku powstał już następca (SHA-1) i wycofano SHA-0 ze względu na nieujawnione oficjalne wady.
Kolejne algorytmy to GOST(1994),RIPEMD-160(1996),
Tiger(1996), Panama(1998), Whirlpool(2000), SHA-256(2001), RadioGatun (2006), Skein(2008), BLAKE(2008), Grostl(2008), Keccak(SHA-3)(2008),
JH(2008), BLAKE2(2012),BLAKE3(2020)
Z powyżej wymienionych algorytmów, najbardziej znanymi są między innymi rodzina SHA, rodzina Blake. Algorytmy GOST,RIPEMD,TIGER,Panama są już niebezpiecznymi algorytmami, ponieważ znaleziono kolizje.
Algorytm Whirlpool nie jest opatentowany i jest on oddany do domeny publicznej(algorytm ten jest dostępny w programie VeraCrypt jako jeden ze wspieranych funkcji).

Bezpiecze wersje SHA-2 to SHA-512/224, SHA-512/256.
SHA-3 został stworzony, nie po to, aby wycofać swojego poprzednika. Celem SHA-3 jest to, że można bezpośrednio zastąpić SHA-2 w obecnych aplikacjach, jeśli to konieczne.

Dla algorytmu Blake, nie znaleziono jeszcze kolizji, zwłaszcza na Blake3, gdyż został on ogłoszony 9 stycznia 2020 roku. Blake3 jest znacząco szybszy niż jego poprzednicy.

1.2. Zastosowanie

```
test3:$6$D4mWKrGP$U7mjBNRVCJyTsKqg07fCt0R4N1Sj9/O8slfuenJ5Lc9DMNSI7NTKBQH6.fb7uYU.UhnZO
GDyZP3Gm0iIAXOYv1:17243:0:99999:7:::
test4:$6$waHGWFro$dI5ve0Etyf6C5jhBA4H/g2tWUwDMpYrSZpNDajbpp6H9Uol70VBs41aUmGXTfsXFkC4c5
Y0SaREdyISvNCr.s1:17243:0:99999:7:::
test5:$6$77mVd7OC$Iy5ysOTMja6dIAVROQa0DAfDPShe/3yt9Un2uK6zVuPeHQ9vTsTI13AesbB0T5ID2ARZh
G1DE3nHuHtjRDZFT.:17243:0:99999:7:::
```

(przykładowe wpisy z pliku /etc/shadow, wartość po drugim “:” to hash hasła)

- Zabezpieczenia haseł w systemach operacyjnych(linux - hashe w pliku /etc/shadow) - w celach bezpieczeństwa hasła lepiej jest przechowywać w formie ich skrótu, a nie tekstu jawnego

- Zapewnienie integralności danych (MDC-Modification Detection Code) - do wiadomości dodawany jej jest skrót pozwalający określić jej integralność (czy nie została zmodyfikowana w trakcie przesyłania)

- Podpisy cyfrowe - poprawny podpis pozwala sprawdzić czy pochodzi on od oczekiwanego nadawcy

- Bazy antywirusowe (przechowują skróty plików) - dużo łatwiej i szybciej jest przechowywać skrót pliku, i porównywać z nim podejrzane pliki

- Generowanie liczb pseudolosowych - polega na wygenerowaniu z niewielkiej ilości informacji deterministycznie ciągu bitów

- Przechowywanie informacji - jedna z bezpieczniejszych metod przechowywania danych w bazach danych - do ciągu tekstowego można dodać tzw. sól (określony ciąg bitów) a następnie wywołać na nim funkcję hashującą. Dzięki takiemu zabiegowi podczas nieuprzywilejowanego pozyskania informacji z bazy, nie będzie możliwe odgadnięcie np. hasła.

1.3. Przykłady:

-MD(Message Digest) - najbardziej popularna wersja MD5 - algorytm opracowany w 1991 roku, generuje skrót o długości 128 bitów. Razem z rozwojem technologii i mocy obliczeniowej komputerów, algorytm ten nie powinien być stosowany np. w podpisach cyfrowych. (Może jednak znaleźć swoje zastosowanie w HMAC)

-SHA(Secure Hashing Algorithm) - rodzina algorytmów zaprojektowanych przez National Security Agency. Generują odopowednio od 160 do 512 bitowe skróty

-RIPEMD - algorytm opracowany w ramach projektu Unii Europejskiej realizowanego w latach 1988 - 1992. Generuje 128 bitowy skrót , jednak powstała również wersja 160 bitowa. Zostały opublikowane wiadomości generujące ten sam skrót, jednak z racji małego zastosowania funkcji, nie została dokładnie przebadana

-HAVAL - zaproponowany w 1992 roku, generuje skróty od 128 - 256 bitowe. Istnieje możliwość wykonania od 3 do 5 rund algorytmu na blokach wiadomości. W ramach 3 przejść jest o 60% szybszy niż MD5, 5 - równie szybki jak MD5. W 2004 roku opublikowano kolizje tego algorytmu

-ELGAMAL - algorytm używany głównie w kryptografi asymetrycznej, jednak może zostać użyty jako funkcja skrótu. Jest on oparty na problemie logarytmu dyskretnego (szczegółowo opisany w Dodatkowych Informacjach) w ciele modulo p - gdzie p jest dużą liczbą pierwszą.

-BLAKE - Jedna z nowszych “rodzin” algorytmów hashujących. Powstał w celu zastąpienia złamanych i wolnych już algorytmów MD5 i SHA-1. Przełomową wersją algorytmu była wersja BLAKE-2b - szybsza od SHA-2/3 na popularnej architekturze procesora, oraz zapewniający bezpieczeństwo porównywalne z algorytmem SHA-3.Najnowsza wersja BLAKE-3 (opublikowana w styczniu 2020r) jest prawie 14x szybsza od również 256bitowego algorytmu SHA-256 oraz zapewniająca bezpieczeństwo większe niż znane do tej pory algorytmy (nie jest to jednak potwierdzone)

-Keccak (SHA-3) - Zwycięzca konkursu na nowy algorytm hashujący. Ma architekturę “gąbki” - bity wiadomości wejściowej są stopniowo dodawane do algorytmu, następnie mieszane z dużym rejestrem stanu, po czym wynik powstaje przez stopniowe “wyciskanie gąbki”

Algorytmów hashujących jest znacznie więcej - jednak ze względu na bezpieczeństwo stosuje się tylko te, które zostały już zbadane i są odporne na ataki kryptoanalityczne.

1.4. Implementacje

MD2:

Mamy b-bitową wiadomosc jako wejście.
Wiadomość jest rozszerzana do długości, która jest wielokrotnością 16, nawet jeśli wiadomość początkowo ma takie rozmiary to i tak jest dodawane 16 bajtow.

Dodawana jest informacja ile bajtów zostało dołączone do wiadomości.
Informacja ta składa się z 16-N%16 liczb które mają wartość 16-N%16
W następnym kroku jest używana 256-bajtowa “losowa” permutacja skonstruowana z cyfr liczby pi, która wygląda następująco

```
S = [
41, 46, 67, 201, 162, 216, 124, 1, 61, 54, 84, 161, 236, 240, 6, 19,
98, 167, 5, 243, 192, 199, 115, 140, 152, 147, 43, 217, 188, 76, 130, 202,
30, 155, 87, 60, 253, 212, 224, 22, 103, 66, 111, 24, 138, 23, 229, 18,
190, 78, 196, 214, 218, 158, 222, 73, 160, 251, 245, 142, 187, 47, 238, 122,
169, 104, 121, 145, 21, 178, 7, 63, 148, 194, 16, 137, 11, 34, 95, 33,
128, 127, 93, 154, 90, 144, 50, 39, 53, 62, 204, 231, 191, 247, 151, 3,
255, 25, 48, 179, 72, 165, 181, 209, 215, 94, 146, 42, 172, 86, 170, 198,
79, 184, 56, 210, 150, 164, 125, 182, 118, 252, 107, 226, 156, 116, 4, 241,
69, 157, 112, 89, 100, 113, 135, 32, 134, 91, 207, 101, 230, 45, 168, 2,
27, 96, 37, 173, 174, 176, 185, 246, 28, 70, 97, 105, 52, 64, 126, 15,
85, 71, 163, 35, 221, 81, 175, 58, 195, 92, 249, 206, 186, 197, 234, 38,
44, 83, 13, 110, 133, 40, 132, 9, 211, 223, 205, 244, 65, 129, 77, 82,
106, 220, 55, 200, 108, 193, 171, 250, 36, 225, 123, 8, 12, 189, 177, 74,
120, 136, 149, 139, 227, 99, 232, 109, 233, 203, 213, 254, 59, 0, 29, 57,
242, 239, 183, 14, 102, 88, 208, 228, 166, 119, 114, 248, 235, 117, 75, 10,
49, 68, 80, 180, 143, 237, 31, 26, 219, 153, 141, 51, 159, 17, 131, 20]
```

Kroki pokazujące jak dodawana jest suma kontrolna

```
/* Clear checksum. */
    For i = 0 to 15 do:
      Set C[i] to 0.
    end /* of loop on i */

    Set L to 0.

    /* Process each 16-word block. */
    For i = 0 to N/16-1 do

      /* Checksum block i. */
      For j = 0 to 15 do
        Set c to M[i*16+j].
        Set C[j] to S[c xor L].
        Set L to C[j].
       end /* of loop on j */
     end /* of loop on i */
```

Suma kontrolna jest obliczana w sposób:

krok 1:
x=xor(ascii(1 elementu^L)) na początku L=0
Następnie bierzemy S[x] i robimy xor(S[x] i S[x(ale z poprzedniej 16 bajtowej kolumny)])
I tak przechodzimy przez całą wiadomość by w końcu dostać 16 bajtowa sumę kontrolną która dodajemy na koniec wiadomości
Po inicjalizacji bufera MD2 przechodzimy do kroków szyfrujących:

Kroki szyfrujące

```
/* Process each 16-word block. */
  For i = 0 to N'/16-1 do

     /* Copy block i into X. */
     For j = 0 to 15 do
       Set X[16+j] to M[i*16+j].
       Set X[32+j] to (X[16+j] xor X[j]).
      end /* of loop on j */

     Set t to 0.

     /* Do 18 rounds. */
     For j = 0 to 17 do

       /* Round j. */
       For k = 0 to 47 do
         Set t and X[k] to (X[k] xor S[t]).
       end /* of loop on k */

       Set t to (t+j) modulo 256.
     end /* of loop on j */

  end /* of loop on i */
```

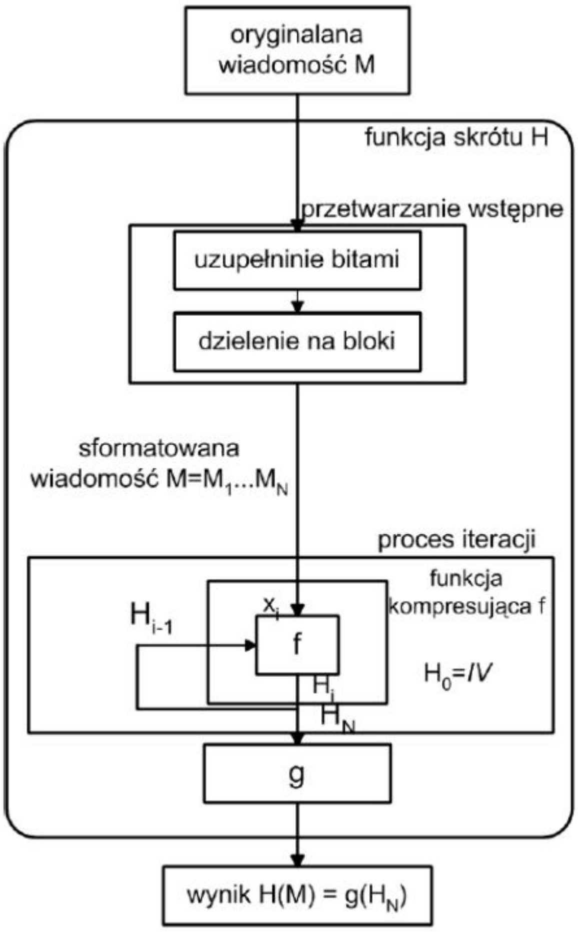
Żeby lepiej zwizualizować sobie kroki zrobione tutaj, to podzielmy 48 bufor na 3 części md_bufor_0, md_bufor_1, md_bufor_2:

```
md_bufor_0=16*0
md_bufor_1=blok na którym obecnie pracujemy
md_bufor_2=xor(md_bufor_1, md_bufor_0)
```

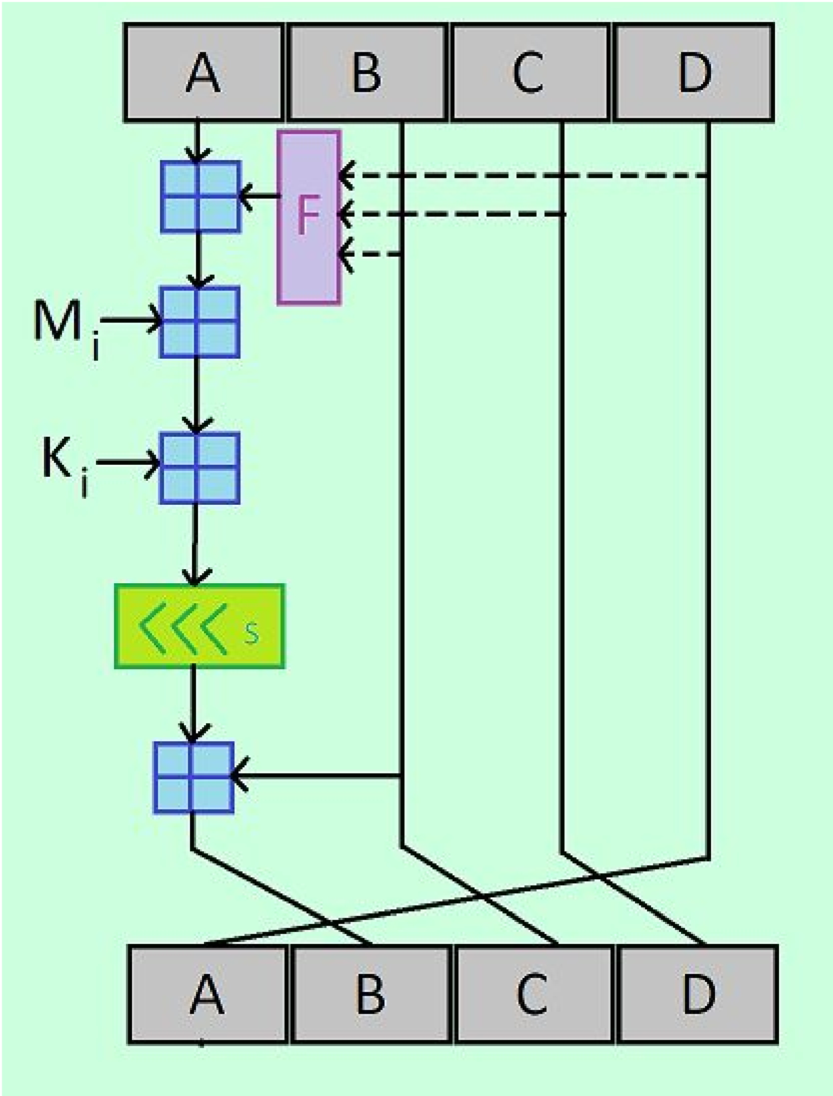
I ta iteracja jest wykonywana n/16-1 razy:

```
suma=0
for j in range(18)
	for i in range(48):
		suma=md_bufor(i)^S[suma]
		md_bufor[i]=suma
	suma=(suma+j)%256
```

Czyli występuje xor pomiędzy całym buforem, a odpowiednikiem sumy w tablicy znaków pi, jak łatwo się domyślić pierwsze 16 bajtów to nasz hash
Następnie zamieniamy na wartość heksadecymalną i tak oto otrzymujemy skrót.



MD5



Schemat jednej rundy algorytmu md5 (F-> odpowiednia funkcja dla obecnej rundy, M_i-> 32 bitowy blok wiadomości, K_i -> 32 bitowa stała, ⊞ -> operacja dodawania modulo 2^{32}, <<<s -> rotacja o s bitów)

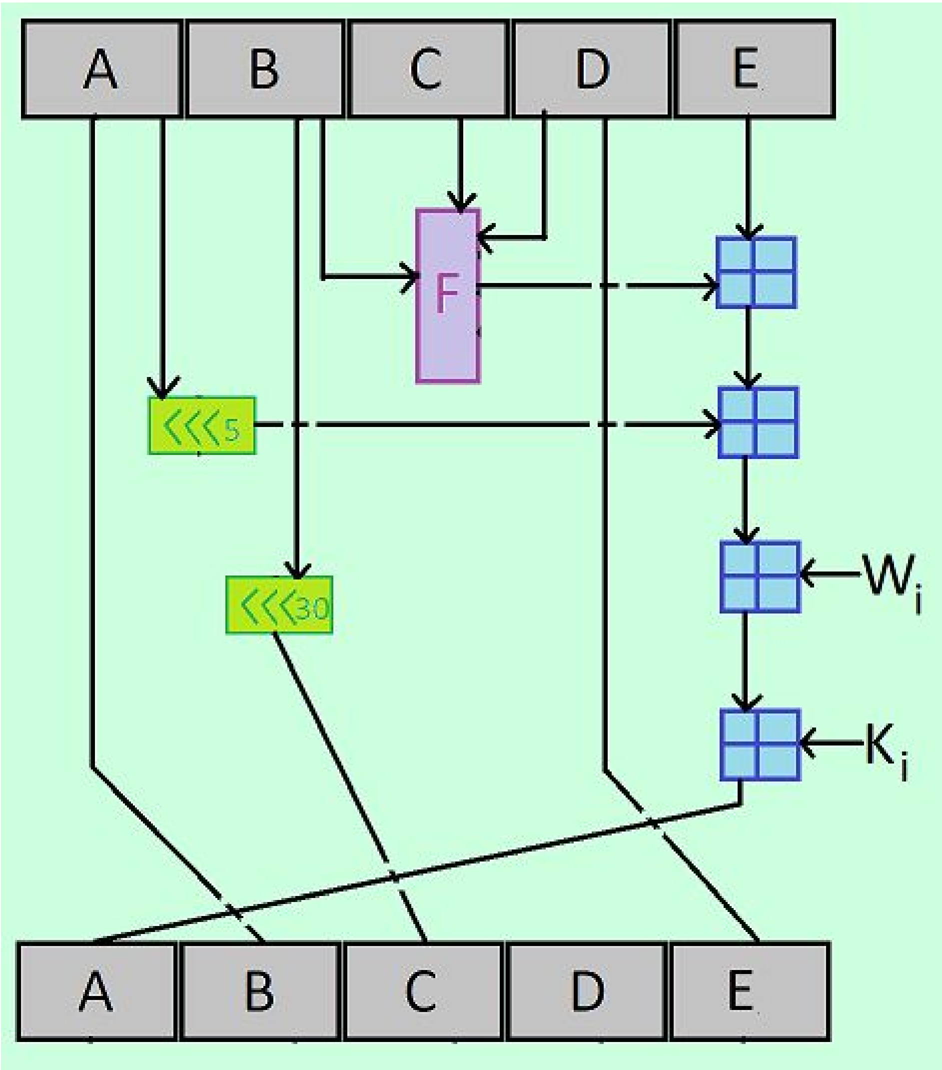
Kroki algorytmu:

1. Zdefiniowane są 2 tablice s[64] i K[64] o stałych wartościach.
2. Do wiadomości wejściowej doklejamy bit o wartości 1
3. Dopełniamy wejście zerami do bloków 512 bitowych, i ostatniego - 448 bitowego
4. Ustawiamy stan początkowy: 0123456789abcdeffedcba9876543210
5. Wykonujemy 64 rundy funkcji (schemat na obrazku powyżej)-> w zależności od numery rundy używamy następujących funkcji (na schemacie oznaczone jako ‘F’):
   - od 1 do 16: $F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$
   - od 17 do 32: $G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$

- od 33 do 48: $H(X,Y,Z) = X \oplus Y \oplus Z$
- od 49 do 64: $I(X,Y,Z) = Y \oplus (X \vee \neg Z)$

6. Zwracamy stan po ostatnim przejściu jako skrót wiadomości

SHA1



Schemat jednej rundy algorytmu SHA1 (F-> odpowiednia funkcja dla obecnej rundy, W_i->rozszerzone słowo dla rundy, K_i -> stała rundy, ⊞ -> operacja dodawania modulo 2^{32}, <<< -> rotacja odpowiednio o 5 i 30 bitów)

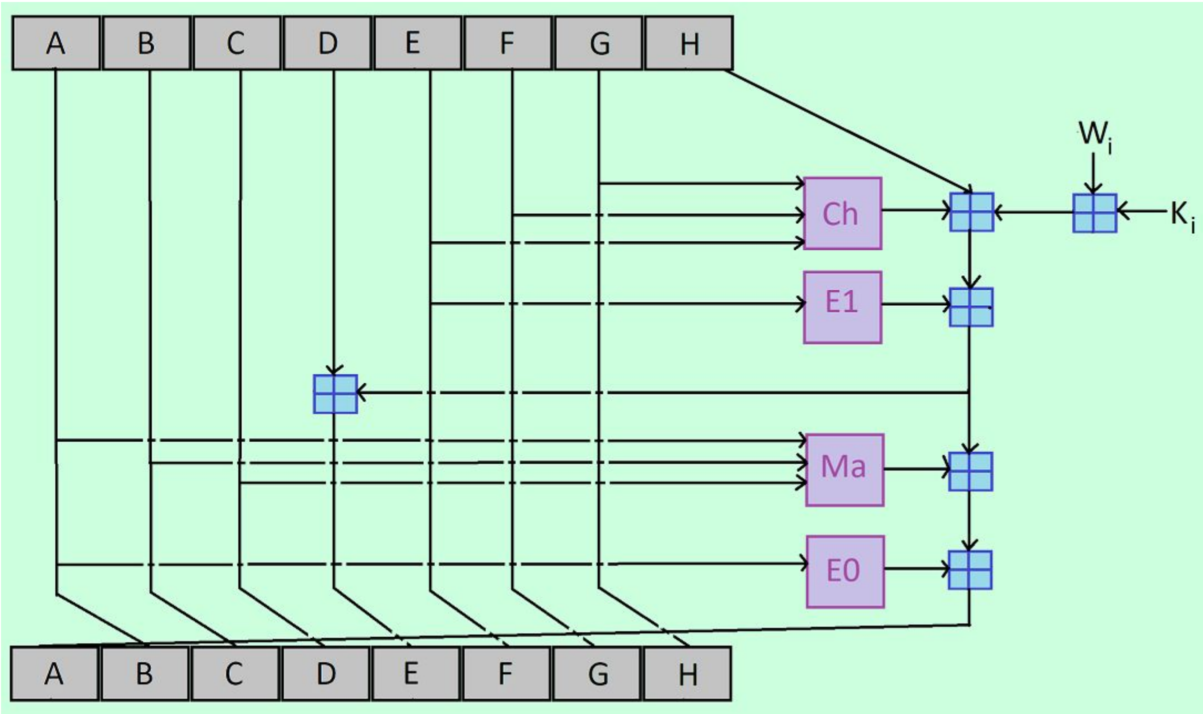
Wiadomość jest uzupełniania w taki sposób by ostatni blok wiadomości miał 448 bitów. Jest to realizowane w taki sposób, że dodawana jest 1 i odpowiednia ilość 0. Następnie do ostatniego bloki jest dodawana 64 bitowa liczba (dlugość wiadomości), która sprawia,że wiadomosc jest wielokrotnościa 512 bitów.Po dopełnieniu jest inicjalizowany 5 zmiennych
A=0x67452301

B=0xefcdab89
C=0x98badcfe
D=0x10325476
E=0xc3d2e1f0
Następnie funkcja pracuje na 512 bitowych blokach i wartości A,B,C,D,E są kopiowane do a,b,c,d,e
Blok 512 bitowy jest dzielony na 16 32-bitowe bloki a następnie rozszerzany do 80 32-bitowych bloków za pomocą petli
for j from 16 to 79
w[j]=(w[j-3] xor w[j-8] xor w[j-14] xor w[j-16]) leftrotate 1
Każdy blok jest szyfrowany w 4 cyklach po 20 operacji
Zainicjowana jest pętla od 0 do 79
jeżeli i ∈<0,19>
f=(b and c) or((Not b) and d)
k=0x5A827999
jeżeli i ∈<20,39>
f=b xor c xor d
k=0x6ED9EBA1
jeżeli i ∈<40,59>
f=(b and c) or (b and d) or (c and d)
k=0x8F1BBCDC
jeżeli i ∈<60,79>
f=b xor c xor d
k=0xCA62C1D6
temp=(a leftrotate 5) + f +e +k + w[i]
e=d
d=c
c=b leftrotate 30
b=a
a=temp

Po skończonej pętli do zmiennych A-E dodawane są wartości:
A=A+a
B=B+b
C=C+c
D=D+d
E=E+e
Hash funkcji jest ustalany poprzez zmienne A-E

Hash=(A leftshift 128) or (B leftshift 96) or (C leftshift 64) or (D leftshift 32) or E

SHA2



Schemat jednej rundy algorytmu SHA2 (W_i->rozszerzone słowo dla rundy, K_i -> stała rundy, $\boxplus$ -> operacja dodawania modulo 2^{32})

Odpowiednie funkcje dla schematu podanego powyżej:

$$\mathrm{Ch}(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G)$$
$$\mathrm{Ma}(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$$
$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$
$$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$$

Uwaga - podane wartości są obliczone dla algorytmu SHA-256, dla algorytmu SHA-512 należy przyjąć inne liczby!

Kroki algorytmu (wartości dla wersji SHA-256):

1. Zdefiniowane są dwie tablice h[8] - pierwsze 8 liczb pierwszych jako 32 bitowe słowa, k[64]- pierwsze 32 bity części ułamkowej pierwszych 64 liczb pierwszych
2. Do wiadomości wejściowej doklejamy bit o wartości 1
3. Dopełniamy wejście zerami do bloków 512 bitowych, i ostatniego - 448 bitowego, dołączamy długość wiadomości jako 64 bitową liczbę całkowitą
4. Każdy 512 bitowy blok dzielimy na 16 32-bitowe porcje.
5. Rozszerzamy 16 32-bitowe słowa na 64 32-bitowe słowa

6. Dla każdego 512 bitowego bloku wykonujemy 64 rundy algorytmu (schemat na obrazku powyżej), gdzie początkowe A-H odpowiadają wartościom tablicy h[]
7. Na koniec łączymy wartości nowej tablicy h[] i otrzymujemy skrót wiadomości

BLAKE 3

Kroki algorytmu:

1. Wejście jest dzielone na 1 KiB części, a następnie jest “formatowane” w formę drzewa binarnego. Każda z części jest kompresowana (hash-owana - jednak twórcy BLAKE’a używają formy ‘compress’) oddzielnie.

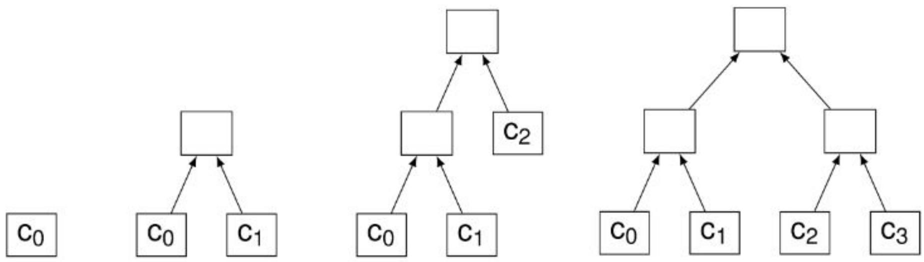


Figure 1: Example tree structures, from 1 to 4 chunks.

2. Każda 1 KiB część jest dzielona na bloki po 64B -> 512b (ostatni blok jeżeli nie jest ‘pełny’ jest uzupełniany zerami). Każdy z bloków jest następnie parsowany na 16 32-bitowe słowa (m_0 - m_{15})
3. Na każdym 32 bitowym słowie jest wykonywana funkcja kompresująca, która przyjmuje następujące wartości:
    - wartości wiążące (input chaining value) - $h_0...h_7$ (256b)
    - blok wiadomości (message block) - $m_0...m_{15}$ (512b)
    - 64 bitowy licznik t
    - liczba bajtów wejściowych bloku b (32b)
    - flaga d (32b)

    Dla każdego “liścia” struktury drzewa:

- pierwszy liść po lewej stronie na którym zaczyna działać funkcja przyjmuje początkowe $h_o...h_7 = m_0...m_7$, natomiast po prawej: $h_o...h_7 = m_8...m_{15}$.
- Licznik t dla węzła rodzica (“parent node”) przyjmuje zawsze 0
- Liczba bajtów b dla węzła rodzica przyjmuje zawsze 64
- Dla pierwszego bloku przyjmowana jest flaga CHUNK_START, ostatniego - CHUNK_END, dla węzła rodzica - PARENT

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ IV_0 & IV_1 & IV_2 & IV_3 \\ t_0 & t_1 & b & d \end{pmatrix}$$

Na początek inicjowanych jest 16 wartości v_0...v_{15} odpowiadających wcześniej opisanym danym, oraz gdzie IV_0...IV_3 są stałymi dla algorytmu BLAKE, oraz d jest jedną z flag:

IV_0	0x6a09e667
IV_1	0xbb67ae85
IV_2	0x3c6ef372
IV_3	0xa54ff53a
IV_4	0x510e527f
IV_5	0x9b05688c
IV_6	0x1f83d9ab
IV_7	0x5be0cd19

Flag name	Value
CHUNK_START	2^0
CHUNK_END	2^1
PARENT	2^2
ROOT	2^3
KEYED_HASH	2^4
DERIVE_KEY_CONTEXT	2^5
DERIVE_KEY_MATERIAL	2^6

Następnie wykonywanych jest 7 rund, po 8 operacji każda

$$\begin{array}{llll} G_0(v_0, v_4, v_8, v_{12}) & G_1(v_1, v_5, v_9, v_{13}) & G_2(v_2, v_6, v_{10}, v_{14}) & G_3(v_3, v_7, v_{11}, v_{15}) \\ G_4(v_0, v_5, v_{10}, v_{15}) & G_5(v_1, v_6, v_{11}, v_{12}) & G_6(v_2, v_7, v_8, v_{13}) & G_7(v_3, v_4, v_9, v_{14})\,. \end{array}$$

Operacje wykonywane są kolejno na każdej z kolumn bloku stanu początkowego, oraz każdej z przekątnych.

Funkcja G(a,b,c,d) ma następującą postać:

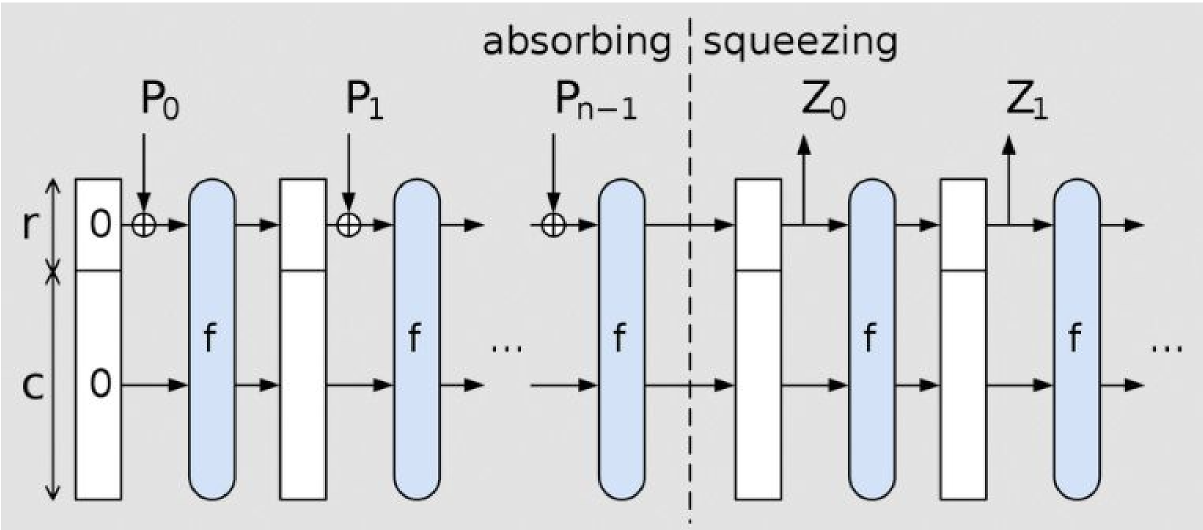
$$\begin{aligned} a &\leftarrow a + b + m_{2i+0} \\ d &\leftarrow (d \oplus a) \ggg 16 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \ggg 12 \\ a &\leftarrow a + b + m_{2i+1} \\ d &\leftarrow (d \oplus a) \ggg 8 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \ggg 7 \end{aligned}$$

gdzie: $\oplus$ - XOR, + - dodawanie modulo 2^{32}, >>> rotacja bitowa “w prawo” oraz m_x to x-owe słowo wiadomości.
Po każdej z rund 32 bitowe słowa wiadomości są permutowane wg. odpowiedniej kolejności:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	6	3	10	7	0	4	13	1	11	12	5	9	14	15	8

4. Skrót wiadomości to wynik ($v_0 ... v_{15}$) przejścia algorytmu przez wszystkie 1 KiB części (dla przypomnienia: każda z części jest dzielona na bloki, a bloki na 32 bitowe słowa) od najdalszych "liści" drzewa binarnego do korzenia.

Keccak (SHA-3)



(źródło: https://en.wikipedia.org/wiki/SHA-3)

Kroki algorytmu:

1. Do wiadomości o długości N dodajemy bity (sekwencja 101010..), tak aby całkowita długość była podzielna przez r - "długość stanu" (=25 w domyślnej implementacji) .
2. Dzielimy wiadomość na bloki - $P_0 ... P_{n-1}$
3. Inicjalizujemy stan S=[] -> tablica (domyślnie o rozmiarze 5x5x64 (64 to rozmiar bloku)) wypełniona zerami.
4. Absorpcja wejścia:
    - każdy blok rozszerzamy ciągiem c bitów zerowych do uzyskania bloku o długości b (w standardowej implementacji b = 5*5*64)
    - S=f(XOR(P_i,S) - gdzie f jest funkcją przekształcającą (opisane dalej)
5. Inicjujemy Z jako pusty ciąg, Z=""
6. Jeżeli (len(Z) < d = określona długość wyjścia skrótu (224-512 b)):
    - dodajmy pierwsze r bitów ze stanu S do Z
    - jeżeli len(z) jest dalej mniejsze niż d, aplikujemy f(S)

7. Obcinamy Z do d bitów.

Funkcja przekształcająca:

Niech a będzie tablicą 3 wymiarową - a[i][j][k] = a[5][5][64] w domyślnej implementacji.
Funkcja f składa się z 12 + 2l (gdzie rozmiar bloku = 2^l) rund 5 mniejszych funkcji:

θ (theta)
$a[i][j][k] \leftarrow a[i][j][k] \oplus \text{parity}(a[0...4][j-1][k]) \oplus \text{parity}(a[0...4][j+1][k-1])$,
gdzie parity to funkcja parzystości - w ciągu sprawdzamy liczbę jedynek - jeżeli jest parzysta dodajemy bit 0 na koniec wiadomości.

ρ (rho)
Rotujemy bitowo każde z 25 słów przez liczbę trójkątną (0,1,3,6,10,15...)

π (pi)
Wykonujemy permutację: $a[3i+2j][i] \leftarrow a[i][j]$

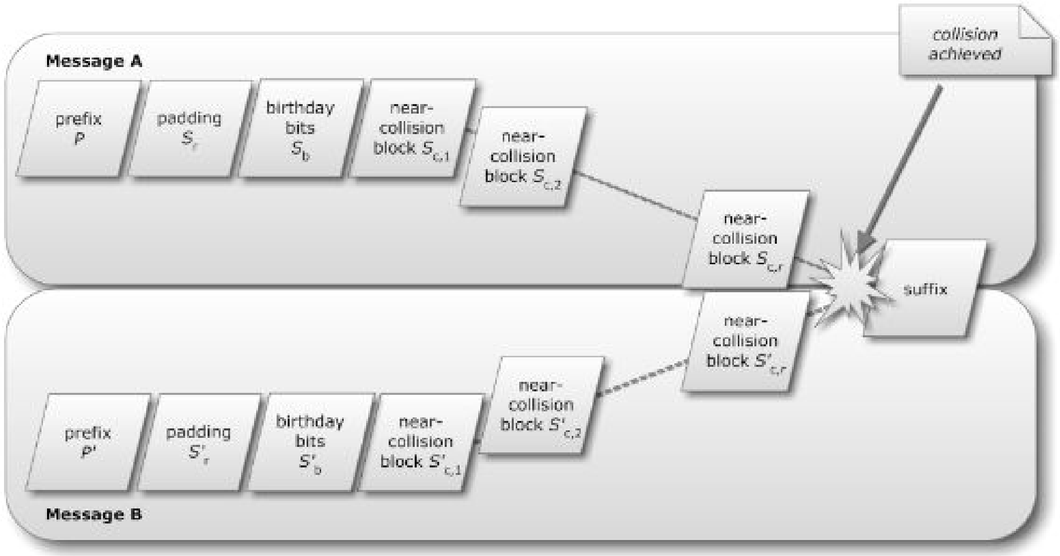
χ (chi)
$a[i][j][k] \leftarrow a[i][j][k] \oplus (\neg a[i][j+1][k] \,\&\, a[i][j+2][k]$

ι (iota)
$a[0][0][2^m-1] \oplus \text{LSFR}[m+7n]$, gdzie LSFR - to Rejestr przesuwający z liniowym sprzężeniem zwrotnym stopnia 8

1.5. Typy i próby ataków:

a) **Collision attack** - atak polegający na znalezieniu dwóch takich samych hashy dla różnych wiadomości. Do tej pory udało się tak "złamać" algorytmy MD5 i SHA-1
   - MD5 - w marcu 2013 udowodniono, że można znaleźć kolizję w czasie 2^{18} prób - taki proces trwa sekundy na domowym komputerze
   - SHA-1 - (160 bitowy) algorytm w założeniu miał gwarantować bezpieczeństwo rzędu 2^{80} . W styczniu 2020 roku autorzy ataku przedstawili dokumentację, która pokazuje że są w stanie znaleźć kolizję przy $2^{61.2}$ próbach

b) **Chosen prefix collision attack** - dla dwóch prefixów p1 i p2, należy znaleźć takie dodatkowe wiadomości m1 i m2 że h(p1 || m1) = h(p2 || m2) , gdzie || oznacza operację konkatenacji.



   - MD5 - atak zajmuje 2^{39} prób (dla przeciętnego komputera to sprawa kilku godzin) , gdzie twórcy algorytmu założyli że wymagane jest 2^{64} takich prób
   - SHA-1 - ci sami autorzy co w przypadku zwykłego collision attack na ten algorytm, opublikowali dokument wskazujący liczbę $2^{63.4}$ jako wystarczającą na znalezienie w.w kolizji.

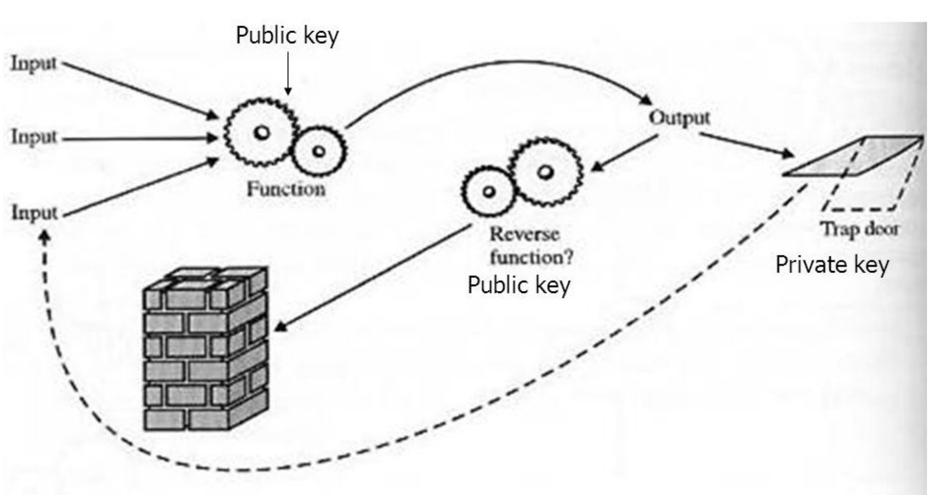
c) **Preimage attack** - polega na znalezieniu wiadomości, która ma zadaną wartość hash. Wyróżnia się 2 rodzaje "odporności" algorytmu na tego typu ataki.
   1. *preimage resistance* - niemożliwym jest do obliczenia (z punktu widzenia ograniczonych zasobów) wiadomości, której hash odpowiada danemu (mając y, nie obliczymy x => h(x) = y)
   2. *second-preimage resistance* - znając jedną wiadmość x=> h(x)=y, znalezienie drugiej takiej wiadomości x'=> h(x')=y jest niewykonalne
   - najlepszy atak na algorytm MD5 (128 bitowy) wymagał $2^{123.4}$ prób

- poniższa tabela pokazuje złożoności wymagane do ataków tego typu na algorytm BLAKE

Version	Rounds	free-start collision	free-start (2nd) preimage	collision	(2nd) preimage
BLAKE-28	1.5 rounds	2^{80}	2^{160}	2^{80}	2^{160}
	2 rounds	2^{96}	2^{192}	-	2^{209}
	2.5 rounds	2^{96}	2^{192}	-	2^{209}
BLAKE-32	1.5 rounds	2^{96}	2^{192}	2^{96}	2^{192}
	2 rounds	2^{112}	2^{224}	-	2^{241}
	2.5 rounds	2^{112}	2^{224}	-	2^{241}
BLAKE-48	1.5 rounds	2^{128}	2^{256}	2^{128}	2^{256}
	2 rounds	2^{160}	2^{320}	-	2^{355}
	2.5 rounds	2^{160}	2^{320}	-	2^{355}
BLAKE-64	1.5 rounds	2^{192}	2^{384}	2^{192}	2^{384}
	2 rounds	2^{224}	2^{448}	-	2^{481}
	2.5 rounds	2^{224}	2^{448}	-	2^{481}

2. Definicja funkcji zapadkowej (trapdoor function)

Funkcja zapadkowa (trapdoor function) - funkcja łatwa do wykonania w jedną stronę, natomiast do efektywnego przeprowadzenia operacji odwrotnej wymaga użycia sekretnej informacji. Analogia do zapadni (trapdoor) polega na tym, że łatwo jest spaść, natomiast ciężko jest się wydostać - chyba że ma się drabinę (sekretna informacja).



Funkcje hashu nie są funkcjami zapadkowymi - ponieważ w ich implementacjach nie uwzględniono mechanizmu polegającego na odwróceniu funkcji za pomocą specjalnej informacji.

Historia funkcji zapadkowych pokrywa się z historią funkcji jednokierunkowych (Punkt 1.1) -> dokładnie podczas ustalania protokołu Diffiego - Hellmana. Tam po raz pierwszy pojawiła się definicja funkcji trapdoor -> jednak znalezienie takowej było znacznie trudniejsze niż się początkowo wydawało. Jedną z pierwszych propozycji było użycie problemu sumy podzbioru (link) - jednak jak się szybko okazało - sugestia była niewłaściwa.

messages into output cryptograms. As such a public key system is really a set of *trap-door one-way functions.* These are functions which are not really one-way in that simply computed inverses exist. But given an algorithm for the forward function it is computationally infeasible to find a simply computed inverse. Only through knowledge of certain *trap-door information* (e.g., the random bit string which produced the *E-D* pair) can one easily find the easily computed inverse.

(Pierwsza wzmianka o funkcjach zapadkowych - źródło: New Directions in Cryptography Diffie, Hellman r. 1976)

Do tej pory najbardziej właściwymi funkcjami spełniającymi wyżej wymienione role są algorytmy RSA oraz Rabina (o algorytmie Rabina więcej w Dodatkowych Informacjach). Obie z nich są związane z faktoryzacją dużych liczb pierwszych.

Funkcje związane z problemem logarytmu dyskretnego nie są uważane za funkcje zapadkowe - nie ma w niej “zapadni” umożliwiającej łatwe obliczanie logarytmów.

Dodatkowe informacje:

1. **Problem logarytmu dyskretnego:**
   Logarytmem dyskretnym elementu **b** przy podstawie **a** (określonych w danej grupie skończonej) jest liczba całkowita **c**, dla której zachodzi następująca równość:
   $$a^c = b$$
   w kryptografii problem tego zagadnienia polega właśnie na znalezieniu takiej liczby. Logarytm dyskretny jest niejednoznaczny -> w jednym ciele możemy znaleźć wiele logarytmu dyskretnych elementu **b** przy podstawie **a.**
   Najprostszą metodą znajdywania logarytmu (jednak również najbardziej złożoną obliczeniowo) jest potęgowanie wszystkich **a** przez wszystkie możliwe **c** z danej grupy.
   Najszybsza obecnie znana metoda obliczania logarytmu dyskretnego (sito ciała liczbowego) ma złożoność czasową:
   $$e^{c \cdot \log_2^{\frac{1}{3}}(p) \cdot \log_2^{\frac{2}{3}}(\log_2(p))}$$
   , gdzie c- const.
   Najbardziej optymalna metoda obliczania logarytmu (redukcja Pohliga-Hellmana) polega na redukcji problemu do podobnych podproblemów grup niższego rzędu.
   Żadna ze znanych metod nie posiada złożoności wielomianowej względem bitów wejścia.

   Problem logarytmu dyskretnego w kryptografii możemy spotkać w zagadnieniach t.j: protokół Diffiego-Hellmana, algorytmie ElGamal czy kryptografii krzywych eliptycznych.

2. **Algorytm Rabina:**
   1. Wybieramy dwie duże liczby pierwsze **p** i **q** , takie że: p = 3 mod 4 i q = 3 mod 4
   n = pq => **n** jest kluczem publicznym, natomiast para (**p**,**q**) jest kluczem prywatnym

   Wiadomośc **M** jest szyfrowana: najpierw następuje konwersja wiadomości do liczby **m**, gdzie **m**<**n**, następnie **c** = m^2 mod **n**, gdzie **c** jest szyfrogramem

   W celu odszyfrowania wiadomości:
   1. obliczamy:

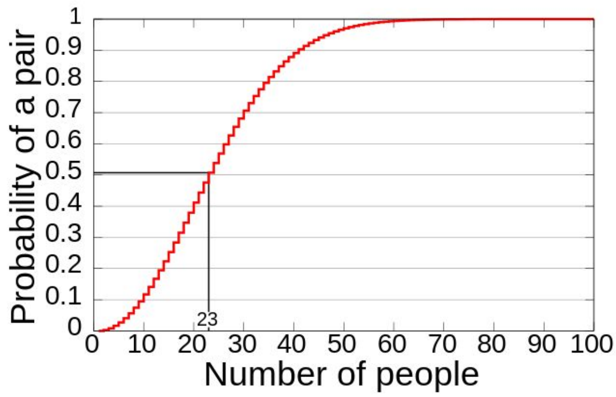
$$m_p = c^{\frac{1}{4}(p+1)} \bmod p$$
$$m_q = c^{\frac{1}{4}(q+1)} \bmod q$$

2. następnie korzystając z algorytmu Euklidesa wyznaczamy takie y_p i y_q , że $y_p * p + y_q * q = 1$
3. Korzystając z Chińskiego twierdzenia o resztach obliczamy:

$$r_1 = (y_p \cdot p \cdot m_q + y_q \cdot q \cdot m_p) \bmod n$$
$$r_2 = n - r_1$$
$$r_3 = (y_p \cdot p \cdot m_q - y_q \cdot q \cdot m_p) \bmod n$$
$$r_4 = n - r_3$$

,gdzie któraś reszta jest poszukiwaną przez nas wiadomością **m**.

3. Paradoks dnia urodzin:

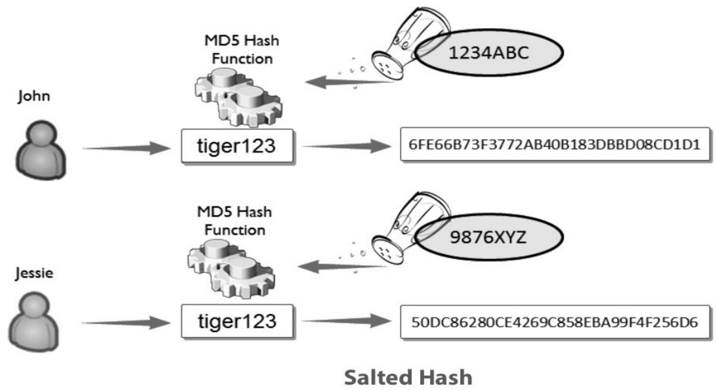


(wykres ilustrujący poszukiwane prawdopodobieństwo w zależności od liczby osób)

Paradoks związany z obliczaniem prawdopodobieństwa tego samego dnia urodzin. Problem polega na znalezieniu minimalnej liczby osób, tak aby prawdopodobieństwo wystąpienia pary, która ma urodziny tego samego dnia była większa bądź równa 50%. Takich osób wystarczy zaledwie 23. Natomiast ok. 60 osób daje nam już 99% prawdopodobieństwo wystąpienia takiego zdarzenia.

W kryptografii problem ten jest podstawą ataku urodzinowego. Atak ten sugeruje znalezienie kolizji znacznie szybciej niż sugerowałby rozmiar przeciwdziedziny funkcji haszującej. Np. dla 128 bitowego algorytmu MD5 wystarczy wygenerować ok. $1{,}1774 * 2^{64}$ skrótów, aby mieć 50% prawdopodobieństwo że znaleźliśmy kolizję.

4. Sól:



(źródło:https://forum.huawei.com/enterprise/en/what-is-salting-cyber-security-awareness/thread/492569-867)

Losowy ciąg bitów dodawany do hasła podczas obliczania funkcji skrótu. Ma to na celu zabezpieczenie bazy danych przed tzw. atakiem słownikowym - atak polegający na znajomości hashy popularnych słów używanych jako haseł - następnie porównywany jest skrót hasła wraz ze skrótem w słowniku.

Sól wykorzystuje się także aby celowo “wydłużyć” rozmiar hasła bez wymuszania tego na użytkowniku.
Podczas weryfikacji hasła dodajemy sól w tym samym miejscu co przy pierwszej generacji skrótu i porównujemy otrzymane wyniki.

Bibliografia:

http://journals.bg.agh.edu.pl/AUTOMAT/2016.20.2/automat.2016.20.2.39.pdf
http://wazniak.mimuw.edu.pl/images/2/20/Bsi_05_wykl.pdf

https://www.amw.gdynia.pl/images/AMW/Menu-zakladki/Nauka/Zeszyty_naukowe/Numery_archiwalne/2013/ZN_2013_2/11_Rodwald%20P.pdf
http://hydrus.et.put.poznan.pl/~remlein/wykl_08.pdf
https://www.mimuw.edu.pl/~alx/bsk/wyklad6.pdf

http://marc-stevens.nl/research/papers/StLdW%20-%20Chosen-Prefix%20Collisions%20for%20MD5%20and%20Applications.pdf
https://eprint.iacr.org/2009/238.pdf

https://ee.stanford.edu/~hellman/publications/24.pdf
https://github.com/BLAKE3-team/BLAKE3

https://en.wikipedia.org/wiki/Linear-feedback_shift_register
https://en.wikipedia.org/wiki/Parity_(mathematics)#Additional_applications
https://en.wikipedia.org/wiki/SHA-3

http://professor.unisinos.br/linds/teoinfo/Keccak.pdf

https://crypto.stackexchange.com/questions/62790/md5-chosen-prefix-collision-attack
https://eprint.iacr.org/2019/459.pdf