RIPEMD-160
Алгоритам за хеширање

Стефан Теофиловски
Јуни, 2023

1. Вовед

Во полето на криптографијата и безбедноста на податоци, хеш функциите имаат клучна улога во обезбедување на интегритет и автентичност на информациите кои се пренесуваат. Хеш функциите се математички алгоритми (чекори) кои за порака со произволна големина враќаат хеш или шифрирана низа на знаци со фиксна големина. Хешот е уникатен за секој влез и дури мала промена во влезот, резултира со сосема различен излез.

Во периодот на дигиталната криптографија се измислени и имплементирани многу хеш алгоритми како “SHA”, “MD”, “Blake”, “RIPEMD”. Ова се само дел од повеќе фамилии на хеш алгоритми. Секоја фамилија има повеќе верзии и секоја нова нуди повеќе сигурност на податоците и е поотпорна на нападите врз своите претходници.

Во овој документ, ќе се задржиме на RIPEMD, а поконкретно ќе зборуваме за RIPEMD-160. Ќе се запознаеме со неговата историја, кој е мотивот за неговиот развој како и самата имплементација и концептите на хеш алгоритмот и неговата употреба во некои апликации како Биткоин.

2. Историја

RIPEMD-160 или *RACE Integrity Primitives Evaluation Message Digest-160* е криптографска хеш функција направена како подобрување на RIPEMD. RIPEMD алгоритмот е создаден од неколку истражувачи, вклучувајќи ги Hans Dobbertin, Antoon Bosselaers и Bart Preneel, во почетокот на 1990-тите години. Алгоритмот бил доста успешен и ги заменил дотогашните најкористени алгоиртми како MD5 и SHA-1.

RIPEMD за прв пат е објавен во 1996, а подоцна се направени и неколку негови верзии како RIPEMD-128, RIPEMD-160, RIPEMD-256 и RIPEMD-320. Секоја верзија се разликува во должината на хешот и овозможува на корисниците да одберат верзија според своите потреби. RIPEMD-160 со својот излез од 160 бита станува најкористен бидејќи корисниците наоѓаат некоја средина помеѓу ефикасност, брзина и безбедност.

По објавувањето на 160 битната верзија на RIPEMD, алгоритмот бил изложен на многу тестови и напади од криптоаналитичари но не биле пронајдени некои позначајни недостатоци. Сепак, со самиот развој на технологијата, било неопходно да се развијат и некои нови посилни хеш функции како што се денес SHA-3 и Blake2.

3. Мотивација

Со објавувањето на MD4 хеш алгоритмот, Роналд Ривест не запознава со нови принципи на дизајн на криптографски алгоритми. Новиот дизајн вклучува пополнување на пораката со дополнителни битови, итеративно процесирање, пресметување во рунди, зависно поврзани променливи и слично. Секогаш нови принципи носат и неочекувани недостатоци. Нападите на A. Bosselaers и B. Boer го покажале токму тоа. Иако не претставувале позначајна закана на алгоритмот во тоа време, сепак тие покажале некои нови криптоаналитичарски техники кои подоцна ќе бидат проширени и ќе има успешни напади врз овој алгоритам.

Ова доведува до потребата за развој на поусовршени хеш алгоритми како RIPEMD и MD5. Во 1995, H. Dobbertin успева да најде колизии во хеширањето на MD4, а за тоа му биле потребни само неколку секунди на персонален компјутер. Ова јасно го отстранува MD4 од безбеден за користење. Со мало проширување на техниките во нападот, успешно се најдени и колизии за MD5 и RIPEMD.

Главната причина за развој на RIPEMD-160 верзијата се нападите на груба сила. Во тоа време ваквите напади биле скапи и барале скапа опрема, но со оглед на Муровиот закон за намалување на цената на компјутерските сметачи за 4 пати на секој 3 години, можело да се заклучи дека алгоритмите RIPEMD и MD5 нема да бидат сигурни во следните неколку години. Поради ова, има потреба од зголемување на излезот на хешот и било претставена посилна верзија на RIPEMD, односно RIPEMD-160 кој се користи и 20-30 години подоцна.

4. Имплементација

Имплементацијата и дизајнот на алгоритмот е сличен со MD4 алгоритмот. Разликата е во тоа што сега излезот на хешот е 160 бита и бројот на рунди е зголемен на 5, а бројот на операции кои се извршуваат на 80. Алгоритмот ќе го поделиме во неколку чекори:

1. Додавање битови на пораката (Message Padding)

RIPEMD-160 работи со 512-битни блокови од пораката, па затоа доколку должината на пораката во битови не е делива со 512 (или има помалку битови од 512) мора да додадеме битови.

Прво додаваме ‘1’ бит на крајот од пораката, проследен со ‘0’ битови се додека не се исполнува условот:

- if size%512==448

Зошто 448, а не 0? На крајот од пораката оставаме место за 64 бита во кој ја запишуваме должината на оригиналната порака претставена во 64 бита. Со ова може да дефинираме дека на крајот треба да се задоволува условот:

- size + padded_bits + 64 % 512 = 0

Со ова сме сигурни дека пораката може да ја поделиме на блокови од 512 бита и да продолжиме кон следниот чекор.

```python
# Message padding
def message_padding(message):
    bits = ''.join(format(byte, '08b') for byte in message)

    size = len(bits)

    if len(bits) % 512 != 448:
        bits += '1'

        while len(bits) % 512 != 448:
            bits += '0'

    bits += format(size, '064b')

    return bits
```

Слика 1. Message padding in python

2. Иницијализација

Хешот во RIPEMD-160 се состои од бафер од пет 32-битни збора (вкупно 160 бита). Овие зборови се означуваат како A, B, C, D, E и имаат предефинирана иницијална вредност, и тоа:

A	0x67452301
B	0xEFCDAB89
C	0x98BADCFE
D	0x10325476
E	0xC3D2E1F0

3. Булиеви функции и рунди

Алгоритмот работи во 5 рунди и за секоја рунда се користи различна функција за пресметување:

Рунда 1	$f1(x, y, z) = x \oplus y \oplus z$
Рудна 2	$f2(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$
Рунда 3	$f3(x, y, z) = (x \vee \neg y) \oplus z$
Рунда 4	$f4(x, y, z) = (x \wedge z) \vee (y \wedge \neg z)$
Рунда 5	$f5(x, y, z) = x \oplus (y \vee \neg z)$

4. Пермутации

Секој блок од 512 бита повторно го делиме на 16 блока од по 32 бита. Ќе ги означиме со M[0..15]. На добиените блокови прво се врши ρ пермутација дадена во следнава табела:

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

Значи, добиваме M[0..15] = M[ρ(0)..ρ(15)].

По ова дефинираме пермутација π(i) = 9i + 5 (mod 16). Оваа пермутација дополнително го меша распоредот на 32-битните зборови по секоја операција.

Овие пермутации се направени за да ги задоволат својствата на дифузија и конфузија што се клучни во безбедноста на хеш алгоритмите. Придонесуваат во тоа да слични влезни пораки да не дадат сличен хеш резултат.

5. Поместувања

Секоја операција во алгоритмот содржи поместување (shift) за ‘S’ позиции во лево. S е константа и е дадена со следнава табела:

Рунда	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}
1	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
2	12	13	11	15	6	9	9	7	12	15	11	13	7	8	7	7
3	13	15	14	11	7	7	6	8	13	14	13	12	5	5	6	9
4	14	11	12	14	8	6	5	5	14	12	15	14	9	9	8	6
5	15	12	13	13	9	5	8	6	15	11	12	11	8	6	5	5

6. Константи

Секоја операција во алгоритмот исто така содржи и константа K дадена со следнава табела:

Рунда 1	0x00000000	0x5A827999	0x6ED9EBA1	0x8F1BBCDC	0xA953FD4E
Рунда 2	0x50A28BE6	0x5C4DD124	0x6D703EF3	0x7A6D76E9	0x00000000
Рунда 3	0x5A827999	0x6ED9EBA1	0x8F1BBCDC	0xA953FD4E	0x50A28BE6
Рунда 4	0x6ED9EBA1	0x8F1BBCDC	0xA953FD4E	0x50A28BE6	0x5C4DD124
Рунда 5	0x8F1BBCDC	0xA953FD4E	0x50A28BE6	0x5C4DD124	0x6D703EF3

Константите се внимателно одбрани за да задоволат некои криптографски својства. Добиени се од специфични bit-patterns и некои децимални вредности од корени. Придонесуваат за подобра дифузија и конфузија на алгоритмот.

7. Операции

Откако ја добиваме пораката, прво ја извршуваме функцијата message_padding за додавање на битови. Потоа ја делиме пораката на блокови од 512 битови. Секој 512-битен блок го делиме на 16 збора од по 32 бита.
Пред да почнеме со операциите, за секој од 16-те блока M[0..15] извршуваме пермутација ρ.

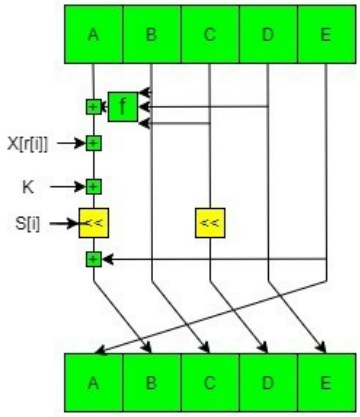
За секој 32 битен збор извршуваме 5 рунди (сите блокови прво ја извршуваат првата рунда, па сите втората итн..). 16 блока по 5 рунди даваат вкупно 80 операции. Една операција изгледа вака:

- $A := (A + f(B, C, D) + M_{\pi} + K)^{<<S} + E$
  $C := C^{<<10}$

каде:
- A, B, C, D, E се зборовите од баферот на хешот кој го иницијализиравме во чекор 2;
- F е функција која зависи од рундата и се дефинира според табелата во чекор 3;
- M е 32 битен збор, и при секој пристап ја користиме пермутацијата π дефинирана во чекор 4;
- S е константа која ја земаме од табелата дефинирана во чекор 5;
- K е константа чија вредност ја земаме од табелата дефинирана во чекор 6;

Пресметувањето го завршуваме така што секој збор од баферот му ја менуваме позицијата за едно место, односно:

- A := E
  B := A
  C := B
  D := C
  E := D



Слика 2. Процес на вршење една операција

Кога ќе завршат сите 80 операции за еден 512-битен збор, на иницијалните бафери на кои ќе им ги доделиме следните имиња: AA, BB, CC, DD, EE, им ги додаваме 32-битните зборови (A, B, C, D, E) добиени од 80-те операции извршени претходно.

- AA := AA + A
  BB := BB + B
  CC := CC + C
  DD := DD + D
  EE := EE + E

Сето ова го повторуваме за секој 512-битен блок.

```python
def ripemd_160(message):

    # Initialize word buffer
    A = 0x67452301
    B = 0xEFCDAB89
    C = 0x98BADCFE
    D = 0x10325476
    E = 0xC3D2E1F0

    padded_message = message_padding(message)

    # Devide the message in 512bit blocks
    blocks = [padded_message[i:i+512] for i in range(0, len(padded_message), 512)]

    for block in blocks:
        words = [block[i:i+32] for i in range(0, len(block), 32)] # Devide the block into 16 32-bit words
        for i in range(len(words)):
            words[i] = words[rho_permutation[i]] # Do rho permutation for each word
        a, b, c, d, e = A, B, C, D, E

        # 80 operations for each 512-bit block (16 words times 5 rounds)
        for i in range(0, 80):
            r=-1
            # Define the round function for each word
            if i<=16:
                fn = round_one(b,c,d)
                r = 0
            elif i<=32:
                fn = round_two(b,c,d)
                r = 1
            elif i<48:
                fn = round_three(b,c,d)
                r = 2
            elif i<64:
                fn = round_four(b, c, d)
                r = 3
            else:
                fn = round_five(b, c, d)
                r = 4

            # Update the words
            tmp_e = e
            e = d
            d = (c<<10) & 0xFFFFFFFF
            c = b
            b = ((a + fn + int(words[pi_permutation(i)],16) + K[r][i%5])<<SHIFTS[r][i%15] + tmp_e) & 0xFFFFFFFF
            a = tmp_e

        # Update the word buffer
        A = (A + a) & 0xFFFFFFFF
        B = (B + b) & 0xFFFFFFFF
        C = (C + c) & 0xFFFFFFFF
        D = (D + d) & 0xFFFFFFFF
        E = (E + e) & 0xFFFFFFFF
```

Слика 3. Имплементација на RIPEMD-160 алгоритмот во python

На крајот вршиме спојување на 5те збора од баферот и го добиваме крајниот резултат (хешот) од 160 бита кој ќе има облик:

- AABBCCDDEE

5. Предности и недостатоци

Како и сите алгоритми така и RIPEMD-160 има свои поволности но и свои недостатоци. Некои од нив ќе ги споменеме во оваа точка.

Предности:

- ➔ Безбедност: RIPEMD-160 има големо ниво на безбедност. Не е откриен практичен напад врз него и е отпорен на колизии.
- ➔ Ефикасност: постои некоја средина помеѓу брзината и безбедноста што го прави алгоритмот погоден за многу апликации
- ➔ RIPEMD-160 е доста проучуван и истражуван од многу криптографи и криптоаналитичари, и претставува основа за некои принципи и нови алгоритми кои се појавуваат после него.

Недостатоци:

- ➔ Развиен е во средината на 1990-тите, што значи веќе е стар и новата технологија и зголемувањето на компјутерската моќ бара некои понови алгоритми со поголеми хешови како што е SHA-3.
- ➔ RIPEMD-160, иако бил користен, сепак не го добил вниманието на поширокатa публика како што е случајот со фамилијата на хеш функции SHA.
- ➔ Иако нема практични напади, имало многу теоретски напади кои претставуваат потенцијални закани за алгоритмот.

6. Каде се користи?

RIPEMD-160 е користен во многу области од криптографијата како дигитални потписи, хеширање лозинки, проверување на интегритетот на пораки и датотеки (да се осигураме дека не биле изменети) и слично.

Можеби најзначајната употреба ја има во „blockchain” технологиите, а поконкретно кај Bitcoin-от. Подетално, RIPEMD-160 се користи за генерирање на адреси на корисниците. Ќе објасниме накратко како кога корисник креира дигитален паричник на Bitcoin мрежата, му се доделува уникатна адреса.

1. Bitcoin користи ECDSA (Ecliptic curve digital signature algorithm) за креирање на клуч со 130 карактери.

2. Овој клуч потоа се хешира со SHA-256, со што големината му се намалува на 32 карактери (бајти).

3. За дополнително намалување на големината на клучот, се применува токму RIPEMD-160. Резултатот од SHA-256 се хешира повторно со RIPEMD-160 и се намалува големината на 20 бајти со цел да се овозможи полесна употреба и запишување.

4. Овој добиен резултат двојно се хешира, повторно со SHA-256. Од хешот се земаат првите 4 бајти и се додаваат на крај на клучот добиен со RIPEMD-160 како “checksum”.

* checksum е дел од целиот податок и се користи за проверување на грешки и интегритет.

5. Со помош на функцијата Base-58 која служи за претворање на бинарен број во текст читлив за човекот (избегнувајќи слични знаци како O и 0 или I и l) и мрежниот префикс се добива крајниот резултат, односно приватниот клуч за адресата на корисникот.

* мрежен префикс – се користи за да се одреди дали адресата служи за тест мрежи или вистински биткоин мрежи.

Треба да напоменеме дека во поновите верзии на Bitcoin, во последните години (2022-сега), веќе не се користат овие алгоритми и се заменуваат со посовремени алгоритми како SHA-3 и Blake2.

7. Заклучок

За крај, можеме да кажеме дека RIPEMD-160 игра голема улога во историјата на криптографијата и хеш функциите. Неговата отпорност на практични напади, го направиле погоден за многу апликации, а посебно во областа на криптовалутите. Претставува основа за градење и изучување на нови побезбедни алгоритми. Ограничен е поради неговата старост, поради зголемена компјутерска моќ, во поново време се заменува со други побезбедни хеш алгоритми како SHA-3, Blake2, Argon2, SHAKE-128.

8. Референци

- https://en.wikipedia.org/wiki/MD5
- https://en.wikipedia.org/wiki/RIPEMD
- https://www.geeksforgeeks.org/ripemd-hash-function/
- https://en.bitcoin.it/wiki/RIPEMD-160
- *Farhad Ahmed Sagar, 2016, Cryptographic Hashing Functions – MD5,* https://cs.indstate.edu/~fsagar/doc/paper.pdf
- *Hans Dobbertin, Antoon Bosselaers, Bart Preneel, 1996, RIPEMD-160: A Strengthened Version of RIPEMD,* https://homes.esat.kuleuven.be/~bosselae/ripemd160/pdf/AB-9601/AB-9601.pdf