Kryptografia i Ochrona Danych - projekt: Funkcja hashująca SHA-256

Kacper Skelnik, 291566

2021

1 Wstęp

Celem projektu było napisanie programu wykonującego funkcję skrótu SHA-256. Jest to funkcja przyporządkowująca słowu (dla SHA-256 32-bitowych) krótką wartość o stałym rozmiarze. Dla zbioru S, doskonała funkcja haszująca przyporządkowuje każdemu elementowi z tego zbioru liczbę całkowitą bez kolizji. Innymi słowy, funkcja ta jest iniekcją. Ponadto funkcja skrótu w zastosowaniach kryptograficznych powinna zachowywać się jak funkcje losowa na ile to możliwe, będąc jednocześnie funkcjami deterministycznymi, łatwymi do obliczenia. Jest tak dlatego, że nie powinno być możliwości wywnioskowania żadnych użytecznych informacji na temat wiadomości, mając do dyspozycji tylko jej skrót.

Poniżej przedstawię i omówię napisaną przeze mnie implementację funkcji SHA-256 w czystym języku python (bez użycia dodatkowych biblotek)

2 Program

Omawiany program jest w całości dostępny na GitHub'ie. Składa się z pojedynczej klasy która może być używana w innych skryptach poprzez wykonanie funkcji `def hash_message(msg)`, która przyjmuje jeden argument (msg: string) będący wiadomością do hashowania. Zasada działania procesu jest następująca:

1. Na początku wraz ze stworzeniem instancji klasy, tworzone są inicjalne wartości stałych.

```
    def __init__(self):
        self.k = [
            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,
        ]

        self.h = [
            0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
            0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19,
        ]
```

W tym przypadku:

- `self.k` - to pierwsze 32 bity ułamkowych części pierwiastka sześciennego pierwszych 64 liczb pierwszych {2, ... ,311}
- `self.h` - to pierwsze 32 bity ułamkowych części pierwiastka kwadratowego pierwszych 8 liczb pierwszych {2, ... ,19}

2. Następnie definiowane są funkcje pomocnicze

```
    def leftrotate(self, x, c):
        # left rotate the number x by c bytes
        x &= 0xFFFFFFFF
        return ((x << c) | (x >> (32 - c))) & 0xFFFFFFFF

    def rightrotate(self, x, c):
        # right rotate the number x by c bytes
        x &= 0xFFFFFFFF
        return ((x >> c) | (x << (32 - c))) & 0xFFFFFFFF

    def leftshift(self, x, c):
        # left shift the number x by c bytes
        return x << c

    def rightshift(self, x, c):
        # right shift the number x by c bytes
        return x >> c
```

zgodnie z opisami:

- `leftrotate(self, x, c)` - wykonuje obrót bitowy w lewo
- `rightrotate(self, x, c)` - obrót bitowy w prawo
- `leftshift(self, x, c)` - przesunięcie bitowe w lewo
- `rightshift(self, x, c)` - przesunięcie bitowe w prawo

3. Następnie definiowana jest funkcja wykonująca wstępną obróbkę przetwarzanej wiadomości.

```
    def process_message(self, message):
        data = bytearray(message)
        dat_len_bits = (8 * len(data)) & 0xFFFFFFFFFFFFFFFF
        # Add a single '1' bit at the end
        data.append(0x80)
        # Padding with zeros as long as the input bits length     448 (mod 512)
        while len(data) % 64 != 56:
            data.append(0)
        # Append original length in bits to message
        data += dat_len_bits.to_bytes(8, byteorder='big')

        assert len(data) % 64 == 0, "Error in padding"
        return data
```

Kolejne kroki wykonywane wewnątrz tej funkcji to:

- Zamiana wiadomości teksowej na bity
- dodanie pojedynczego bitu '1' na końcu wiadomości
- dodanie tylu zer na końcu wiadomości aż jej długość (w bitach) nie będzie wynosiła 448 (mod 512)
- dodanie na koniec przetworzonej wiadomości jej oryginalnej długość w bitach

4. Zaraz potem następuje definicja głównej funkcji wykonującej trzon logiki.

```
    def update_hash(self, data):
        # https://en.wikipedia.org/wiki/SHA-2#Pseudocode
        h0, h1, h2, h3, h4, h5, h6, h7 = self.h
        for offset in range(0, len(data), 64):
            chunks = data[offset : offset + 64]
            w = [0]*64

            for i in range(16):
                w[i] = int.from_bytes(chunks[4*i : 4*i + 4], byteorder='big')
            for i in range(16, 64):
                # ^ - python XOR
                s0 = (self.rightrotate(w[i-15], 7) ^ self.rightrotate(w[i-15], 18) ^\
                self.rightshift(w[i-15], 3)) & 0xFFFFFFFF
                s1 = (self.rightrotate(w[i-2], 17) ^ self.rightrotate(w[i-2], 19) ^\
                self.rightshift(w[i-2], 10)) & 0xFFFFFFFF
                w[i] = (w[i-16] + s0 + w[i-7] + s1) & 0xFFFFFFFF
            a, b, c, d, e, f, g, h = h0, h1, h2, h3, h4, h5, h6, h7

            for i in range(64):
                S1 = (self.rightrotate(e, 6) ^ self.rightrotate(e, 11) ^\
                self.rightrotate(e, 25)) & 0xFFFFFFFF
                # ~ - python not
                ch = ((e & f) ^ ((~e) & g)) & 0xFFFFFFFF
                temp1 = (h + S1 + ch + self.k[i] + w[i]) & 0xFFFFFFFF
                S0 = (self.rightrotate(a, 2) ^ self.rightrotate(a, 13) ^\
                self.rightrotate(a, 22)) & 0xFFFFFFFF
                maj = ((a & b) ^ (a & c) ^ (b & c)) & 0xFFFFFFFF
                temp2 = (S0 + maj) & 0xFFFFFFFF

                new_a = (temp1 + temp2) & 0xFFFFFFFF
                new_e = (d + temp1) & 0xFFFFFFFF
                # Rotate the 8 variables
                a, b, c, d, e, f, g, h = new_a, a, b, c, new_e, e, f, g

            # update h's values
            h0 = (h0 + a) & 0xFFFFFFFF
            h1 = (h1 + b) & 0xFFFFFFFF
            h2 = (h2 + c) & 0xFFFFFFFF
            h3 = (h3 + d) & 0xFFFFFFFF
            h4 = (h4 + e) & 0xFFFFFFFF
            h5 = (h5 + f) & 0xFFFFFFFF
            h6 = (h6 + g) & 0xFFFFFFFF
            h7 = (h7 + h) & 0xFFFFFFFF
            self.h = [h0, h1, h2, h3, h4, h5, h6, h7]
```

Na początku następuje inicjalizacja wartość skrótu. Następnie wykonujemy pętlę po 512-bitowych (64 bajtowych) fragmentach wiadomości, gdzie kolejno:

- inicjalizujemy nową, 64 elementową tablicę w
- fragment dzielimy na szesnaście 32-bitowych słów i zapisujemy w $w[0..15]$
- pozostałe miejsca w tablicy w obliczamy zgodnie z wytycznymi podanymi przez NIST
- tworzymy nowe zmienne tymczasowe i przypisujemy im obecne wartości skrótu
- obliczamy nowe wartości h na podstawie hashowanego przed chwilą zdania w w pętli zawierającej 64 iteracje. Zgodnie z wytycznymi NIST.

5. po wykonaniu powyższego kodu mamy do dyspozycji 8, 32-bitowych wartości które należy połączyć

```
    def digest(self):
        return sum(self.leftshift(x, 32 * i) for i, x in enumerate(self.h[::-1]))
```

6. oraz doprowadzić do odpowiedniego formatu

```
    def hexdigest(self):
        digest = self.digest()
        raw = digest.to_bytes(32, byteorder='big')
        format_str = '{:0' + str(64) + 'x}'
        return format_str.format(int.from_bytes(raw, byteorder='big'))
```

Ostatecznie wszystko to spina w całość funkcja:

```
    def hash_message(self, message):
        msg = self.process_message(bytes(message, encoding='utf8'))
        self.update_hash(msg)
        return self.hexdigest()
```

3 Wyniki

Aby przetestować kod najlepiej porównać wyniki jakie zwraca, z wynikami zwracanymi przez rekomendowane metody. W tym przypadku będzie to funkcja `update()` oraz `hexdigest()` z biblioteki **hashlib**

```
In [1]: from sha256 import SHA256

In [2]: SHA256().hash_message("The quick brown fox jumps over the lazy dog")
Out[2]: 'd7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592'

In [3]: import hashlib
        h = hashlib.sha256()
        h.update(b"The quick brown fox jumps over the lazy dog")
        h.hexdigest()
Out[3]: 'd7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592'

In [4]: hashlib.sha256().update(b"The quick brown fox jumps over the lazy dog")
        assert SHA256().hash_message("The quick brown fox jumps over the lazy dog") == h.hexdigest()
```

Rysunek 1: Porównanie własnej implementacji (na górze) z wbudowaną w bibliotekę **hashlib** (na dole)

4 Podsumowanie

Jak widać obie metody zwracają dokładnie to samo. Oznacza to że moja implementacja jest prawidłowo wykonana. Natomiast jeżeli przetestować wydajność obu rozwiązań, moje wypada dużo gorzej

```
In [6]: start_time = time.time()
        for i in range(10000):
            SHA256().hash_message("The quick brown fox jumps over the lazy dog")
        print("--- %s seconds ---" % (time.time() - start_time))

        --- 4.360559463500977 seconds ---

In [8]: start_time = time.time()
        for i in range(10000):
            hashlib.sha256()
            h.update(b"The quick brown fox jumps over the lazy dog")
            h.hexdigest()
        print("--- %s seconds ---" % (time.time() - start_time))

        --- 0.023920774459838867 seconds ---
```

Rysunek 2: Porównanie własnej implementacji (na górze) z wbudowaną w bibliotekę hashlib (na dole)

Jak widać na powyższym rysunku implementacja biblioteki **hashlib** jest ok 180 razy szybsza. Wynika to z faktu, że jest to część **CPython'a**, czyli standardowej implementacji języka Python napisanej w języku C.