Міністерство освіти і науки України

Національний технічний університет «ХПІ»

Навчально-науковий інститут комп’ютерних наук та інформаційних технологій

Кафедра комп’ютерної інженерії та програмування

ЗВІТ

з лабораторної роботи № 1

з дисципліни «Сучасні технології безпечного програмування»

«ХЕШУВАННЯ»

Виконав:

студент гр. КН-Н922б

Кулик Д.І.

Перевірив:

Бульба С. С.

Харків – 2022

Мета роботи: Дослідити принципи роботи хешування.

Індивідуальне завдання

Дослідити існуючі механізми хешування. Реалізувати алгоритм хешування SHA (будь-якої версії). Реалізацію інших алгоритмів хешування слід обмовити з викладачем.

Довести коректність роботи реалізованого алгоритму шляхом порівняння результатів з існуючими реалізаціями (напр. утилітою sha1sum).

Хід роботи

SHA-2 (Secure Hash Algorithm 2), частиною якого є SHA-256, є одним із найпопулярніших алгоритмів хешування. Криптографічний хеш, який також часто називають «дайджест», «відбиток пальця» або «підпис», — це майже ідеально унікальний рядок символів, який генерується з окремого фрагмента введеного тексту. SHA-256 генерує 256-бітний (32-байтний) підпис.

Важливі фрагменти програми

```python
# Padding
length = len(message) * 8  # len(message) is number of BYTES!!!

message.append(0x80)

while (len(message) * 8 + 64) % 512 != 0:
    message.append(0x00)

message += length.to_bytes(8, 'big')  # pad to 8 bytes or 64 bits

assert (len(message) * 8) % 512 == 0, "Padding did not complete properly!"

# Parsing
blocks = []  # contains 512-bit chunks of message
for i in range(0, len(message), 64):  # 64 bytes is 512 bits
    blocks.append(message[i:i+64])
```

Рисунок 1 – Pre-Processing

```
# Setting Initial Hash Value
h0 = 0x6a09e667
h1 = 0xbb67ae85
h2 = 0x3c6ef372
h3 = 0xa54ff53a
h5 = 0x9b05688c
h4 = 0x510e527f
h6 = 0x1f83d9ab
h7 = 0x5be0cd19
```

Рисунок 2 – Initialize Hash Values (h)

```
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
]
```

Рисунок 3 – Initialize Round Constants (k)

```
# SHA-256 Hash Computation
for message_block in blocks:
```

Рисунок 4 – Chunk Loop

```
# Prepare message schedule
message_schedule = []
for t in range(0, 64):
    if t <= 15:
        # adds the t'th 32 bit word of the block,
        # starting from leftmost word
        # 4 bytes at a time
        message_schedule.append(bytes(message_block[t*4:(t*4)+4]))
        # print(message_schedule)
    else:
        term1 = _sigma1(int.from_bytes(message_schedule[t-2], 'big'))
        term2 = int.from_bytes(message_schedule[t-7], 'big')
        term3 = _sigma0(int.from_bytes(message_schedule[t-15], 'big'))
        term4 = int.from_bytes(message_schedule[t-16], 'big')

        # append a 4-byte byte object
        schedule = ((term1 + term2 + term3 + term4) %
                    2**32).to_bytes(4, 'big')
        message_schedule.append(schedule)

assert len(message_schedule) == 64
```

Рисунок 5 – Create Message Schedule

```
    # Initialize working variables
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4
    f = h5
    g = h6
    h = h7

    # Iterate for t=0 to 63
    for t in range(64):
        t1 = ((h + _capsigma1(e) + _ch(e, f, g) + K[t] +
               int.from_bytes(message_schedule[t], 'big')) % 2**32)

        t2 = (_capsigma0(a) + _maj(a, b, c)) % 2**32

        h = g
        g = f
        f = e
        e = (d + t1) % 2**32
        d = c
        c = b
        b = a
        a = (t1 + t2) % 2**32
```

Рисунок 6 – Compression

```
# Compute intermediate hash value
h0 = (h0 + a) % 2**32
h1 = (h1 + b) % 2**32
h2 = (h2 + c) % 2**32
h3 = (h3 + d) % 2**32
h4 = (h4 + e) % 2**32
h5 = (h5 + f) % 2**32
h6 = (h6 + g) % 2**32
h7 = (h7 + h) % 2**32
```

Рисунок 7 – Modify Final Values

Результати роботи програми

```
C:\Users\Daniil\PycharmProjects\stbp\Scripts\python.exe C:/Users/Daniil/PycharmProjects/stbp/LABS/kulyk01/main.py
Введений текст: Daniil Kulyk
Розроблений SHA256 = 88d44f1bf7c5886b9f0eb28963037d7d9fff9e99a7c24a869124fede16bc46b8
Hashlib SHA256     = 88d44f1bf7c5886b9f0eb28963037d7d9fff9e99a7c24a869124fede16bc46b8
```

Рисунок 8 – Результат виконання програми

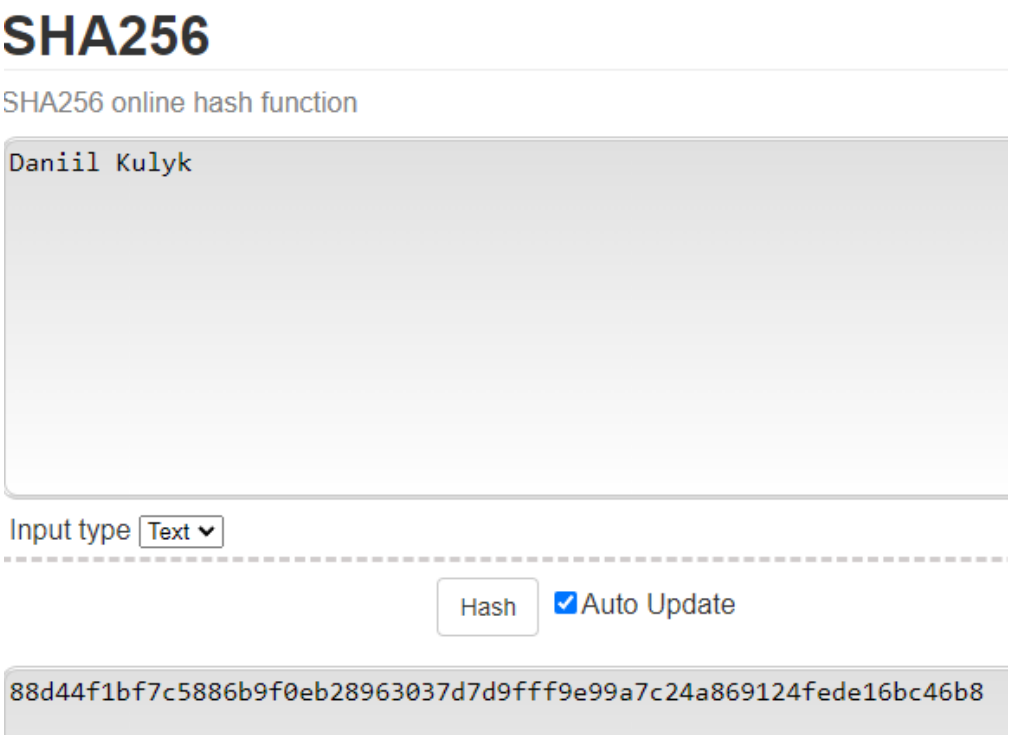

Рисунок 8 – Результат хешування за допомогою ресурсу https://emn178.github.io/online-tools/sha256.html

При порівнянні можемо побачити що результат виконання реалізацій алгоритму однаковий в усіх випадках.

Висновки: в результаті виконання лабораторної роботи було досліджено принципи роботи хешування. В результаті порівняння власної реалізації алгоритму з вже реалізованими була виявлена ідентичність роботи, що доводить коректність першого.