Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9

Выполнил студент группы…….....КС-30 Лихолат Полина Николаевна
Ссылка на репозиторий: ……..https://github.com/MUCTR-IKT-CPP/Likholat_algorithms.git

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович

Дата сдачи: 13.03.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать 1 из ниже приведенных алгоритмов хеширования:

- SHA1

Доп вариант для тех кто хочет посложнее:

- SHA3

Для реализованной хеш функции провести следующие тесты:

- Провести сгенерировать 1000 пар строк длинной 128 символов отличающихся друг от друга 1,2,4,8,16 символов и сравнить хеши для пар между собой, проведя поиск одинаковых последовательностей символов в хешах и подсчитав максимальную длину такой последовательности. Результаты для каждого количества отличий нанести на график, где по оси x кол-во отличий, а по оси y максимальная длинна одинаковой последовательности.
- Провести N = 10^i(i от 2 до 6) генерацию хешей для случайно сгенерированных строк длинно 256 символов, и выполнить поиск одинаковых хешей в итоговом наборе данных, результаты привести в таблице где первая колонка это N генераций, а вторая таблица наличие и кол-во одинаковых хешей, если такие были.
- Провести по 1000 генераций хеша для строк длинной n (64, 128, 256, 512, 1024, 2048, 4096, 8192)(строки генерировать случайно для каждой серии), подсчитать среднее время и построить зависимость скорости расчета хеша от размера входных данных

Описание метода/модели.

Общие сведения:

Криптографическая хеш-функция - это математический алгоритм, который отображает данные произвольного размера в битовый массив фиксированного размера.

Результат, производимый хеш-функцией, называется «хеш-суммой» или же просто «хешем», а входные данные часто называют «сообщением».

Для идеальной хеш-функции выполняются следующие условия:
a) хеш-функция является детерминированной, то есть одно и то же сообщение приводит к одному и тому же хеш-значению
b) значение хеш-функции быстро вычисляется для любого сообщения
c) невозможно найти сообщение, которое дает заданное хеш-значение
d) невозможно найти два разных сообщения с одинаковым хеш-значением
e) небольшое изменение в сообщении изменяет хеш настолько сильно, что новое и старое значения кажутся некоррелирующими

Свойства:

Криптографическая хеш-функция должна уметь противостоять всем известным типам криптоаналитических атак.

В теоретической криптографии уровень безопасности хеш-функции определяется с использованием следующих свойств:

Pre-image resistance

Имея заданное значение ***h***, должно быть сложно найти любое сообщение ***m*** такое, что ***h = hash(m)***.

Second pre-image resistance

Имея заданное входное значение ***m1***, должно быть сложно найти другое входное значение ***m2*** такое,

что ***hash(m1) = hash(m2)***.

SHA-1 (Secure Hash Algorithm 1) - это алгоритм хеширования, который создает 160-битный хеш-код из входного сообщения произвольной длины (не более 2^64 битов).

Алгоритм SHA-1 работает по следующему алгоритму:

- Предварительная обработка: Входное сообщение дополняется нулями до 512 битов. Затем к сообщению дополняется дополнительный блок, который содержит длину исходного сообщения в битах (записывается в 64 битах).
- Инициализация переменных: SHA-1 использует 5 32-битных переменных (A, B, C, D, E), которые инициализируются начальными значениями (заданными в стандарте).
- Обработка блоков: Входное сообщение разбивается на блоки по 512 бит. Каждый блок обрабатывается следующим образом:
- Блок разбивается на 16 32-битных слов.
- Значения переменных A, B, C, D, E изменяются в зависимости от значений слов блока и предыдущих значений переменных.
- Значения переменных A, B, C, D, E сохраняются для следующего блока.
- Формирование хеш-кода: После обработки всех блоков значения переменных A, B, C, D, E составляют 160-битный хеш-код.

SHA-1 является одним из наиболее распространенных алгоритмов хеширования, но сейчас он считается устаревшим и рекомендуется использовать более безопасные алгоритмы, такие как SHA-2 или SHA-3.

Выполнение задачи.

```
def sha1_hash(message):
    # Initialize variables
    h0 = 0x67452301
    h1 = 0xEFCDAB89
    h2 = 0x98BADCFE
    h3 = 0x10325476
    h4 = 0xC3D2E1F0

    # Append padding bits and length
    ml = len(message) * 8
    message += b'\x80'
    while len(message) % 64 != 56:
        message += b'\x00'
    message += struct.pack('>Q', ml)

    # Process message in 512-bit blocks
    for i in range(0, len(message), 64):
        block = message[i:i+64]
        w = list(struct.unpack('>16L', block))
        for j in range(16, 80):
            w.append(left_rotate(w[j-3] ^ w[j-8] ^ w[j-14] ^ w[j-16], 1))

        # Initialize hash value for this block
        a = h0
        b = h1
        c = h2
        d = h3
        e = h4

        # Main loop
        for j in range(80):
            if j < 20:
                f = (b & c) | ((~b) & d)
                k = 0x5A827999
            elif j < 40:
                f = b ^ c ^ d
                k = 0x6ED9EBA1
            elif j < 60:
                f = (b & c) | (b & d) | (c & d)
                k = 0x8F1BBCDC
            else:
                f = b ^ c ^ d
                k = 0xCA62C1D6

            temp = left_rotate(a, 5) + f + e + k + w[j] & 0xffffffff
            e = d
            d = c
```

```
            c = left_rotate(b, 30)
            b = a
            a = temp

        # Add this block's hash to result so far
        h0 = h0 + a & 0xffffffff
        h1 = h1 + b & 0xffffffff
        h2 = h2 + c & 0xffffffff
        h3 = h3 + d & 0xffffffff
        h4 = h4 + e & 0xffffffff

    # Produce the final hash value
    return '%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)
```

1. Инициализируем начальные значения переменных, которые называются "хэш-значениями":
   - h0 = 0x67452301
   - h1 = 0xEFCDAB89
   - h2 = 0x98BADCFE
   - h3 = 0x10325476
   - h4 = 0xC3D2E1F0
2. Добавляем в сообщение биты заполнения и длину сообщения. Это делается для того, чтобы сообщение имело длину, кратную 512 битам, как это требуется для дальнейшей обработки:
   - Добавляем бит "1" в конец сообщения
   - Добавляем биты "0" в конец сообщения до тех пор, пока длина сообщения в битах не станет кратной 512
   - Добавляем длину сообщения (в битах) в конец сообщения в виде 64-битного числа
3. Разбиваем сообщение на блоки по 512 бит.
4. Для каждого блока делаем следующее:
   - Разбиваем блок на 16 слов по 32 бита каждое
   - Для i от 16 до 79 выполняем следующее вычисление: w[i] = left_rotate(w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16], 1), где left_rotate - это функция циклического сдвига влево на заданное количество бит, а ^ - операция "исключающее ИЛИ".
   - Инициализируем переменные a, b, c, d, e значениями h0, h1, h2, h3, h4 соответственно.
   - Выполняем главный цикл 80 раз, где каждую итерацию:
     - Вычисляем f и k в зависимости от значения j
     - Вычисляем промежуточное значение temp
     - Обновляем переменные a, b, c, d, e
     - Обновляем хэш-значения h0, h1, h2, h3, h4 на основе значений a, b, c, d, e
     - Переходим к следующему блоку
5. Получаем окончательное хэш-значение путем объединения пяти хэш-значений h0, h1, h2, h3, h4 в одну строку.

```
def left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff
```

Эта функция **left_rotate** реализует циклический сдвиг 32-битового числа **n** на **b** бит влево, а также возвращает результат сдвига, усеченный до 32 бит.
Для выполнения циклического сдвига влево на **b** бит в **n** используется операция побитового сдвига влево << на **b** бит. Также выполняется побитовое логическое или | с результатом побитового сдвига **n** вправо на **32-b** бит (то есть на **32** минус **b** бит) для сохранения смысла, потерянного при сдвиге.
В итоге, функция возвращает результат сдвига, усеченный до 32 бит, используя операцию побитового И

& с числом **0xffffffff**, которое представляет максимально возможное 32-битное число (в двоичном виде это 32 единицы). Это обусловлено тем, что в Python все целые числа имеют динамический размер, поэтому результатом операции сдвига может быть целое число большего размера.

Результаты тестов:

Тест 1:
По графику, полученному в результате проведения теста 1 (Рис. 1), мы видим, что совпадение в хеше уменьшается от увеличения количества разных символов в строке

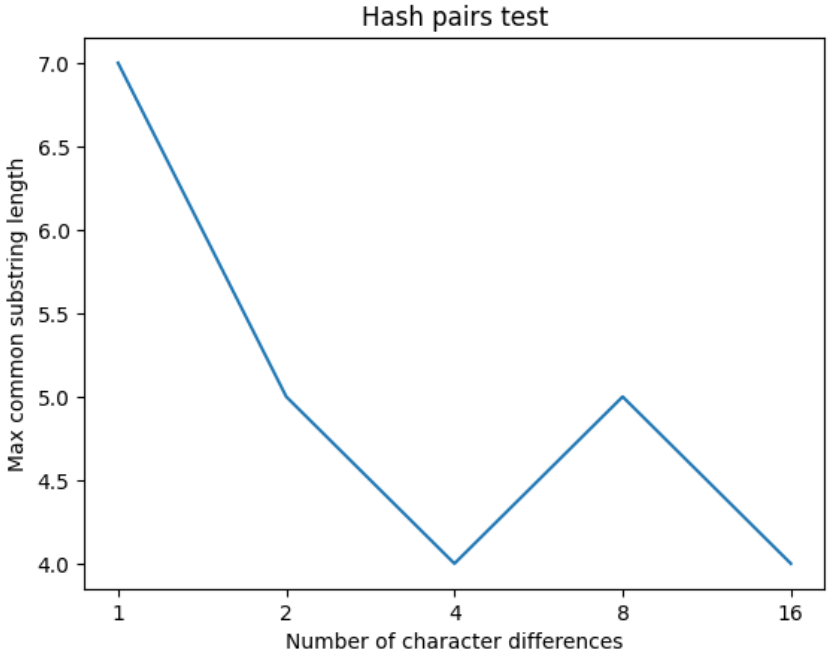

Рис. 1 График зависимости максимального совпадения хеша от количества разных символов в строке

Тест 2:
Результаты теста 2 приведены ниже, как мы видим - ни одного дублирующегося хеша не было обнаружено.

{"100": {"has_duplicates": false, "num_duplicates": 1},
"1000": {"has_duplicates": false, "num_duplicates": 1},
"10000": {"has_duplicates": false, "num_duplicates": 1},
"100000": {"has_duplicates": false, "num_duplicates": 1},
"1000000": {"has_duplicates": false, "num_duplicates": 1}}

Тест 3:
По графику, приведенному ниже (Рис. 2), мы видим, что данная реализация хеширования затрачивает достаточно времени, и при большом объеме вычислений может быть проблемой для проекта.

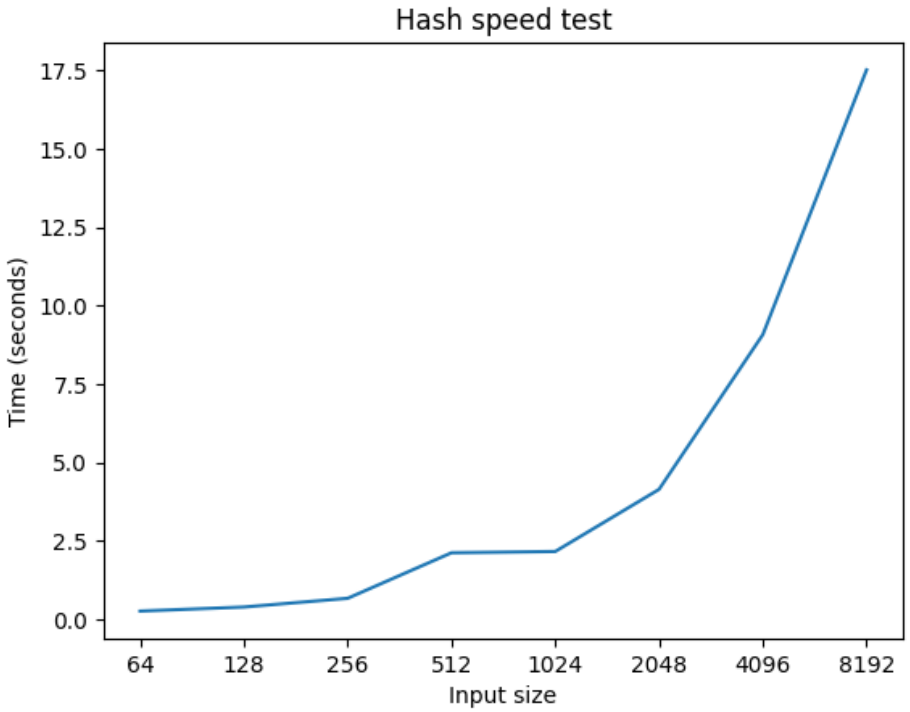

Рис. 2 График зависимости время хеширования от длины строки

Заключение.

Хеш-функция SHA-1 широко применялась в прошлом для обеспечения целостности данных, проверки подлинности и шифрования. Однако, в настоящее время рекомендуется использовать более сильные хеш-функции, такие как SHA-256, SHA-384 или SHA-512, так как SHA-1 стала уязвимой к атакам, основанным на коллизиях, которые позволяют найти два разных входных сообщения с одним и тем же хешем. Это означает, что вредоносные пользователи могут создавать поддельные данные с тем же хешем, что и оригинальные данные, что делает SHA-1 ненадежной для использования в криптографических приложениях. Если вы все еще используете SHA-1 для своих приложений, рекомендуется обновиться до более безопасной хеш-функции.