现代密码学作业 SHA-256 实现文档

> 计 26 郑睿阳

前言

- 本文档是关于清华大学计算机系 2024-春 现代密码学课程实现 SHA-256 算法的作业。其不仅承担了对于程序的说明，也承担了对于 SHA-256 介绍的作用，可供复习，学习时参考使用。

简介

- SHA-256 算法是美国国家安全局研发的安全哈希加密算法 (**S**ecure **H**ash **A**lgorithm) 的一类。该算法能够接受信息并为其生成一个长度为 256 bits 的信息摘要，因而得名。所谓信息摘要，便是算法在接受明文后，通过其对应的哈希加密算法计算出来的一个关于该明文的哈希值 (默认读者已经熟悉哈希的概念) 作为该明文的 "身份证" 。如果有不法者篡改我们需要传输的明文，这个 "身份证" 的计算结果就会改变，从而能够被及时的识别出来。在 SHA-256 当中，哈希碰撞的概率极低，算法的安全性从而得到保障。

函数说明

- 以下是一些在加密过程当中我们会用到的函数：

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) && (4.2)\\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) && (4.3)
\end{aligned}
$$

$$
\begin{aligned}
\sum_0^{\{256\}}(x) &= ROTR^{2}(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) && (4.4)\\
\sum_1^{\{256\}}(x) &= ROTR^{6}(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) && (4.5)\\
\sigma_0^{\{256\}}(x) &= ROTR^{7}(x) \oplus ROTR^{18}(x) \oplus SHR^{3}(x) && (4.6)\\
\sigma_1^{\{256\}}(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) && (4.7)
\end{aligned}
$$

- 其中，$ROTR(x)$ 指的是向右循环移位，$SHR(x)$ 指的是向右移 (但是不循环)。

预处理阶段

Step 1 填充

- 假设明文 M 长为 l bits, 我们希望将消息填充为长度为 512 bits 倍数的信息。
  - 首先将 '1' 放置在消息的末尾, 并后接 k bit '0', 使得 k 是方程 $l+1+k \equiv 448(mod 512)$ 的最小解。
  - 现在得到的消息 M' 的长度应该是模 512 余 448 的。然后在后面再填充 64 bits 的信息代表原来明文 M 的长度。
  - 在本人的代码视线当中，填充通过函数 `padding(unsigned char* data, int len)` 实现:

```
    vector<unsigned char> buf;
    int padding(unsigned char* data, int len) {
        // len is in bytes, fill buf with the padding data
        // First, find the smallest k, so that len + 1 + k = 448 mod 512
        int first_mod = (len * 8 + 1) % 512;
        int k = (first_mod <= 448) ? (448 - first_mod) : (512 - first_mod +
448);
        for (int i = 0; i < len; i++) {
            buf.push_back(data[i]);
        }
        int pad_len = ((k + 1) >> 3);
        for (int i = 0; i < pad_len; i++) {
            if (i == 0) {
                buf.push_back((unsigned char)0x80);
            }
            else {
                buf.push_back((unsigned char)0x00);
            }
        }
        // Finally, 64 more bits for the initial length
        char chars[8];
        intToChars((len << 3), chars);
        for (int i = 7; i >= 0; i--) {
            buf.push_back(chars[i]);
        }
        return buf.size();
    }
```

    其中，`intToChars(uint32_t num, char* chars)` 函数的目的是原来明文的长度转化为 `char` 类型，从而将其对应的 64 bits 附加在 `buf` 的后面。具体代码实现如下:

```
void intToChars(uint32_t num, char* chars) {
        long long n = (long long)num;
        for (int i = 0; i < 8; ++i) {
            chars[i] = (n >> (i * 8)) & 0xFF;
        }
    }
```

Step 2 信息切片

- 将刚才得到的信息 M' 切成 N 个 512 bits 的 **模块**。由于 512 bits 的输入信息可以被表示为 16 个 32 bits 的词语的连接 ($32 * 16 = 512$), 因此我们可以将第 i 个**模块** 拆分成 $M_0^{(i)}, M_1^{(i)}, ..., M_{15}^{i}$。其中, 这 16 部分每个都是 32 bit 的。

Step 3 设置初始 Hash 值

- 在 SHA-256 算法当中, 存在 8 个初始的 Hash 值。在后面我们将会看到它们的用途。

- 初始的 Hash 值设置如下:

$$
\begin{aligned}
H_0^{(0)} &= \texttt{8C3D37C819544DA2} \\
H_1^{(0)} &= \texttt{73E1996689DCD4D6} \\
H_2^{(0)} &= \texttt{1DFAB7AE32FF9C82} \\
H_3^{(0)} &= \texttt{679DD514582F9FCF} \\
H_4^{(0)} &= \texttt{0F6D2B697BD44DA8} \\
H_5^{(0)} &= \texttt{77E36F7304C48942} \\
H_6^{(0)} &= \texttt{3F9D85A86A1D36C8} \\
H_7^{(0)} &= \texttt{1112E6AD91D692A1}
\end{aligned}
$$

加密阶段

- 假设密码有 N 个长为 512 bit 的模块, 则
- 对于每一个模块 (也就意味着以下的步骤要循环 N 轮) :
  - 首先计算出对于这个模块加密所必需的 64 个 W_i 值, 计算方法为:

$$
W_t = \begin{cases}
M_t^{(i)} & 0 \le t \le 15 \\
\sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63
\end{cases}
$$

注意，在 SHA-256 当中 $t \in [0, 64)$，只取这个范围的即可。具体的代码实现如下:

```
for (int i = 0; i < 64; i++) {
    if (i < 16) {
        w[i] = concat_char(offset + (i << 2));
    }
    else {
        w[i] = calc_w(i);
    }
}
```

- 在上面的代码当中，`concat_char(uint32 start)` 的作用是将 `char` 类型的 `buf` 中的内容转化为 `uint32` 类型的数据:

```
uint32 concat_char(uint32 start) {
    uint32 result2 = (uint32)buf[start + 3];
    result2 |= ((uint32)buf[start + 2] << 8);
    result2 |= ((uint32)buf[start + 1] << 16);
    result2 |= ((uint32)buf[start] << 24);
    return result2;
}
```

- 而 `calc_w(int i)` 则是采用了上面的定义:

```
uint32 calc_w(int i) {
    return (sigma1(w[i - 2]) + w[i - 7] + sigma0(w[i - 15]) + w[i - 16]);
}
```

- 然后，将算法当中我们所需要用到的 a 到 h 这 8 个变量进行赋值:

$$a = H_0^{(i-1)}$$
$$b = H_1^{(i-1)}$$
$$c = H_2^{(i-1)}$$
$$d = H_3^{(i-1)}$$
$$e = H_4^{(i-1)}$$
$$f = H_5^{(i-1)}$$
$$g = H_6^{(i-1)}$$
$$h = H_7^{(i-1)}$$

- 具体代码如下:

```
a = hash[0];
b = hash[1];
c = hash[2];
d = hash[3];
e = hash[4];
f = hash[5];
g = hash[6];
h = hash[7];
```

- 然后进入主循环: 在这里我们循环 63 轮:

3. For t=0 to 63:

{

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$

$$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

}

- 具体代码为:

```
for (int i = 0; i < 64; i++) {
    uint32 T1 = h + Sigma1(e) + Ch(e, f, g) + SHA256_K[i] + w[i];
    uint32 T2 = Sigma0(a) + Maj(a, b, c);
    h = g;
    g = f;
    f = e;
    e = d + T1;
    d = c;
    c = b;
    b = a;
    a = T1 + T2;
}
```

- 最后，将哈希值进行更新:

4. Compute the i^{th} intermediate hash value $H^{(i)}$:

$$
\begin{aligned}
H_0^{(i)} &= a + H_0^{(i-1)} \\
H_1^{(i)} &= b + H_1^{(i-1)} \\
H_2^{(i)} &= c + H_2^{(i-1)} \\
H_3^{(i)} &= d + H_3^{(i-1)} \\
H_4^{(i)} &= e + H_4^{(i-1)} \\
H_5^{(i)} &= f + H_5^{(i-1)} \\
H_6^{(i)} &= g + H_6^{(i-1)} \\
H_7^{(i)} &= h + H_7^{(i-1)}
\end{aligned}
$$

    - 具体的代码实现为:

```
hash[0] = a + hash[0];
hash[1] = b + hash[1];
hash[2] = c + hash[2];
hash[3] = d + hash[3];
hash[4] = e + hash[4];
hash[5] = f + hash[5];
hash[6] = g + hash[6];
hash[7] = h + hash[7];
```

- 将以上的步骤重复 N 轮，最后得到的 `hash[8]` 当中的内容即为我们所需要得到的数字签名。

优化情况

- 实际上，在程序当中并没有什么刻意进行优化的部分。所做的一些潜在的提升效率的操作包括使用位运算代替模运算等。SHA-256 的程序性能在不经过刻意优化的情况下已经可以达到相关要求。

效率测量

- 对于长度为 8k bits 的明文进行加密, 最终由于加密速度较快，我们无法准确测量其加密速度:

```
● (base) PS D:\Cryptography\Cryptography\SHA256> .\SHA256.exe .\8kbit_data.txt .\output.txt
  inputfile: .\8kbit_data.txt outputfile: .\output.txt
 the hash value is :
 dd 0b 20 c1 e9 06 b7 64 fc d0 1b 41 aa 06 92 49 4d 5c b6 01 b5 63 2c 3e 11 c7 83 ee 3f a9 13 6e
 SHA 256 Use: 0ms
 The Speed is: nanMbps
● (base) PS D:\Cryptography\Cryptography\SHA256> .\SHA256.exe .\8kbit_data.txt .\output.txt
  inputfile: .\8kbit_data.txt outputfile: .\output.txt
 the hash value is :
 dd 0b 20 c1 e9 06 b7 64 fc d0 1b 41 aa 06 92 49 4d 5c b6 01 b5 63 2c 3e 11 c7 83 ee 3f a9 13 6e
 SHA 256 Use: 0ms
 The Speed is: nanMbps
● (base) PS D:\Cryptography\Cryptography\SHA256> .\SHA256.exe .\8kbit_data.txt .\output.txt
  inputfile: .\8kbit_data.txt outputfile: .\output.txt
 the hash value is :
 dd 0b 20 c1 e9 06 b7 64 fc d0 1b 41 aa 06 92 49 4d 5c b6 01 b5 63 2c 3e 11 c7 83 ee 3f a9 13 6e
 SHA 256 Use: 0ms
 The Speed is: nanMbps
```

- 对于长度为 8M bits 的明文进行加密，重复多次实验结果如下:

```
● (base) PS D:\Cryptography\Cryptography\SHA256> .\SHA256.exe .\8mbit_data.txt .\output.txt
  inputfile: .\8mbit_data.txt outputfile: .\output.txt
 the hash value is :
 3e 0b 85 cb 61 de f8 87 86 be e8 b7 ef fc 25 1e 21 e4 4d 17 62 e0 84 9e 01 9c 3a 7b e3 d4 8e f6
 SHA 256 Use: 6ms
 The Speed is: 1333.33Mbps
● (base) PS D:\Cryptography\Cryptography\SHA256> .\SHA256.exe .\8mbit_data.txt .\output.txt
  inputfile: .\8mbit_data.txt outputfile: .\output.txt
 the hash value is :
 3e 0b 85 cb 61 de f8 87 86 be e8 b7 ef fc 25 1e 21 e4 4d 17 62 e0 84 9e 01 9c 3a 7b e3 d4 8e f6
 SHA 256 Use: 6ms
 The Speed is: 1333.33Mbps
● (base) PS D:\Cryptography\Cryptography\SHA256> .\SHA256.exe .\8mbit_data.txt .\output.txt
  inputfile: .\8mbit_data.txt outputfile: .\output.txt
 the hash value is :
 3e 0b 85 cb 61 de f8 87 86 be e8 b7 ef fc 25 1e 21 e4 4d 17 62 e0 84 9e 01 9c 3a 7b e3 d4 8e f6
 SHA 256 Use: 6ms
 The Speed is: 1333.33Mbps
```

可以认为其速度为 1333.33 Mbps。

参考资料

- SHA-256 官方文档: Secure Hash Standard (SHS)
- B站-哈希函数 (SHA256/SHA3-Keccak) 作者: 朱忠山-密码学硕士 哈希函数 (SHA256/SHA3-Keccak)_哔哩哔哩_bilibili