# 实验四 密码运算

### 一、实验介绍

本实验将通过GNU标准的C语言和ARM汇编代码，在ARM平台上用加密指令实现SHA256算法。

实验说明：本实验通过GNU标准的C语言和汇编代码，使用加密指令来实现SHA256算法。通过对README.md文件的加密，验证了加密算法的作用。

### SHA256算法概述

SHA256是一个哈希函数。哈希函数，又称散列算法，是一种从任何一种数据中创建小的数字“指纹”的方法。散列函数把消息或数据压缩成摘要，使得数据量变小，将数据的格式固定下来。该函数将数据打乱混合，重新创建一个叫做散列值（或哈希值）的指纹。散列值通常用一个短的随机字母和数字组成的字符串来代表。对于任意长度的消息，SHA256都会产生一个256bit长的哈希值，称作消息摘要。SHA256用一个64位的数据来表示原始消息的长度。因此，通过SHA256计算的消息长度必须要小于2^64。长度信息的编码方式为64-bit的大尾端整数。关于大尾端的含义，可以自行查询。

SHA256算法的主要过程如下。

#### 1、常量初始化

SHA256算法中用到了8个哈希初值以及64个哈希常量。其中，SHA256算法的8个哈希初值如下：

h0 := 0x6a09e667

h1 := 0xbb67ae85

h2 := 0x3c6ef372

h3 := 0xa54ff53a

h4 := 0x510e527f

h5 := 0x9b05688c

h6 := 0x1f83d9ab

h7 := 0x5be0cd19

 这些初值是对自然数中前8个质数（2,3,5,7,11,13,17,19）的平方根的小数部分取前32bit而来。在SHA256算法中，用到的64个常量如下：

428a2f98 71374491 b5c0fbcf e9b5dba5

3956c25b 59f111f1 923f82a4 ab1c5ed5

d807aa98 12835b01 243185be 550c7dc3

72be5d74 80deb1fe 9bdc06a7 c19bf174

e49b69c1 efbe4786 0fc19dc6 240ca1cc

2de92c6f 4a7484aa 5cb0a9dc 76f988da

983e5152 a831c66d b00327c8 bf597fc7

c6e00bf3 d5a79147 06ca6351 14292967

27b70a85 2e1b2138 4d2c6dfc 53380d13

650a7354 766a0abb 81c2c92e 92722c85

a2bfe8a1 a81a664b c24b8b70 c76c51a3

d192e819 d6990624 f40e3585 106aa070

19a4c116 1e376c08 2748774c 34b0bcb5

391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3

748f82ee 78a5636f 84c87814 8cc70208

90befffa a4506ceb bef9a3f7 c67178f2

和8个哈希初值类似，这些常量是对自然数中前64个质数

(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97…)的立方根的小数部分取前32bit而来。

#### 2、信息预处理（pre-processing）

SHA256算法中的预处理就是在想要Hash的报文后面补充需要的信息，使整个消息的长度满足指定的结构。该指定的结构由三个部分组成：基本报文、填充比特和附加的64位基本报文长度值。

信息预处理分为两个步骤：附加填充比特和附加长度。

附加填充比特

在报文末尾进行填充，使报文长度在对512取模以后的余数是448。填充是这样进行的：先补第一个比特为1，然后都补0，直到长度满足对512取模后余数是448。需要注意的是，信息必须进行填充，也就是说，即使长度已经满足对512取模后余数是448，补位也必须要进行，这时要填充512个比特。因此，填充是至少补一位，最多补512位。

例：以信息“abc”为例显示补位的过程。

a,b,c对应的16进制数分别是61,62,63，于是原始信息的二进制编码为：01100001 01100010 01100011。

补位第一步，首先补一个“1” ：0110000101100010 011000111。

补位第二步,补423个“0”：01100001 01100010 01100011 10000000 00000000 … 00000000。

补位完成后的数据如下（为了简介用16进制表示）：

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000

为什么是448? 因为在第一步的预处理后，第二步会再附加上一个64bit的数据，用来表示原始报文的长度信息。而448+64=512，正好拼成了一个完整的结构。

附加长度值

附加长度值就是将原始数据（第一步填充前的消息）的长度信息补到已经进行了填充操作的消息后面。由于消息“abc”是3个字符，占用24个bit。因此，在进行了补长度的操作以后，整个消息就变成下面这样了（16进制格式）。

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

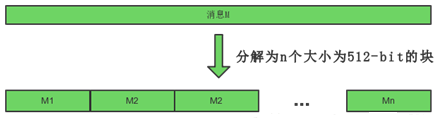
00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000018

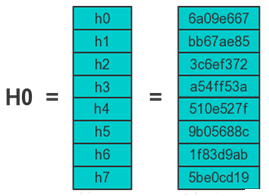
#### 3、计算消息摘要

现在来介绍SHA256算法的主体部分，即消息摘要是如何计算的。

首先：将消息分解成512-bit大小的块。



假设消息M可以被分解为n个块，于是整个算法需要做的就是完成n次迭代，n次迭代的结果就是最终的哈希值，即256bit的数字摘要。一个256-bit的摘要的初始值H0，经过第一个数据块进行运算，得到H1，即完成了第一次迭代。H1经过第二个数据块得到H2，……，依次处理，最后得到Hn，Hn即为最终的256-bit消息摘要。图中256-bit的Hi被描述8个小块，这是因为SHA256算法中的最小运算单元称为“字”（Word），一个字是32位。此外，第一次迭代中，映射的初值设置为前面介绍的8个哈希初值，如下图所示：



下面开始介绍每一次迭代的内容：

先构造64个字（word），对于每一块，将块分解为16个32-bit的big-endian的字，记为w[0], …, w[15]，也就是说，前16个字直接由消息的第i个块分解得到。其余的字由如下迭代公式得到：

https://img2018.cnblogs.com/blog/1488104/201909/1488104-20190911152042421-1781681943.png

 最后进行64次循环。

### 三、各文件代码：

用加密指令实现SHA256算法的C源码sha256.c

/\* sha256.c\*/

#include <stdio.h>

#include <string.h>

#include <stdint.h>

#include <stdbool.h>

void sha256\_block\_data\_order (uint32\_t \*ctx, const void \*in, size\_t num);

// SHA-256 initial hash value

const uint32\_t H\_0[8] = {

0x6a09e667,

0xbb67ae85,

0x3c6ef372,

0xa54ff53a,

0x510e527f,

0x9b05688c,

0x1f83d9ab,

0x5be0cd19,

};

void sha256\_print\_hash(uint32\_t \*H, const char \*title) {

printf("-----------------------------------------------------------------------------------------\n");

printf("| %-85s |\n", title);

printf("-----------------------------------------------------------------------------------------\n");

printf("| H[0] | H[1] | H[2] | H[3] | H[4] | H[5] | H[6] | H[7] |\n");

printf("| %08x | %08x | %08x | %08x | %08x | %08x | %08x | %08x |\n", H[0], H[1], H[2], H[3], H[4], H[5], H[6], H[7]);

printf("-----------------------------------------------------------------------------------------\n");

}

// initialize hash value

void sha256\_init(uint32\_t \*H) {

H[0] = 0x6a09e667;

H[1] = 0xbb67ae85;

H[2] = 0x3c6ef372;

H[3] = 0xa54ff53a;

H[4] = 0x510e527f;

H[5] = 0x9b05688c;

H[6] = 0x1f83d9ab;

H[7] = 0x5be0cd19;

}

int main(int argc, char \*\*argv) {

unsigned int i;

// check arguments

if (argc != 2) {

printf("Usage: %s <input file>\n", argv[0]);

return -1;

}

// open input file

FILE \*fp = fopen(argv[1], "r");

if (!fp) {

printf("Error opening file '%s' for reading.\n", argv[1]);

return -1;

}

// initialize hash value

uint32\_t H[8];

memcpy(H, H\_0, 8\*4);

// read file and calculate hash

uint64\_t bits = 0;

unsigned char buffer[64];

size\_t len;

while (len = fread(buffer, 1, sizeof(buffer), fp)) {

bits += len \* 8;

// preserve value of len by breaking on end of file (or error)

if (len < 64) {

break;

}

sha256\_block\_data\_order(H, buffer,1);

}

// add padding

if (len < 56) {

// padd current block to 56 byte

buffer[len] = 0x80;

i = len + 1;

} else {

// fill up current block and update hash

buffer[len] = 0x80;

for (i = len + 1; i < 64; i++) {

buffer[i] = 0x00;

}

sha256\_block\_data\_order(H, buffer,1);

// add (almost) one block of zero bytes

i = 0;

}

for (; i < 56; i++) {

buffer[i] = 0x00;

}

// add message length in bits in big endian

for (i = 0; i < 8; i++) {

buffer[63 - i] = bits >> (i \* 8);

}

sha256\_block\_data\_order(H, buffer,1);

// print hash

sha256\_print\_hash(H, "Final Hash");

// convert hash to char array (in correct order)

for (i = 0; i < 8; i++) {

buffer[i\*4 + 0] = H[i] >> 24;

buffer[i\*4 + 1] = H[i] >> 16;

buffer[i\*4 + 2] = H[i] >> 8;

buffer[i\*4 + 3] = H[i];

}

// print hash

printf("Hash:\t");

for (i = 0; i < 32; i++) {

printf("%02x", buffer[i]);

}

printf("\n");

return 0;

}

sha256-armv8-aarch32.S源代码

/\* sha256-armv8-aarch32.S \*/

.text

.code 32

# SHA256 assembly implementation for ARMv8 AArch32

.global sha256\_block\_data\_order

.type sha256\_block\_data\_order,%function

.align 2

sha256\_block\_data\_order:

.Lsha256prolog:

vpush {q4}

vldmia r0, {s0, s1, s2, s3, s4, s5, s6, s7}

add r2, r1, r2, lsl #6

.Lsha256loop:

vld1.32 {d16-d19}, [r1]!

adr r3, .LKConstant256

vmov q2, q0

vmov q3, q1

vld1.32 {d20-d23}, [r1]!

vrev32.8 q8, q8

vldmia r3!, {d24, d25}

vrev32.8 q9, q9

vrev32.8 q10, q10

vldmia r3!, {d26, d27}

vadd.i32 q12, q8, q12

sha256su0.32 q8, q9

vrev32.8 q11, q11

vldmia r3!, {d28, d29}

vadd.i32 q13, q9, q13

vmov q4, q2

sha256h.32 q2, q3, q12

sha256h2.32 q3, q4, q12

sha256su0.32 q9, q10

sha256su1.32 q8, q10, q11

vldmia r3!, {d30, d31}

vadd.i32 q14, q10, q14

vmov q4, q2

sha256h.32 q2, q3, q13

sha256h2.32 q3, q4, q13

sha256su0.32 q10, q11

sha256su1.32 q9, q11, q8

vldmia r3!, {d24, d25}

vadd.i32 q15, q11, q15

vmov q4, q2

sha256h.32 q2, q3, q14

sha256h2.32 q3, q4, q14

sha256su0.32 q11, q8

sha256su1.32 q10, q8, q9

vldmia r3!, {d26, d27}

vadd.i32 q12, q8, q12

vmov q4, q2

sha256h.32 q2, q3, q15

sha256h2.32 q3, q4, q15

sha256su0.32 q8, q9

sha256su1.32 q11, q9, q10

vldmia r3!, {d28, d29}

vadd.i32 q13, q9, q13

vmov q4, q2

sha256h.32 q2, q3, q12

sha256h2.32 q3, q4, q12

sha256su0.32 q9, q10

sha256su1.32 q8, q10, q11

vldmia r3!, {d30, d31}

vadd.i32 q14, q10, q14

vmov q4, q2

sha256h.32 q2, q3, q13

sha256h2.32 q3, q4, q13

sha256su0.32 q10, q11

sha256su1.32 q9, q11, q8

vldmia r3!, {d24, d25}

vadd.i32 q15, q11, q15

vmov q4, q2

sha256h.32 q2, q3, q14

sha256h2.32 q3, q4, q14

sha256su0.32 q11, q8

sha256su1.32 q10, q8, q9

vldmia r3!, {d26, d27}

vadd.i32 q12, q8, q12

vmov q4, q2

sha256h.32 q2, q3, q15

sha256h2.32 q3, q4, q15

sha256su0.32 q8, q9

sha256su1.32 q11, q9, q10

vldmia r3!, {d28, d29}

vadd.i32 q13, q9, q13

vmov q4, q2

sha256h.32 q2, q3, q12

sha256h2.32 q3, q4, q12

sha256su0.32 q9, q10

sha256su1.32 q8, q10, q11

vldmia r3!, {d30, d31}

vadd.i32 q14, q10, q14

vmov q4, q2

sha256h.32 q2, q3, q13

sha256h2.32 q3, q4, q13

sha256su0.32 q10, q11

sha256su1.32 q9, q11, q8

vldmia r3!, {d24, d25}

vadd.i32 q15, q11, q15

vmov q4, q2

sha256h.32 q2, q3, q14

sha256h2.32 q3, q4, q14

sha256su0.32 q11, q8

sha256su1.32 q10, q8, q9

vldmia r3!, {d26, d27}

vadd.i32 q12, q8, q12

vmov q4, q2

sha256h.32 q2, q3, q15

sha256h2.32 q3, q4, q15

sha256su1.32 q11, q9, q10

vldmia r3!, {d28, d29}

vadd.i32 q13, q9, q13

vmov q4, q2

sha256h.32 q2, q3, q12

sha256h2.32 q3, q4, q12

vldmia r3!, {d30, d31}

vadd.i32 q14, q10, q14

vmov q4, q2

sha256h.32 q2, q3, q13

sha256h2.32 q3, q4, q13

vadd.i32 q15, q11, q15

vmov q4, q2

sha256h.32 q2, q3, q14

sha256h2.32 q3, q4, q14

vmov q4, q2

sha256h.32 q2, q3, q15

sha256h2.32 q3, q4, q15

teq r1, r2

vadd.i32 q0, q0, q2

vadd.i32 q1, q1, q3

bne .Lsha256loop

.Lsha256epilog:

vstmia r0, {s0, s1, s2, s3, s4, s5, s6, s7}

vpop {q4}

bx lr

.align 5

.LKConstant256:

.word 0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5

.word 0x3956c25b,0x59f111f1,0x923f82a4,0xab1c5ed5

.word 0xd807aa98,0x12835b01,0x243185be,0x550c7dc3

.word 0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174

.word 0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc

.word 0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da

.word 0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7

.word 0xc6e00bf3,0xd5a79147,0x06ca6351,0x14292967

.word 0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13

.word 0x650a7354,0x766a0abb,0x81c2c92e,0x92722c85

.word 0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3

.word 0xd192e819,0xd6990624,0xf40e3585,0x106aa070

.word 0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5

.word 0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0x682e6ff3

.word 0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208

.word 0x90befffa,0xa4506ceb,0xbef9a3f7,0xc67178f2

.size sha256\_block\_data\_order,.-sha256\_block\_data\_order

.align 2

sha256-armv8-aarch64.S：

/\* sha256-armv8-aarch64.S \*/

.text

.arch armv8-a+crypto

# SHA256 assembly implementation for ARMv8 AArch64

.global sha256\_block\_data\_order

.type sha256\_block\_data\_order,%function

.align 2

sha256\_block\_data\_order:

.Lsha256prolog:

stp x29, x30, [sp,#-64]!

mov x29, sp

adr x3, .LKConstant256

str q8, [sp, #16]

ld1 {v16.4s-v19.4s}, [x3], #64

ld1 {v0.4s}, [x0], #16

ld1 {v20.4s-v23.4s}, [x3], #64

add x2, x1, x2, lsl #6

ld1 {v1.4s}, [x0]

ld1 {v24.4s-v27.4s}, [x3], #64

sub x0, x0, #16

str q9, [sp, #32]

str q10, [sp, #48]

ld1 {v28.4s-v31.4s}, [x3], #64

.Lsha256loop:

ld1 {v5.16b-v8.16b}, [x1], #64

mov v2.16b, v0.16b

mov v3.16b, v1.16b

rev32 v5.16b, v5.16b

rev32 v6.16b, v6.16b

add v9.4s, v5.4s, v16.4s

rev32 v7.16b, v7.16b

add v10.4s, v6.4s, v17.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

sha256su0 v5.4s, v6.4s

rev32 v8.16b, v8.16b

add v9.4s, v7.4s, v18.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

sha256su0 v6.4s, v7.4s

sha256su1 v5.4s, v7.4s, v8.4s

add v10.4s, v8.4s, v19.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

sha256su0 v7.4s, v8.4s

sha256su1 v6.4s, v8.4s, v5.4s

add v9.4s, v5.4s, v20.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

sha256su0 v8.4s, v5.4s

sha256su1 v7.4s, v5.4s, v6.4s

add v10.4s, v6.4s, v21.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

sha256su0 v5.4s, v6.4s

sha256su1 v8.4s, v6.4s, v7.4s

add v9.4s, v7.4s, v22.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

sha256su0 v6.4s, v7.4s

sha256su1 v5.4s, v7.4s, v8.4s

add v10.4s, v8.4s, v23.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

sha256su0 v7.4s, v8.4s

sha256su1 v6.4s, v8.4s, v5.4s

add v9.4s, v5.4s, v24.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

sha256su0 v8.4s, v5.4s

sha256su1 v7.4s, v5.4s, v6.4s

add v10.4s, v6.4s, v25.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

sha256su0 v5.4s, v6.4s

sha256su1 v8.4s, v6.4s, v7.4s

add v9.4s, v7.4s, v26.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

sha256su0 v6.4s, v7.4s

sha256su1 v5.4s, v7.4s, v8.4s

add v10.4s, v8.4s, v27.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

sha256su0 v7.4s, v8.4s

sha256su1 v6.4s, v8.4s, v5.4s

add v9.4s, v5.4s, v28.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

sha256su0 v8.4s, v5.4s

sha256su1 v7.4s, v5.4s, v6.4s

add v10.4s, v6.4s, v29.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

sha256su1 v8.4s, v6.4s, v7.4s

add v9.4s, v7.4s, v30.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

add v10.4s, v8.4s, v31.4s

mov v4.16b, v2.16b

sha256h q2, q3, v9.4s

sha256h2 q3, q4, v9.4s

mov v4.16b, v2.16b

sha256h q2, q3, v10.4s

sha256h2 q3, q4, v10.4s

cmp x1, x2

add v1.4s, v1.4s, v3.4s

add v0.4s, v0.4s, v2.4s

b.ne .Lsha256loop

.Lsha256epilog:

st1 {v0.4s,v1.4s}, [x0]

ldr q10, [sp, #48]

ldr q9, [sp, #32]

ldr q8, [sp, #16]

ldr x29, [sp], #64

ret

.align 5

.LKConstant256:

.word 0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5

.word 0x3956c25b,0x59f111f1,0x923f82a4,0xab1c5ed5

.word 0xd807aa98,0x12835b01,0x243185be,0x550c7dc3

.word 0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174

.word 0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc

.word 0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da

.word 0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7

.word 0xc6e00bf3,0xd5a79147,0x06ca6351,0x14292967

.word 0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13

.word 0x650a7354,0x766a0abb,0x81c2c92e,0x92722c85

.word 0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3

.word 0xd192e819,0xd6990624,0xf40e3585,0x106aa070

.word 0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5

.word 0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0x682e6ff3

.word 0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208

.word 0x90befffa,0xa4506ceb,0xbef9a3f7,0xc67178f2

.size sha256\_block\_data\_order,.-sha256\_block\_data\_order

.align 2

Makefile：

CC = gcc

CFLAGS = -O3 -mcpu=generic+crypto

sha256:sha256.c

$(CC) $(CFLAGS) sha256.c sha256-armv8-aarch64.S -o sha256

README.md：

# SHA-256

This is a very basic implementation of a SHA-256 hash according to the [FIPS

180-4 standard](http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf)

in C. I did it for educational purposes, the code is not optimized at all, but

(almost) corresponds line by line to the standard.

The algorithm to process a small block of data is quite simple and very well

described in the standard. I found that correctly implementing the padding is

much harder.

It does not have any dependencies (except for the C standard library of course)

and can be compiled with `make`. When `sha256sum` is installed, a short test can

be run with `make test`.

Usage:

./main <input file>

```

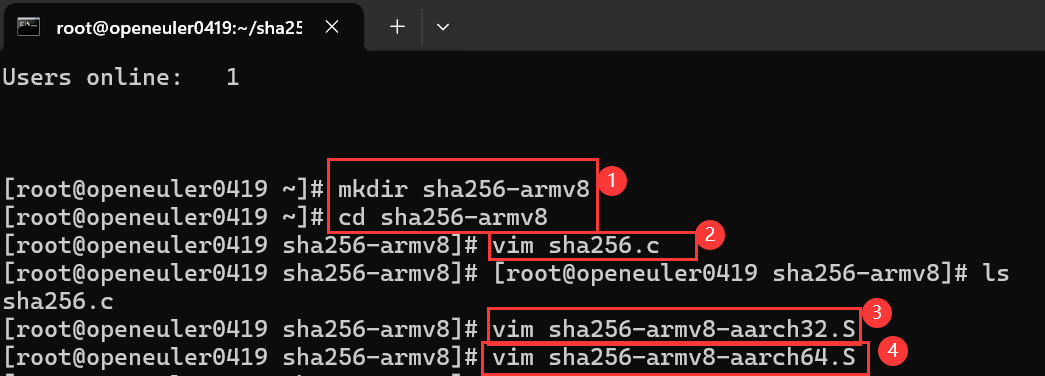
## 四、实验操作

以下步骤以在华为鲲鹏云服务器上执行为例。

一、创建程序源码：

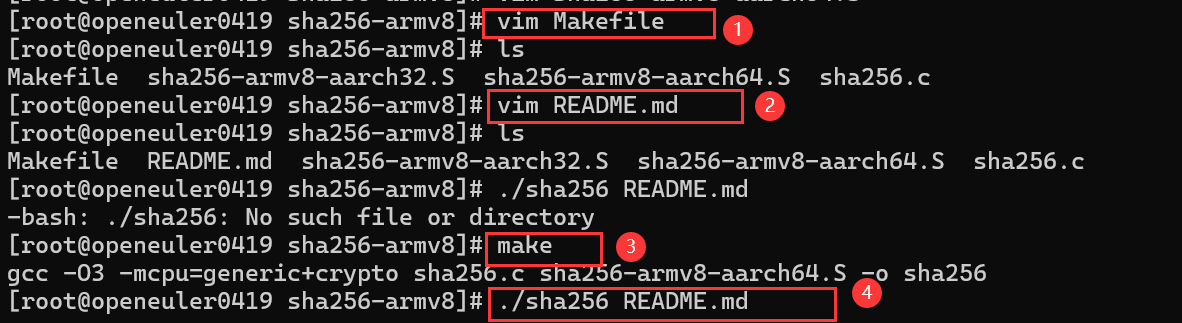
1. 创建sha256-armv8目录存放该程序的所有文件, 并进入sha256-armv8目录
2. 创建用加密指令实现SHA256算法的C源码sha256.c。
3. 创建sha256-armv8-aarch32.S源代码。
4. 创建sha256-armv8-aarch64.S源代码

创建程序源码过程截图：

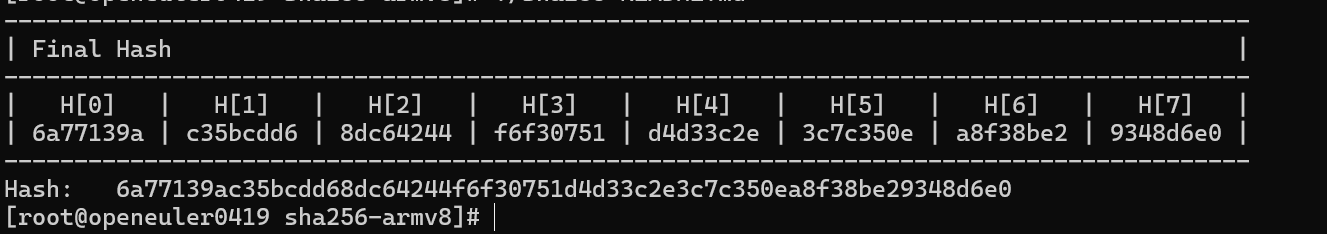


1. 进行编译运行：
2. 创建makefile
3. 创建README.md
4. 编译
5. 运行程序

编译运行过程截图：



运行结果截图：



分析：可以看到通过SHA256加密后，README.md文件的Hash值为：

6a77139ac35bcdd68dc64244f6f30751d4d33c2e3c7c350ea8f38be29348d6e0。

## 思考题

**除了SHA256还有哪些常见的加密算法？**

答：除了SHA-256，还有许多其他常见的加密算法，其中包括：

1.MD5 (Message Digest Algorithm 5)：一种广泛使用的哈希函数，但已经不推荐用于安全目的，因为其存在碰撞攻击的漏洞。

2.SHA-1 (Secure Hash Algorithm 1)：另一种哈希函数，比MD5更安全，但随着时间推移，它也被证明存在碰撞风险，逐渐被弃用。

3.SHA-2 (Secure Hash Algorithm 2)：SHA-2家族包括SHA-224、SHA-256、SHA-384、SHA-512等变体。SHA-256是其中最广泛使用的，提供了更高的安全性和更好的抗碰撞特性。

4.SHA-3 (Secure Hash Algorithm 3)：由NIST作为SHA-2的后续标准推出，提供了一种全新的哈希函数设计，更适合于抵御新型攻击。

5.AES (Advanced Encryption Standard)：对称加密算法，用于加密和解密数据，广泛用于保护敏感信息。

6.RSA (Rivest-Shamir-Adleman)：非对称加密算法，用于数据加密和数字签名，基于公钥和私钥的机制。

7.ECC (Elliptic Curve Cryptography)：另一种非对称加密算法，利用椭圆曲线上的点来实现公钥和私钥的生成和操作，比RSA更高效。

8.Blowfish：对称加密算法，适合于资源受限的环境下，提供了良好的安全性和性能。

9.Twofish：对称加密算法，是Blowfish的改进版本，提供了更高的安全性和更好的性能。

这些加密算法在不同的场景和需求下有着各自的优势和适用性，选择合适的加密算法取决于需要保护的数据类型、性能要求和安全性需求。