期中作业: HMAC-MD5

个人信息

姓名	白家栋
学院	数据科学与计算机学院
专业	软件工程
学号	18342001

目录结构介绍和运行方法

- **目录结构**

```
.
├── Makefile
├── include
│   ├── hmac.h              // hmac 头文件
│   ├── md5.h               // md5 头文件
│   └── test.h              // 测试样例头文件
├── src
    ├── hmac.c
    ├── main.c
    ├── md5.c
    └── test.c
```

- **使用方法**

1. 运行 main 函数(即main.c文件)

   输入:

   ```
   make
   ```

   即可。

2. 运行 test 测试样例(即test.c文件)

   输入:

   ```
   make test
   ```

   即可。

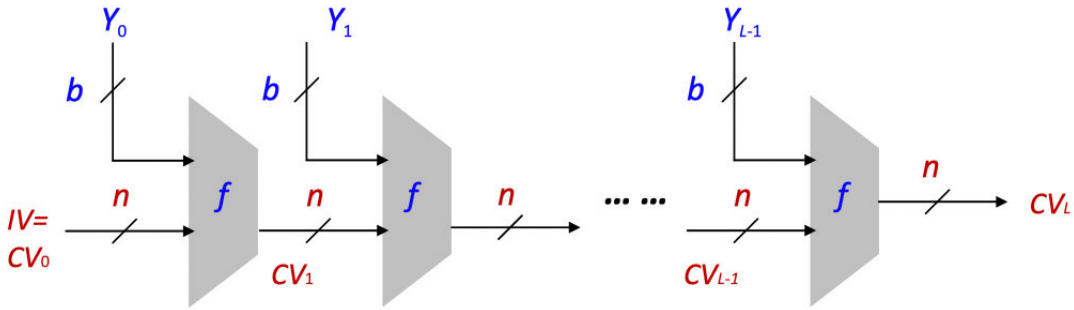
一. MD5 算法原理概述

MD5 是一种 Hash Function. Hash Function 的通式如下:

$$h = H(M)$$

其中，M是一个可变长的消息串，h是一个定长的散列值。MD5 正是这一类函数，它符合 Ralph Merkle 于 1989 年提出的 Hash 函数通用模型，即：

- 把原始消息分成固定长度的数据块
- 将最后一块 padding, 并且在其中包含消息的长度信息
- 设定初始的输入缓冲
- 利用压缩函数，输入第一个块并向下传递，每一个块的输入等于上一个块的输出
- 最后一个块的输出值为 hash 值



具体到 MD5 的实际实现中，可以分为以下几个关键步骤:

1. 初始化 IV, 得到 IV0

上面提到，在分块消息后需要将初始化缓冲输入到第一个块中。这里统一命名向量**IV**。IV 中包含四个 大小为 **32bit** 的寄存器构成，分别是:

- **A:** 0x67452301
- **B**: 0xEFCDAB89
- **C**: 0x98BADCFE
- **D**: 0x10325476

2. 初始化消息分组

基于输入的明文消息，需要按照下面的规则进行分组:

- 判断从当前位置到消息末尾的字节数 m：
  - 如果m >= 64, 说明当前分组不需要填充即可达到 512 bits，直接作为分组返回
  - 如果 m < 64, 说明当前分组需要填充，填充的规则如下:
    - 向后**按位**添加比特值: 100000...., 直到**消息的总长度(以bit为单位)+填充的位数** mod 512 等于 448 停止。
    - 计算消息总长度(原消息，未填充之前的，以bit为单位)，作为最后的64位填充到当前的

消息分组，使得消息总长度为 512。

3. 准备 MD5 压缩函数：HMD5

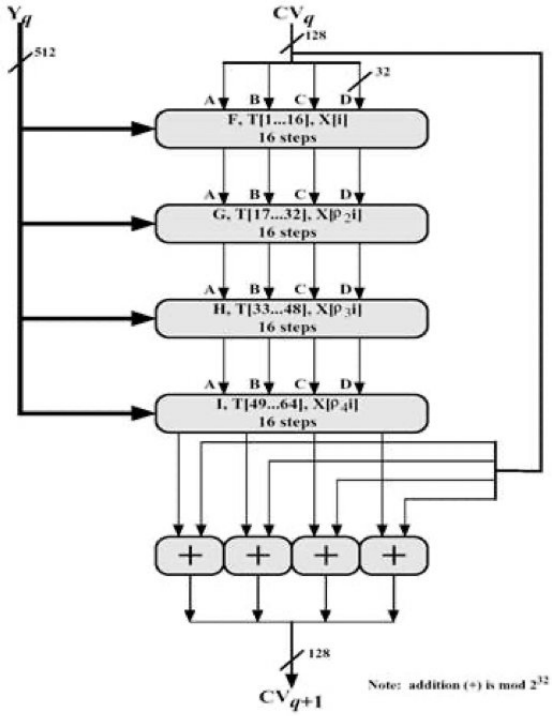
HMD5 接受两个输入:

- 上一个 HMD5 块输出的 IV_i(如果是第0个算法块，则为初始 IV0)，128 位
- 消息分组 M_i，512位

产生一个输出:

- IV_{i+1}，128 位

HMD5 内部结构如下图所示:



具体而言，接受输入的 IV 和 CV 后，要进行4轮循环，每轮循环的流程一致，都要进行**16次**迭代，每轮迭代接收上一次迭代得到的 IV, 迭代的有以下两个步骤:

1. **对 $IV.A$ 进行计算**，计算方式为:

   $a \leftarrow b + ((a + g(b, c, d) + X[k], T[i]) <<< s)$

   其中:

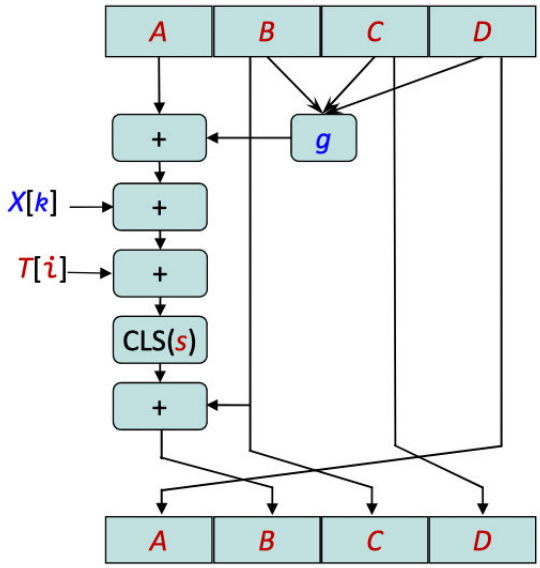
   - a, b, c, d 为 IV 对应缓冲区的 (A, B, C, D)
   - g 为 轮函数 F, G, H, I 中的一个，第一轮使用 F, 第二轮使用 G, 第三轮使用 H, 第四轮使用 I

     F, G, H, I 的定义如下:

轮次	Function g	g(b, c, d)
1	F(b, c, d)	(b & c) | (~b & d)
2	G(b, c, d)	(b & d) | (c & (~d))
3	H(b, c, d)	b ^ c ^ d
4	I(b, c, d)	c ^ (b | (~d))

- X[k], T[i], s 都为给定的值，根据轮次和迭代的次数确定

2. **缓冲区(A, B, C, D) 作循环置换**，迭代的示意图如下:



3. 作为下一个 **HMD5** 的输入来输出

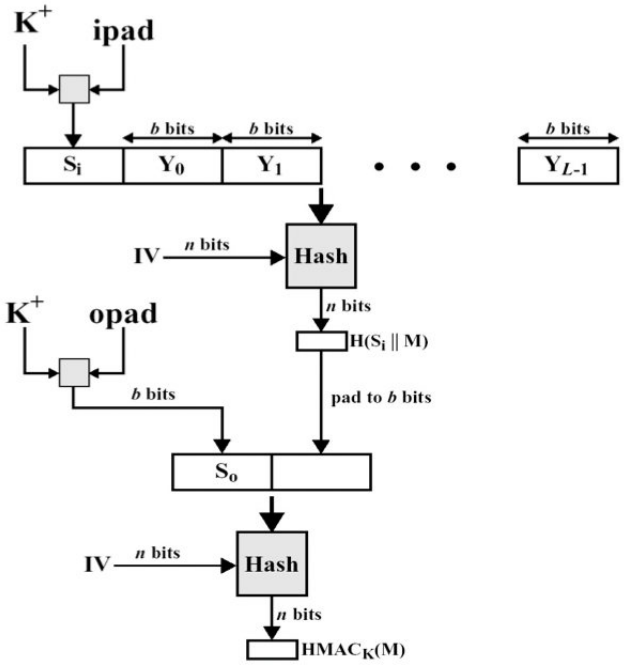
4. 将 IV 拼接

得到了最后一个消息分组的最后一个 HMD5 输出之后，需要将IV中的A, B, C, D按照顺序拼接。同时，每个A, B, C, D在内部还要按照小端的顺序存放。

二. HMAC 算法原理概述

HMAC 是一种通过特别计算方式之后产生的消息认证码（MAC），使用密码散列函数，同时结合一个加密密钥。它可以用来保证资料的完整性，同时可以用来作某个消息的身份验证。

HMAC 算法与 MD5 的关系是：HMAC 可以选择 MD5 作为其使用的密码散列函数，也就是本次作业要实现的 **HMAC-MD5** 。



HMAC-MD5 的步骤比较简单，如下:

- **输入:**
  - 可变长度的消息 **msg**
  - 密钥 **K**
  - block 的字节长度 **B**
- **Step1**: 处理密钥 **K**
  - 如果密钥 K 的字节长度 **小于** B，则在 K 后填充 0，直到 K 的字节程度等于 B
  - 如果密钥 K 的字节长度 **等于** B，则无须额外处理
  - 如果密钥 K 的字节长度 **大于** B，则首先需要 MD5 函数求出 K 对应的 hash 值。由于此时该值固定为16个字节，因此还需要填充48个字节的 0
- **Step2:** 将步骤1中得到的填充过的，长度为 **B** 的密钥与长度同为 **B** 的字节串 **ipad** (ipad 的每一个字节的值都为 `0x36`) 进行 按位异或的操作。
- **Step3:** 将输入的消息填充到 step2 得到的值的末尾
- **Step4**: 将 Step 3 的结果输入到 MD5 函数，得到一个 16 个字节的串
- **Step5**: 将 Step1 的结果与长度为 **B** 的字节串 **opad** (opad 的每一个字节的值都为 `0x5c`) 进行 按位异或的操作。
- **Step6**: 将 Step4 得到的字节串添加到 Step5 结果的末尾
- **Step7**：将 Step6 的结果直接输入 MD5 函数，得到的哈希值作为最终结果

三. 算法设计

首先，由于 HMAC-MD5 算法需要基于 MD5 来实现，因此这里首先介绍 MD5 算法的结构设计。

数据结构设计

由第一部分可知，MD5中有两个重要的数据结构: **IV**, **CV**。其定义如下:

1. IV

```
typedef unsigned int num_t;

typedef struct IV
{
  num_t A;
  num_t B;
  num_t C;
  num_t D;
} IV;
```

可以看到这个结构体有4个数据成员: A, B, C, D。这4个成员分别对应一个 IV 向量中的每一个寄存器。由于 `unsigned int`, 也就是`num_t` 类型的大小为32位，因此这里 `struct IV` 的大小为 128bits。

2. CV

```
typedef struct CV
{
  // 分组后的消息, 共512bits
  num_t *msg;

} CV;
```

CV 的作用是存放分组后的消息，由于 num_t 的大小为32位，所以msg的长度应为16。

MD5 相关方法定义

与实现MD5有关的函数定义全部放在 `include/md5.h` 下，内容如下:

```
// 将4个8位(有效位)的整数转换成一个4个字节的无符号整数
num_t convertOctetToNumber(unsigned int o1, unsigned int o2, unsigned int o3,
unsigned int o4);

// 将消息转换成32位整数数组
num_t *convertMsgToNumbers(unsigned char *msg);

// 将消息长度转换成两个32位整数
num_t *produceLastTwoWord(unsigned long long length);

// 一个Hmd5压缩块
```

```
IV Hmd5(CV msg, IV iv);

// 获取初始向量
IV GetIV0();

// 生成函数
num_t F(num_t b, num_t c, num_t d);

num_t G(num_t b, num_t c, num_t d);

num_t H(num_t b, num_t c, num_t d);

num_t I(num_t b, num_t c, num_t d);

// 循环左移
num_t rotate_left(const num_t value, int shift);

// input表示输入的IV，idx表示第几轮，
IV Round(IV input, num_t idx, num_t xk[16], CV message, num_t s[16], num_t
t[16]);

// 将向量转换为能够输出的char
unsigned char *decode(IV iv);

// 打印结果(16进制的结果，小端)
void print(unsigned char *res);

// 截取子数组
void copySubArray(unsigned char *dst, unsigned char *src, int start, int len);

// md5，用于将普通的字符串直接转成hash值
unsigned char *md5(char *msg);

// md5，用在hmac_md5中
unsigned char *md5_hmac_version(unsigned char *input, long long size);

// 向量相加
IV addIV(IV a, IV b);
```

MD5 关键部分的实现

在 `算法原理部分` 提到，md5 有4个关键步骤，这里逐一介绍其实现:

1. 获取初始向量 IV0

```
IV GetIV0()
{
  IV iv;
  iv.A = convertOctetToNumber(0x1, 0x23, 0x45, 0x67);
  iv.B = convertOctetToNumber(0x89, 0xab, 0xcd, 0xef);
  iv.C = convertOctetToNumber(0xfe, 0xdc, 0xba, 0x98);
  iv.D = convertOctetToNumber(0x76, 0x54, 0x32, 0x10);

  return iv;
};
```

这里要注意的是，由于大小端问题，当我们把 iv 的某个特定寄存器输出时，会得到与上面初始化参数顺序相反的结果(e.g. iv.A = 0x67452301).

2. 初始化消息分组

为了节省时间复杂度，我并没有选择先对消息进行分组再开始输入，而是选择边分组，边进行IV的运算，如下:

```
unsigned char *md5(char *msg)
{
  int group_idx = 0;
  IV iv = GetIV0();
  IV last_iv = GetIV0();
  while (1)
  {
    int base = group_idx * 64;
    int i = 0; // 当前分组的字节数
    while (msg[i + base] != '\0' && i < 64)
    {
      i++;
    }

    if (i == 64)
    {
      // 说明不需要填充，当前分组的长度就为512bit
      unsigned char subMsg[64];

      for (int i = 0; i < 64; i++)
      {
        subMsg[i] = 0;
      }
      memcpy(subMsg, &msg[base], 64);
      num_t *m = convertMsgToNumbers(subMsg);
      CV cv;
```

```
      cv.msg = m;
      iv = Hmd5(cv, iv);
      iv = addIV(iv, last_iv);
      last_iv = iv;
      group_idx++;
    }
    else
    {
      // 说明此时需要填充，填充 10000...，长度使得该长度 % 512 = 448(%64 = 56)
      // 位数
      num_t bit_number = i * 8 + base * 8;
      num_t *klen = produceLastTwoWord(bit_number);

      // 说明这个分组长度小于448bit，需要首先填充1000到448 bit，再添加表示长度的字
      if (i < 56)
      {
        // 64字节
        unsigned char subMsg[64];
        for (int i = 0; i < 64; i++)
        {
          subMsg[i] = 0;
        }

        // 取出组
        memcpy(subMsg, &msg[base], i);

        // 首先填充一个10000000
        subMsg[i] = 0b10000000;

        i++;
        // 填充0
        int num_to_pad = 56 - i;
        for (int j = 0; j < num_to_pad; j++)
        {
          subMsg[i] = 0;
          i++;
        }

        // 转换成16个字
        num_t *m = convertMsgToNumbers(subMsg);
        m[14] = klen[0];
        m[15] = klen[1];

        // 填充完成
        CV cv;
        cv.msg = m;
        iv = Hmd5(cv, iv);
      }
      else
```

```
      {
        // 首先要把当前的分组填充到64个字节，然后要创建一个新分组，填入56个字节的0和最终长度
        // 填满当前分组
        unsigned char subMsg[64];
        for (int i = 0; i < 64; i++)
        {
          subMsg[i] = 0;
        }
        memcpy(subMsg, &msg[base], i);

        // 填充1
        subMsg[i] = 0b10000000;
        i++;
        // 填充0
        int num_to_pad = 64 - i;
        for (int j = 0; j < num_to_pad; j++)
        {
          subMsg[i] = 0;
          i++;
        }
        num_t *m = convertMsgToNumbers(subMsg);
        CV cv;
        cv.msg = m;
        iv = Hmd5(cv, iv);

        iv = addIV(iv, last_iv);
        last_iv = iv;
        // 创建新的分组
        num_t m2[16];
        for (int j = 0; j < 16; j++)
        {
          m2[j] = 0;
          if (j >= 14)
          {
            m2[j] = klen[j - 14];
          }
        }
        cv.msg = m2;
        iv = Hmd5(cv, iv);
      }
      break;
    }
  }
  IV ret = addIV(iv, last_iv);
  unsigned char *res = decode(ret);
  return res;
}
```

3. HMD5 压缩函数实现

HMD5 需要经过4轮，每轮有16次迭代，由于每轮的逻辑一致，因此我这里直接使用了一个 `Round` 函数作为 每轮的实现，然后 HMD5 中直接调用即可，实现如下:

```
// Hmd5 的实现
IV Hmd5(CV msg, IV iv)
{
  // 这里直接调用4轮，将结果输出即可
  for (int i = 0; i < 4; i++)
  {
    iv = Round(iv, i, XK[i], msg, S[i], T[i]);
  }
  return iv;
};

// Round 的实现
IV Round(IV input, num_t idx, num_t xk[16], CV message, num_t s[16], num_t
t[16])
{
  num_t (*g)(num_t, num_t, num_t);
  // 根据该轮的索引选择轮函数
  if (idx == 0)
  {
    g = &F;
  }
  else if (idx == 1)
  {
    g = &G;
  }
  else if (idx == 2)
  {
    g = &H;
  }
  else if (idx == 3)
  {
    g = &I;
  }

  // 每轮有16次迭代
  for (int i = 0; i < 16; i++)
  {
    num_t g_out = g(input.B, input.C, input.D);
    num_t x = message.msg[xk[i]];
    num_t ti = t[i];
    num_t temp = rotate_left(g_out + x + input.A + ti, s[i]);
    num_t new_a = temp + input.B;

    // 循环右移
```

```
    IV new_input;
    new_input.A = input.D;
    new_input.C = input.B;
    new_input.D = input.C;
    new_input.B = new_a;
    input = new_input;
  }
  return input;
};
```

HMAC 方法定义

hmac 所使用到的函数定义列举在 `include/hmac.h` 中，如下:

```
// 获取ipad
unsigned char *ipad(int times);

// 获取opad
unsigned char *opad(int times);

// 获取padding后的key
unsigned char *padKey(unsigned char *key, int key_size, int B);

// HMAC主体
unsigned char *HMAC(unsigned char *key, int B, unsigned char *msg);

// 按位异或
unsigned char *XOR(unsigned char *key, unsigned char *val, int size);
```

HMAC 关键方法实现

首先，在得到输入之后，HMAC要对于用户输入的key进行padding，padding函数是在`padKey`这个函数中实现的，实现如下:

```
unsigned char *padKey(unsigned char *key, int key_size, int B)
{
  unsigned char *ret = (unsigned char *)malloc(sizeof(unsigned char) * B);

  unsigned char *new_key = key;

  // 如果key的大小大于B, 需要先哈希 再padding
  if (key_size > B)
  {
    new_key = md5_hmac_version(key, key_size);
```

```
    key_size = 16;
  }
  // 添加0进行padding
  for (int i = 0; i < B; i++)
  {
    ret[i] = 0;
    if (i < key_size)
    {
      ret[i] = key[i];
    }
  }
  return ret;
};
```

padding之后的key首先需要与 `ipad` 进行异或，也就是 `XOR` 函数，实现如下:

```
// 逐位异或
unsigned char *XOR(unsigned char *key, unsigned char *val, int size)
{
  unsigned char *ret = (unsigned char *)malloc(sizeof(unsigned char) * size);
  for (int i = 0; i < size; i++)
  {
    ret[i] = key[i] ^ val[i];
  }
  return ret;
}
```

将用户输入的信息添加到异或后的key的末尾，并输入到md5函数中得到一个哈希住，作为下一轮的输入:

```
// 在 HMAC 中
unsigned char *pad_key = padKey(key, key_size, B);
  unsigned char *h1 = (unsigned char *)malloc((msg_size + B) * sizeof(unsigned
char));
  unsigned char *key1 = XOR(pad_key, ipad(B), B);
  // 拼接key和msg，大小为msg_size + B
  for (int i = 0; i < msg_size + B; i++)
  {
    h1[i] = 0;
    if (i < B)
    {
      h1[i] = key1[i];
    }
    else
    {
      h1[i] = msg[i - B];
    }
  }
```

```
unsigned char *out1 = md5_hmac_version(h1, B + msg_size);
```

得到 `out1` 之后，仅需要首先将padding后的key与`opad`异或，再将out1添加到异或结果的末尾，最后输入md5得到一个哈希值返回即可，如下:

```
unsigned char *h2 = (unsigned char *)malloc((16 + B) * sizeof(unsigned char));
  unsigned char *key2 = XOR(pad_key, opad(B), B);

  for (int i = 0; i < 16 + B; i++)
  {
    h2[i] = 0;
    if (i < B)
    {
      h2[i] = key2[i];
    }
    else
    {
      h2[i] = out1[i - B];
    }
  }

  return md5_hmac_version(h2, B + 16);s
```

四. 编译运行以及验证样例

关于验证样例，我参考了 rfc2104, 即HMAC的白皮书，以及rfc1321, 即MD5的白皮书中的样例，如下:

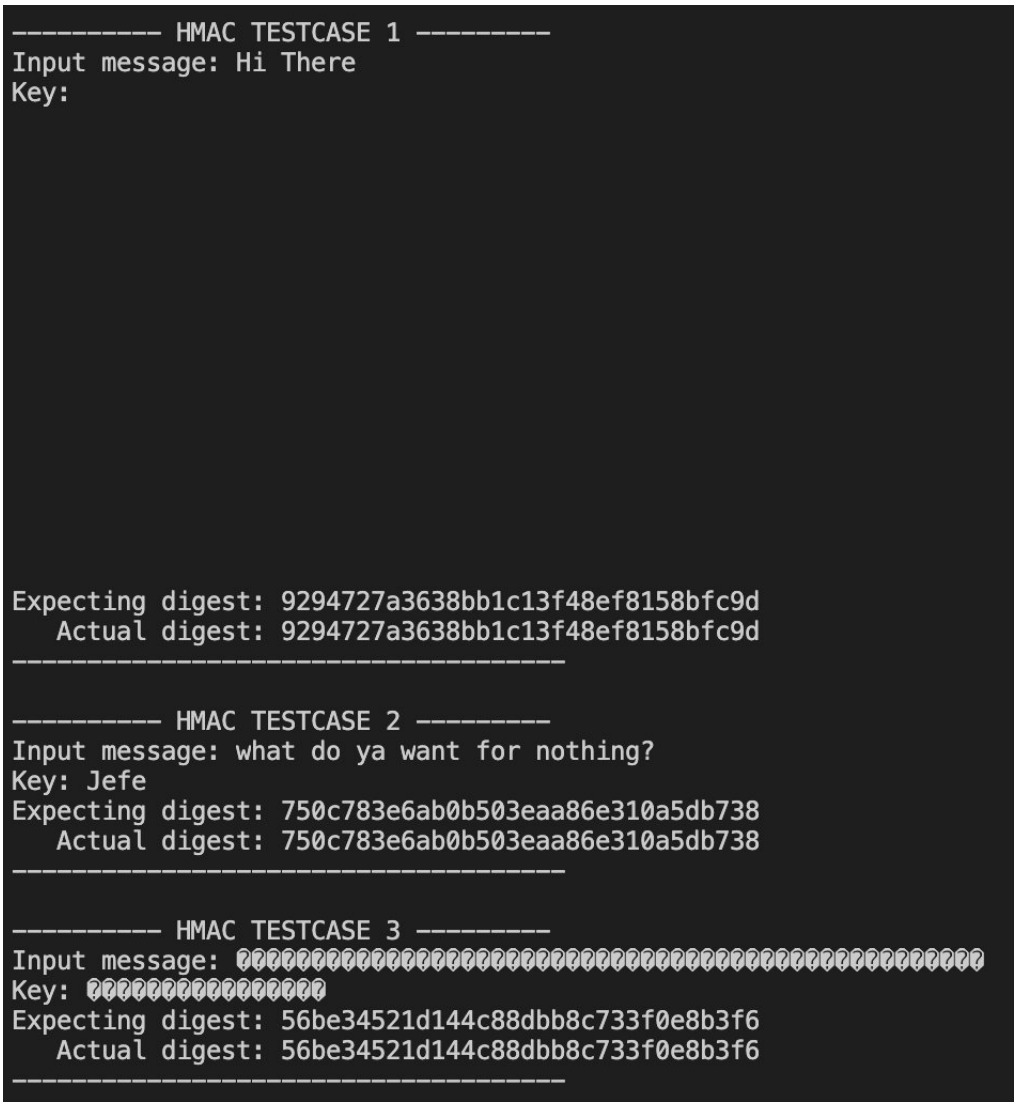
- **HMAC的三组样例**

```
key =         0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b      密钥
key_len =     16 bytes
data =        "Hi There"                                      消息
data_len =    8  bytes
digest =      0x9294727a3638bb1c13f48ef8158bfc9d   消息摘要(即结果)

key =         "Jefe"
data =        "what do ya want for nothing?"
data_len =    28 bytes
digest =      0x750c783e6ab0b503eaa86e310a5db738

key =         0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
key_len         16 bytes
data =          0xDDDDDDDDDDDDDDDDDDDD...
                ..DDDDDDDDDDDDDDDDDDDD...
                ..DDDDDDDDDDDDDDDDDDDD...
                ..DDDDDDDDDDDDDDDDDDDD...
                ..DDDDDDDDDDDDDDDDDDDD
data_len =      50 bytes
digest =        0x56be34521d144c88dbb8c733f0e8b3f6
```

- 我的代码的HMAC测试结果

输入: `make test`, 结果如下:



- **MD5 的7组样例**

```
MD5 ("") = d41d8cd98f00b204e9800998ecf8427e
MD5 ("a") = 0cc175b9c0f1b6a831c399e269772661
MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0
MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b
MD5 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789") =
d174ab98d277d9f5a5611c2c9f419d9f
MD5 ("12345678901234567890123456789012345678901234567890123456789012345678
78901234567890") = 57edf4a22be3c955ac49da2e2107b67a
```

- 我的MD5的测试结果

```
---------- MD5 TESTCASE 1 ----------
Input message:
Expecting digest: d41d8cd98f00b204e9800998ecf8427e
   Actual digest: d41d8cd98f00b204e9800998ecf8427e
------------------------------------

---------- MD5 TESTCASE 2 ----------
Input message: a
Expecting digest: 0cc175b9c0f1b6a831c399e269772661
   Actual digest: 0cc175b9c0f1b6a831c399e269772661
------------------------------------

---------- MD5 TESTCASE 3 ----------
Input message: abc
Expecting digest: 900150983cd24fb0d6963f7d28e17f72
   Actual digest: 900150983cd24fb0d6963f7d28e17f72
------------------------------------

---------- MD5 TESTCASE 4 ----------
Input message: message digest
Expecting digest: f96b697d7cb7938d525a2f31aaf161d0
   Actual digest: f96b697d7cb7938d525a2f31aaf161d0
------------------------------------

---------- MD5 TESTCASE 5 ----------
Input message: abcdefghijklmnopqrstuvwxyz
Expecting digest: c3fcd3d76192e4007dfb496cca67e13b
   Actual digest: c3fcd3d76192e4007dfb496cca67e13b
------------------------------------

---------- MD5 TESTCASE 6 ----------
Input message: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
Expecting digest: d174ab98d277d9f5a5611c2c9f419d9f
   Actual digest: d174ab98d277d9f5a5611c2c9f419d9f
------------------------------------

---------- MD5 TESTCASE 7 ----------
Input message: 12345678901234567890123456789012345678901234567890123456789012345678901234567890
Expecting digest: 57edf4a22be3c955ac49da2e2107b67a
   Actual digest: 57edf4a22be3c955ac49da2e2107b67a
------------------------------------
```