现代密码学第二次作业

计 34 2013011377 杨志灿

第一题

1. SHA-1 算法简述

SHA-1 杂凑算法以一长度小于 2^{64} 的 01 串（原文）为输入，输出是一个 160 位的杂凑值。

SHA-1 先将原文分成 512 位一组，其中最后一组的后 64 位是原文的长度（位数）。剩余的不足部分使用 100000…（1 后若干个零）补齐。

分组后，SHA-1 算法依次处理每个分组，迭代计算杂凑值，具体过程较复杂，可参考代码或 FIPS PUB 180-4 中的伪代码。

2. 寻找碰撞

考虑到寻找到碰撞的两个信息最好可以转换为 ASCII 码显示，故不能采用前一次的输出作为下一次的输入的方法（不能保证输入的是可见字符）。故最终决定采用随机生成原文的方式寻找碰撞。

在随机生成输入文本的时候使用了一个小技巧，就是使用固定的初始化随机种子。这样只需存储是生成的第几个输入，就可以还原出文本而不需记录文本，可以大大节约内存空间的使用。

3. 工程概述

SHA-1.h 中实现了 SHA-1 算法，其中

```
uint32_t* SHA1(const char * msg, bool print = false)
```

其中 msg 是字符数组格式的原文，print 表示是否输出中间过程。输出 5 个 32 位整型数，共同组成 160 位杂凑值。

值得注意的是，本次实验的输入文本都比较短，只有一个分组，迭代次数为 1，所以中间过程只有初始化和最终结果两个，而 SHA-1 算法的初始化状态是常数，就不在本文中赘述了。

```
uint32_t* SHA1(const bool * msg, uint64_t size, bool print = false)
```

其中 msg 是 bool 数组格式的原文，size 表示原文的位数，print 表示是否输出中间过程。输出 5 个 32 位整型数，共同组成 160 位杂凑值。

main.cpp 是使用 SHA-1 算法计算文本的杂凑值，运算结果见第 4 节。

collision.cpp 是寻找碰撞的程序，其中

```
void find_collision(int len = 10)
```

find_collision 寻找杂凑值前 50 位相同的碰撞对，len 表示随机文本的长度，寻找到碰撞对会输出碰撞对的生成编号。根据这两个生成编号，可以重新调用如下函数还原文本。

```
void print_msg(unsigned int id, int len = 10)
```

其中 id 表示生成编号，len 表示生成的文本长度。

4. 实验结果

Plaintext	SHA-1
yangzhican2013011377	70d6cde6226cf95cf7a0e5b4a6eca6168e7b3cac
yezipeng2013011404	19f45aa3b28665b0fdbe3480408e3d0d581c2ff4
D2/;q<lVQq	6179f4612671b706258304541701437890943446
e,N>rF 5es	6179f4612671b15d162c5282f97a8ee4e222ed77

后两个文本是寻找到的长度为 10 的一对碰撞对，他们的杂凑值的前 50 位是相同的。

第二题

1. AES 加密算法简述

AES 加密算法是一种对称加密算法，同时 AES 也是一种分组密码，分组长度为 128 比特。常见的 AES 有三种不同的密钥和对应的轮数，分别是：

AES-128：128 比特密钥/10 轮

AES-192：192 比特密钥/12 轮

AES-256：256 比特密钥/14 轮

此次实现的是 AES-128，即密钥与分组均为 128 比特。

在将原文分组后，组与组之间的加密有多种操作模式，常见的有 ECB、CBC、CFB、OFB、CTR 等。因此次加密的原文只有一个分组，故不用考虑操作模式的问题。

AES 的操作大多以 128 比特为单位，且划分为 4*4 个 8 比特字称为状态。首先要生成 10 个轮密钥，包括初始密钥在内则共有 11 个轮密钥。生成轮密钥后，进行十次循环，每次使用一个轮密钥并进行相关操作(SubBytes, ShiftRows, MixColumns, AddRoundKeys)。第一轮使用原文作为输入，此后每轮接着使用上一轮的输出作为输入，最终输出密文。

2. 工程概述

AES.h 中实现了 AES 加密算法，其中比较重要的函数有：

```
uint8_t* cipher(const uint8_t* plaintext, const uint8_t key[4*Nk])
```

其中 plaintext 是 128 比特的原文，以字为单位传入，如本题的原文输入即为：

```
const uint8_t plaintext[16] =
{0x32,0x43,0xF6,0xA8,0x88,0x5A,0x30,0x8D,0x31,0x31,0x98,0xA2,0xE0,
0x37,0x07,0x34};
```

key 是 128 比特的密钥，本题中的密钥即为：

```
const uint8_t key[16] =
{0x3A,0xE1,0x15,0x62,0xA8,0xF3,0xC7,0x1A,0x2B,0xF6,0xDF,0xA1,0x50,
0x9B,0xCA,0xF1};
```

输出是 128 比特的密钥，以 16 个字为格式输出。

除了 AES 主算法，AES.h 中还提供了一些中间函数的接口，如扩展轮密钥的函数 keyExpansion：

```
void keyExpansion(const uint8_t key[4*Nk], uint8_t w[Nr+1][Nb][4])
```

以密钥 key 作为输入，输出 11 个轮密钥 w，轮密钥的格式是 4*4 的 8 比特字，也即 AES 主算法使用的状态表示。

同时也可以直接使用轮密钥作为密钥输入来进行 AES 加密：

```
uint8_t* cipher(const uint8_t* plaintext, const uint8_t w[Nr+1][Nb][4])
```

3. 实验结果

```
Plaintext:
3243f6a8885a308d313198a2e0370734

Key:
3ae11562a8f3c71a2bf6dfa1509bcaf1

Round Keys:
Round  0: 3ae11562a8f3c71a2bf6dfa1509bcaf1
Round  1: 2f95b4318766732bac90ac8afc0b667b
Round  2: 06a6958181c0e6aa2d504a20d15b2c5b
Round  3: 3bd7acbfba174a1597470035461c2c6e
Round  4: afa633e515b179f082f679c5c4ea55ab
Round  5: 385a51f92deb2809af1d51cc6bf70467
Round  6: 70a8d4865d43fc8ff25ead4399a9a924
Round  7: e37be268be381ee74c66b3a4d5cf1a80
Round  8: e9d92f6b57e1318c1b878228ce4898a8
Round  9: a09fede0f77edc6cecf95e4422b1c6ec
Round 10: 5e2b2373a955ff1f45aca15b671d67b7

Cryptotext:
d448fce815633ad1b43a7ae2489a2a69
```

第三题

1. GCM 认证加密算法简述

GCM 认证加密算法的实现主要参考的是《The Galois/Counter Mode of Operation (GCM)》[1]，与课堂讲解的有些许不同，但算法本质是一样的。

GCM 认证加密算法的输入包括原文 P(Plaintext),密钥 K(secret Key)，初始向量 IV(Initial Vector)以及附加认证消息 AAD(Additional Authenticated Data)。输出包括密文 C(Ciphertext)和认证标签 T(Tag)。

在本次实验中，加密算法使用第二问中的 AES-128，且只有附加认证消息，即原文 P 为空，此时自然有密文 C 也为空。这种情况下的 GCM 认证加密算法只有认证的功能，此时 GCM 即为 GMAC 认证算法。

2. 工程概述

首先，uint128_t.h 中封装了一种数据类型 uint128_t，顾名思义，就是一个 128 比特的数据，支持一些正常的位运算以及 GMAC 算法中涉及的操作，还有一些接口上的转换，如 uint128_t 与 uint8_t 指针之间的双向转换等。

GMAC.h 中实现了 GMAC 认证算法，其中最重要的是两个 encrypt 函数：

```
uint128_t encrypt(const bool * A, uint64_t lenA, const uint8_t IV[12],
const uint8_t key[16], bool print = false)
```

其中 A 是比特格式的附加认证消息,lenA 是附加认证消息的位数,IV 是初始向量，key 是密钥，print 表示是否打印中间过程信息。输出 128 位的认证标签。

```
uint128_t encrypt(const char * A, const uint8_t IV[12], const uint8_t
key[16], bool print = false)
```

这个 encrypt 函数接受另一种格式的附加认证消息：即字符串格式。

3. 实验结果

H:
66e94bd4ef8a2c3b884cfa59ca342b2e

E(K, Y0):
58e2fccefa7e3061367f1d57a4e7455a

GHASH(H, A, C):

[1] http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

00000000000000000000000000000000

T:
58e2fccefa7e3061367f1d57a4e7455a

Key:
3ae11562a8f3c71a2bf6dfa1509bcaf1

A:
yangzhican2013011377

H:
aba31e4ecb741bf0ce521dbf7e2d77f7

E(K, Y0):
80521c59ee0c2766f91fddc7b8dc9b79

GHASH(H, A, C):
4d5259507a686963616e323031333031

T:
cd00450994644e059871eff789efab48

即以“yangzhican2013011377”作为认证消息输入的话，输出的 tag 是 0xcd00450994644e059871eff789efab48。