



创新创业实践汇总报告

年级：2021 级

姓名：秦成钰 小组：56

院 系：网络空间安全

专 业：密码科学与技术

2020 年 8 月 1 日

目录

1Project1	4
1.1 简介	
1.2 实验用到的小技巧	
1.3 操作流程	
1.4 实验结果	
2Project2	9
2.1 简介	
2.2 实验用到的小技巧	
2.3 操作流程	
2.4 实验结果	
3Project3	12
3.1 简介	
3.2 实验用到的小技巧	
3.3 操作流程	
3.4 实验结果	
4Project4	15
4.1 简介	
4.2 实验用到的小技巧	
4.3 操作流程	
4.4 实验结果	
5Project5	18
5.1 简介	
5.2 实验用到的小技巧	
5.3 操作流程	
5.4 实验结果	
6Project9	21
6.1 简介	
6.2 实验用到的小技巧	
6.3 操作流程	
6.4 实验结果	
7Project10	24
7.1 简介	
7.2 实验用到的小技巧	
7.3 操作流程	
7.4 实验结果	
8Project11	28
8.1 简介	

8.2 实验用到的小技巧	
8.3 操作流程	
8.4 实验结果	
9Project13	3 错误！未定义书签。
9.1 简介	
9.2 实验用到的小技巧	
9.3 操作流程	
9.4 实验结果	
10Project18	34
10.1 简介	
10.2 实验用到的小技巧	
10.3 操作流程	
10.4 实验结果	
11Project19	34
11.1 简介	
11.2 实验用到的小技巧	
11.3 操作流程	
11.4 实验结果	
12Project22	38
12.1 简介	
12.2 实验用到的小技巧	
12.3 操作流程	
12.4 实验结果	
以上 Project 均由秦成钰个人完成	
附录	41
后记	41

Project1:implement the naïve birthday attack of reduced SM3

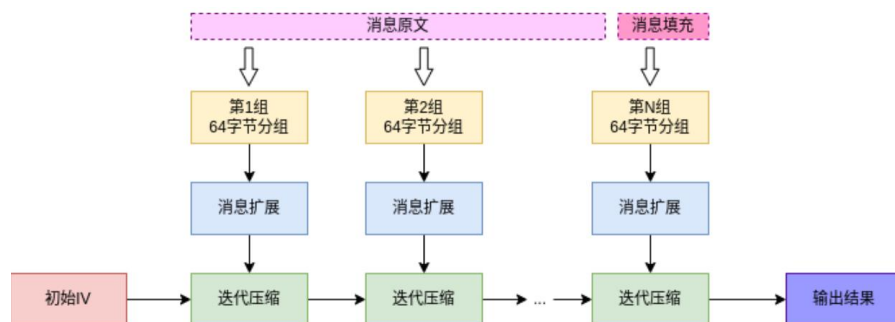
1:简介

生日攻击 (birthday attack) 是利用概率论中的生日问题, 找到冲突的 Hash 值, 伪造报文, 使身份验证算法失效. 可以用数学推导得生日攻击成功率大于一半时需要的次数。

$$Q(H) \approx \sqrt{\frac{\pi}{2}H}$$

可见生日攻击对于密文空间较小的算法具备良好的攻击能力。

SM3 是一种密码杂凑算法, 用于替代 MD5/SHA-1/SHA-2 等国际算法, 适用于数字签名和验证、消息认证码的生成与验证以及随机数的生成, 可以满足电子认证服务系统等应用需求, 于 2010 年 12 月 17 日发布。它是在 SHA-256 基础上改进实现的一种算法, 采用-结构, 消息分组长度为 512bit, 输出的摘要值长度为 256bit。在本次实验中为了方便, 我们采用简化的 sm3 进行生日攻击。



原理图如上所示

关键词：生日攻击，sm3，杂凑算法

2. 实现中用到的小技巧和创新的点：

实验中使用了可以自己选取长度 sm3。

```
int n;//前n bit
cout << "请输入大小，以bit为单位:";
cin >> n;

int num = pow(2, n);
```

为了更好的完成，进行了消息填充

```
//消息填充
string padding(string str)
{
    int n;//输入消息长度
    n = str.length();
    int r_n = n;//记录原始消息长度
    int k = 0;
    while ((n + 1 + k) % 512 != 448)
    {
        k += 1;
        str += '0';//填充0
    }
    //n = n + 1 + k;//现在的消息长度
    string llen = dec_binget(r_n);
    int llength = llen.size();
    for (int i = 0; i <= (64 - llength); i++)
    {
        llen = '0' + llen;
    }
    str = str + llen;//填充完成
    //cout << str;
    return str;
}
```

3. 操作流程:

调入相关库函数。

```
#include <iostream>
#include <string>
#include <cmath>
```

输入 sm3。

```
int n;//前n bit
cout << "请输入大小，以bit为单位:";
cin >> n;
```

开始计时。

```
clock_t start, end;
start = clock();
```

进行碰撞和生日攻击,当相同后退出循环。

```
int M1 = rand() % (num + 1);
int M2 = rand() % (num + 1);

while (SM3(dec_binget(M1)).substr(0, n / 4) != SM3(dec_binget(M2)).substr(0, n / 4))
{
    M1 = rand() % (num + 1);
    M2 = rand() % (num + 1);
}
```

计算时间并退出程序。

```
end = clock();
cout << "成功时间消耗为" << (float)(end - start) * 1000 / CLOCKS_PER_SEC << "ms";
return 0;
```

Sm3 的部分函数如下:

迭代压缩函数:

```

//迭代压缩函数
string iteration(string str)
{
    int num = str.size() / 128;
    string V = "7380166F4914B2B9172442D7DA8A0600A96F30BC163138AAE38DEE4DB0FB0E4E";
    //string v = hex_bin(V);
    string B = "", extensionB = "", compressB = "";
    for (int i = 0; i < num; i++)
    {
        B = str.substr((i * 128), 128);
        extensionB = extension(B);
        compressB = compress(extensionB, V);
        V = XOR(compressB, V);
    }
    return V;
}

```

模运算

```

//模运算
string Mod(string str1, string str2)
{
    string res1 = hex_binget(str1);
    string res2 = hex_binget(str2);
    char temp = '0';
    string res = "";
    for (int i = res1.size() - 1; i >= 0; i--)
    {
        res = ccxor(ccxor(res1[i], res2[i]), temp) + res;
        if (ccand(res1[i], res2[i]) == '1')
        {
            temp = '1';
        }
        else
        {
            if (ccand(res1[i], res2[i]) == '1')
            {
                temp = ccxor('1', temp);
            }
            else
            {
                temp = '0';
            }
        }
    }
}

```

与操作：

```
//与操作
string AND(string str1, string str2)
{
    string answer = "";
    str1 = hex_binget(str1);
    str2 = hex_binget(str2);
    for (int k = 0; k < str1.size(); k++)
    {
        if (str1[k] == '1' && str2[k] == '1')
        {
            answer += '1';
        }
        else
        {
            answer += '0';
        }
    }
    return bin_hexget(answer);
}
```

4. 实现结果：

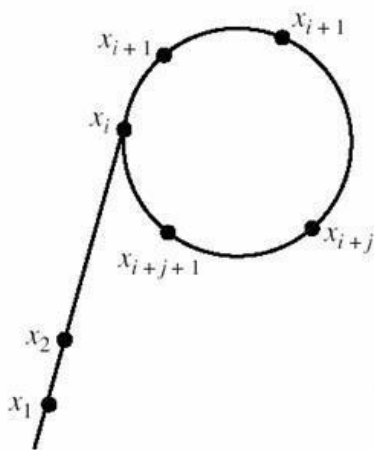
请输入大小，以bit为单位:8
成功时间消耗为9959ms

时间消耗为 9959ms。经过本次实验认识到安全的杂凑杂凑算法应该具有一定的输出空间，从而更好的抵抗生日攻击。

Project2:implement the Rho method of reduced SM3

1:简介

Rho 算法其实是一种概率上的算法，虽然是靠概率，但是其准确率非常高，更重要的是，该算法效率高。其主要基于密码学其中的一个“生日悖论”来进行算法的设计，利用了循环迭代的思想。



Rho 算法的原理图如上所示。

SM3 是一种密码杂凑算法，用于替代 MD5/SHA-1/SHA-2 等国际算法，适用于数字签名和验证、消息认证码的生成与验证以及随机数的生成，可以满足电子认证服务系统等应用需求，于 2010 年 12 月 17 日发布。它是在 SHA-256 基础上改进实现的一种算法，采用-结构，消息分组长度为 512bit，

输出的摘要值长度为 256bit。在本次实验中为了方便，我们采用简化的 sm3 进行 Rho 攻击。

关键词： sm3 Rho 简化

2. 实现中用到的小技巧和 innovation 点：

本实验通过移位来实现转换

```
int reversebytes32(int message) {  
    return (message & 0x000000FFU) << 24 | (message & 0x0000FF00U) << 8 |  
           (message & 0x00FF0000U) >> 8 | (message & 0xFF000000U) >> 24;  
}
```

本实验用到了循环往复进行查找的实验思想

```
while (2) {  
    memcpy(hash_val3, hash_val2, 32);  
    sm3((char*)hash_val2, hash_val2);  
    if (!memcmp(hash_val0, hash_val2, 2)) {  
        cout << "碰撞到的信息为: " << endl;  
        dump_buffer((char*)hash_val3, 32);  
        cout << "碰撞到的信息的哈希值为: " << endl;  
        dump_buffer((char*)hash_val2, 32);  
        break;  
    }  
}
```

3. 操作流程

调入相关库函数

```
#include<iostream>  
#include<bitset>  
#include<fstream>  
#include<sstream>
```

对程序计时并输出相关时间

```

clock_t startTime = clock();
sm3_rho_collide();
clock_t endTime = clock();
cout << "整个程序用时为" << (double(endTime - startTime) / CLOCKS_PER_SEC) * 1000 << "ms" ;

```

Rho 攻击过程

```

int sm3_rho_collide() {
    int hash_val0[8];
    int hash_val2[8];
    int hash_val3[8];

    cout << "待碰撞信息为: ";
    char plaintext[] = "test";
    cout << plaintext << endl;
    sm3(plaintext, hash_val0);
    cout << "待碰撞信息的哈希值为: " << endl;
    dump_buffer((char*)hash_val0, 32);

    sm3(plaintext, hash_val2);
    while (2) {
        memcpy(hash_val3, hash_val2, 32);
        sm3((char*)hash_val2, hash_val2);
        if (!memcmp(hash_val0, hash_val2, 2)) {
            cout << "碰撞到的信息为: " << endl;
            dump_buffer((char*)hash_val3, 32);
            cout << "碰撞到的信息的哈希值为: " << endl;
            dump_buffer((char*)hash_val2, 32);
            break;
        }
    }

    cout << "可以碰撞" << 2 * 8 << "bit的信息" << endl;
    return 0;
}

```

4. 实现结果

```

待碰撞信息为: test
待碰撞信息的哈希值为:
03 0A CB BE 09 F7 56 5C 0E E7 21 06 78 9E 5A 20 9B 68 61 A6 0D 1E 8E 9D E0 B0 30 DA C1 AA EE 3C
碰撞到的信息为:
0C F5 92 8D C2 E1 61 45 2C BF 30 85 E3 86 6F 47 4A BB 7B D1 8E F5 68 9F 3B A2 22 FD B2 9E B1 55
碰撞到的信息的哈希值为:
03 0A B4 C6 87 8B CC 8C 61 3F B9 07 51 DE F6 32 98 2B 21 3F FC 98 DC C9 E1 55 D2 88 4D 03 67 2F
可以碰撞16比特的信息
整个程序用时为72ms请按任意键继续. . .

```

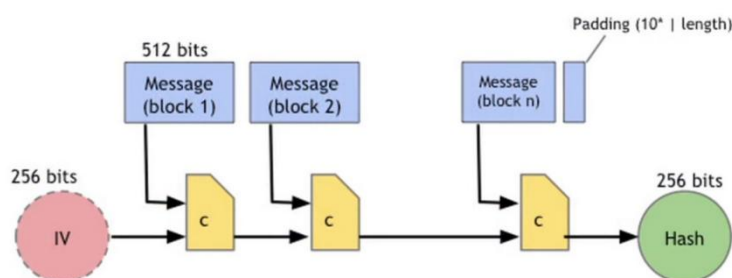
Project3:implement length extension attack for SM3, SHA256, etc.

1:简介

长度扩展攻击（length extension attack），是指针对某些允许包含额外信息的加密散列函数的攻击手段。对于满足以下条件的散列函数，都可以作为攻击对象：1. 加密前将待加密的明文按一定规则填充到固定长度的倍数；2. 按照该固定长度，将明文分块加密，并用前一个块的加密结果，作为下一块加密的初始向量（Initial Vector）。Merkle - Damgård 散列函数满足上列要求。

SHA-2（Secure Hash Algorithm 2）是一种密码散列函数算法标准，由美国国家安全局研发，属于 SHA 算法之一。

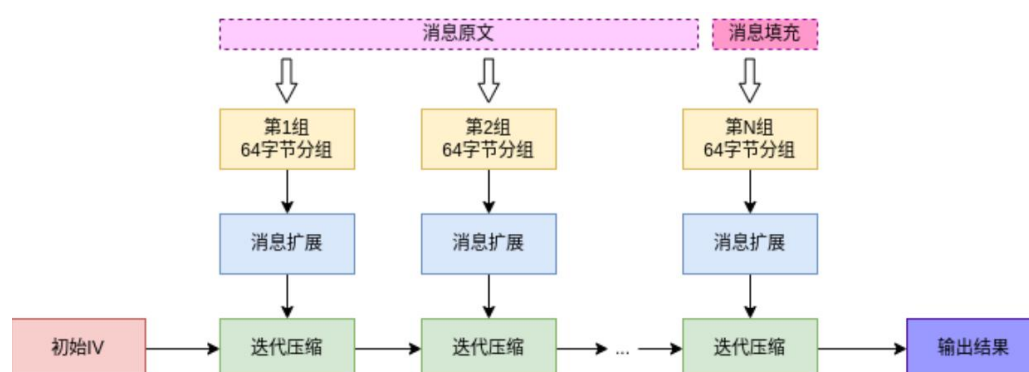
SHA-256 hash function



Theorem: If c is collision-free, then SHA-256 is collision-free.

原理图如上所示。

SM3 是一种密码杂凑算法，用于替代 MD5/SHA-1/SHA-2 等国际算法，适用于数字签名和验证、消息认证码的生成与验证以及随机数的生成，可以满足电子认证服务系统等应用需求，于 2010 年 12 月 17 日发布。它是在 SHA-256 基础上改进实现的一种算法，采用 π -结构，消息分组长度为 512bit，输出的摘要值长度为 256bit。



原理图如上所示

关键词: sm3 sha-256 长度扩展攻击

3. 实现中用到的小技巧和创新的点:

```
int sm3_length_attack(char* memappend, int* hash_val, int length_for_memappend, int length_for_message) {
    int new_hash_val[8];
    memcpy(IV, hash_val, 32);
    sm3forattack(memappend, new_hash_val, length_for_memappend, length_for_message);
    cout << "长度扩展攻击得到的哈希值" << endl;
    dump_buffer((char*)new_hash_val, 32);
    return 0;
}
```

本实验中，用到了在尾部添加新的信息来进行长度扩展攻击

3. 操作流程:

主函数调用如下

```
int main() {
    int hash_val1[8];
    int hash_val2[8];
    char mingwen[67];
    char plaintext[] = "202100210057";
    char testappend[] = "get";

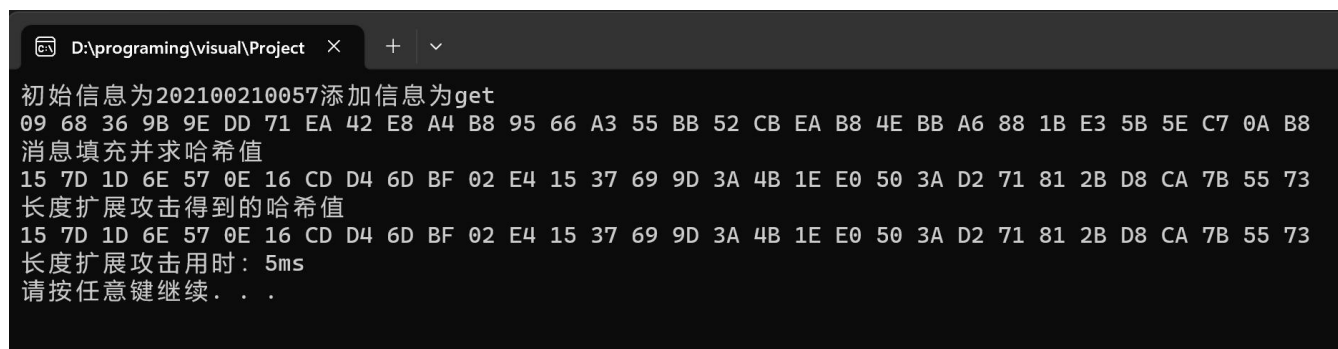
    cout << "初始信息为" << plaintext << "添加信息为" << testappend << endl;
    sm3(plaintext, hash_val1, 3);
    dump_buffer((char*)hash_val1, 32);
    for (int i = 0; i < 8; i++) {
        hash_val2[i] = reversebytes_uint32t(hash_val1[i]);
    }

    clock_t startTime = clock();
    bit_stuffing(plaintext, 3);
    memcpy(mingwen, plaintext_after_stuffing, 64);
    memcpy(&mingwen[64], testappend, 3);
    sm3(mingwen, hash_val1, 67);
    cout << "消息填充并求哈希值" << endl;
    dump_buffer((char*)hash_val1, 32);
    sm3_length_attack(testappend, hash_val2, 3, 64);
    clock_t endTime = clock();
    cout << "长度扩展攻击用时: " << (double(endTime - startTime) / CLOCKS_PER_SEC) * 1000 << "ms" << endl;
    system("pause");
    return 0;
}
```

Sm3 长度扩展攻击

```
int sm3_length_attack(char* memappend, int* hash_val, int length_for_memappend, int length_for_message) {
    int new_hash_val[8];
    memcpy(IV, hash_val, 32);
    sm3forattack(memappend, new_hash_val, length_for_memappend, length_for_message);
    cout << "长度扩展攻击得到的哈希值" << endl;
    dump_buffer((char*)new_hash_val, 32);
    return 0;
}
```

4. 实现结果



```
D:\programing\visual\Project > .\Project.exe
初始信息为202100210057添加信息为get
09 68 36 9B 9E DD 71 EA 42 E8 A4 B8 95 66 A3 55 BB 52 CB EA B8 4E BB A6 88 1B E3 5B 5E C7 0A B8
消息填充并求哈希值
15 7D 1D 6E 57 0E 16 CD D4 6D BF 02 E4 15 37 69 9D 3A 4B 1E E0 50 3A D2 71 81 2B D8 CA 7B 55 73
长度扩展攻击得到的哈希值
15 7D 1D 6E 57 0E 16 CD D4 6D BF 02 E4 15 37 69 9D 3A 4B 1E E0 50 3A D2 71 81 2B D8 CA 7B 55 73
长度扩展攻击用时: 5ms
请按任意键继续...
```

时间为 5ms

Project4:do your best to optimize SM3 implementation (software)

1:简介

SM3 是一种密码杂凑算法，用于替代 MD5/SHA-1/SHA-2 等国际算法，适用于数字签名和验证、消息认证码的生成与验证以及随机数的生成，可以满足电子认证服务系统等应用需求，于 2010 年 12 月 17 日发布。它是在 SHA-256 基础上改进实现的一种算法，采用 Merkle-Damgård 结构，消息分组长度为 512bit，输出的摘要值长度为 256bit。

SIMD(Single Instruction Multiple Data)单指令多数据流，能够读取多个操作数，并把它们打包在大型寄存器的一组指令集。一次获取多个操作数后，存放于一个大型寄存器，再进行运算，从而达到一条指令完成对多个对象计算的效果，实现加速。目前常见编译器对 X86-64 的 CPU 上 128bit 的 SIMD 计算支持比较好，基本对于大多简单的计算都可以做到使用 SIMD 做一个简单的优化，但是对于较为复杂的操作依旧需要手动编写相应的 C/C++或者汇编代码。

关键词：SM3 SIMD

2. 实现中用到的小技巧和创新点：

本实验采取了 SIMD 对 sm3 进行了优化


```

for (j = 4; j < 17; j++)//基于SIMD对消息扩展进行优化
{
    int k = j * 4 + 3;
    tmp10 = _mm_setr_epi32(W[j * 4 - 16], W[j * 4 - 15], W[j * 4 - 14], W[j * 4 - 13]);
    tmp4 = _mm_setr_epi32(W[j * 4 - 13], W[j * 4 - 12], W[j * 4 - 11], W[j * 4 - 10]);
    tmp5 = _mm_setr_epi32(W[j * 4 - 9], W[j * 4 - 8], W[j * 4 - 7], W[j * 4 - 6]);
    tmp6 = _mm_setr_epi32(W[j * 4 - 3], W[j * 4 - 2], W[j * 4 - 1], 0);
    tmp7 = _mm_setr_epi32(W[j * 4 - 6], W[j * 4 - 5], W[j * 4 - 4], W[j * 4 - 3]);
    tmp8 = _mm_xor_si128(left(tmp4, 7), tmp7);
    tmp1 = _mm_xor_si128(tmp10, tmp5);
    tmp2 = left(tmp6, 15);
    tmp1 = _mm_xor_si128(tmp1, tmp2);
    tmp3 = _mm_xor_si128(tmp1, _mm_xor_si128(left(tmp1, 15), left(tmp1, 23)));
    temp[j] = _mm_xor_si128(tmp3, tmp8);
    _mm_maskstore_epi32((int*)&W[j * 4], ze, temp[j]);
    W[k] = P_1(W[k - 16] ^ W[k - 9] ^ (rotate_left(W[k - 3], 15))) ^ (rotate_left(W[k - 13], 7)) ^ W[k - 6];
}

```

3. 操作流程

定义相关变量和输入信息

```

int main(int argc, char* argv[])
{
    unsigned char Hash[32] = { 0 };
    unsigned char str[] = "202100210057";
}

```

进行调用并输入时间

```

clock_t start, stop;
start = clock();
for (int i = 1; i < 10000; i++) {
    SM3(str, len, Hash);
}
stop = clock();
printf("time=%f\n", ((double)(stop - start)) / (CLK_TCK * 10000));
printf("单位s");
return 0;

```

Sm3 如下

```

int SM3(unsigned char* msg, unsigned int msglen, unsigned char* out_hash)
{
    SM3_Init();
    Block(msg, msglen);
    intToString(out_hash);
    return 1;
}

```


Block 函数

```
void Block(unsigned char* message, unsigned int msglen) {
    int i;
    int left = 0;
    unsigned long long total = 0;

    for (i = 0; i < msglen / 64; i++) {
        memcpy(message_buffer, message + i * 64, 64);
        CF(message_buffer);
    }

    total = msglen * 8;
    left = msglen % 64;
    memset(&message_buffer[left], 0, 64 - left);
    memcpy(message_buffer, message + i * 64, left);
    message_buffer[left] = 0x80;
    if (left <= 55) {
        for (i = 0; i < 8; i++)
            message_buffer[56 + i] = (total >> ((8 - 1 - i) * 8)) & 0xFF;
        CF(message_buffer);
    }
    else {
        CF(message_buffer);
        memset(message_buffer, 0, 64);
        for (i = 0; i < 8; i++)
            message_buffer[56 + i] = (total >> ((8 - 1 - i) * 8)) & 0xFF;
        CF(message_buffer);
    }
}
```

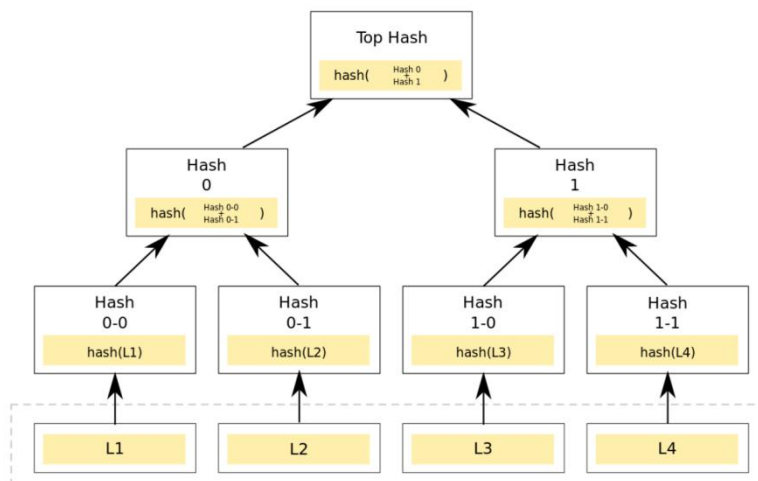
4. 实现结果

time=0.000003
单位s

Project5: Impl Merkle Tree following RFC6962

1:简介

Merkle Tree, 通常也被称作 **Hash Tree** 就是存储 hash 值的一棵树。Merkle 树的叶子是数据块(例如, 文件或者文件的集合)的 hash 值。非叶节点是其对应子节点串联字符串的 hash。Merkle Tree 和 Hash List 的主要区别是, 可以立即验证 Merkle Tree 的一个分支。因为可以将文件切分成小的数据块, 这样如果有一块数据损坏, 仅仅重新下载这个数据块就行了。如果文件非常大, 那么 Merkle tree 和 Hash list 都很到, 但是 Merkle tree 可以一次下载一个分支, 然后立即验证这个分支。



Merkle Tree 如上所示

RFC 文件格式作为 ARPA 网计划的基础起源于 1969 年。RFC 文件只能增加, 不会取消或中途停止发行。新的 RFC 文件可以声明取代旧的 RFC 文件。

RFC 文件是纯 ASCII 文字档格式。RFC 文件是由 Internet Society 审核后给定编号并发行

关键词：RFC6962 Merkle tree

2. 实现中用到的小技巧和创新点：

为了方便模拟本实验采取了 to string 函数来模拟哈希

```
string Hash(string str)
{
    size_t n = h(str);
    string Newstring = to_string(n);
    return Newstring;
}
```

3. 操作流程

Merkle Tree 初始状态如下：

```
struct Merkletree
{
    int left = -1;
    int right = -1;
    int parent = -1;
    int depth = -1;
    int num = -1;
    string hash_i;
};
```

主函数调用如下：先生成 merkletree，然后判断是否成功

```
Merkle_tree();  
if (MT_proof(4))  
    cout << "Merkle_tree_proof成功!" << endl;  
else cout << "Merkle_trtree_proof 失败!" << endl;  
  
if (MT_not_proof(5.5))  
    cout << "MT_not_proof 成功" << endl;  
else cout << "MT_not_proof 失败!" << endl;  
return 0;
```

Merkletree 参数如下

```
const int num = 200000;  
int r = 0;  
int q = -1;  
int p = -1;  
int num1 = num;  
int num2 = 0;  
int tdepth = 0;  
Merkletree arr[2 * num];
```

增加深度的函数如下:

```
void Change_depth(Merkletree* tmp)  
{  
    if (tmp->left == -1 && tmp->right == -1)  
    {  
        tmp->depth++;  
        return;  
    }  
    Change_depth(&(arr[tmp->left]));  
    tmp->depth++;  
    Change_depth(&(arr[tmp->right]));  
}
```

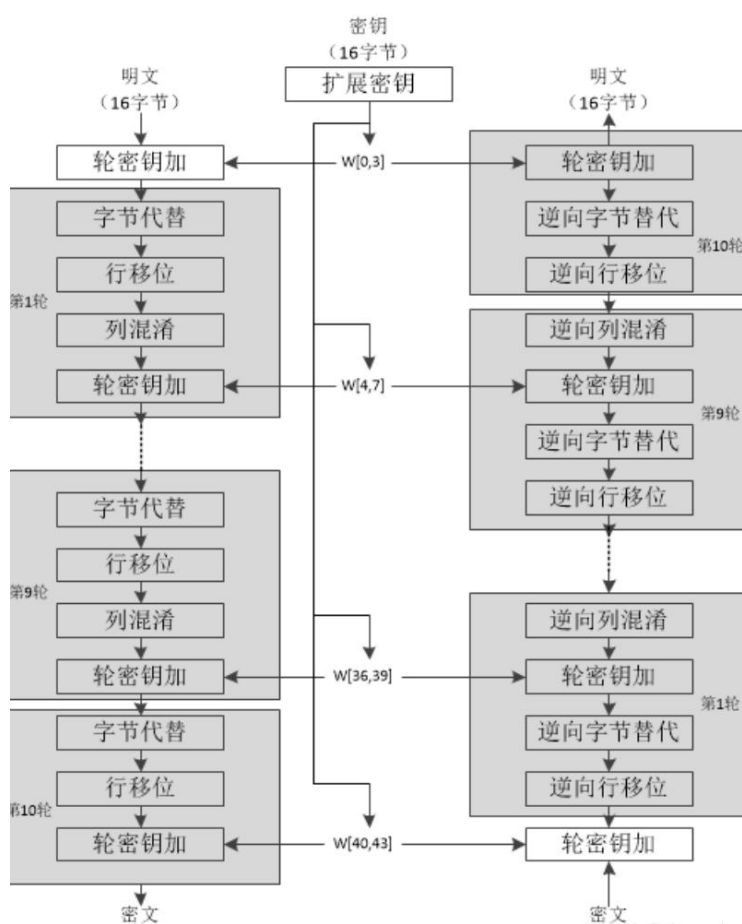
4. 实现结果

```
Merkle_tree_proof成功!  
MT_not_proof 成功
```

Project9: AES / SM4 software implementation

1:简介

AES 算法是对称分组密码算法。数据分组长度必须是 128 bits，使用密钥长度为 128，192 或 256 bits。对于三种不同密钥长度的 AES 算法，分别称为“AES-128”、“AES-192”、“AES-256”。AES 加密算法涉及 4 种操作：字节替代（SubBytes）、行移位（ShiftRows）、列混（MixColumns）和轮密钥加（AddRoundKey）。



AES 加解密流程如上所示

SM4 算法是我国商用密码算法。SM4 算法是一个分组加密算法，分组长度和密钥长度均 128bit。SM4 算法使用 32 轮的非线性迭代结构。SM4 在最后一轮非线性迭代之后加上了一个反序变换，因此 SM4 中只要解密密钥是加密密钥的逆序，它的解密算法与加密算法就可以保持一致。SM4 的主体运算是非平衡 Feistel 结构。

关键词：AES SM4

2. 实现中用到的小技巧和 innovation 点：

```
void AES::invSubByte(word in[]) {  
    int i, j;  
    byte L, R;  
    for (i = 0; i < 4; i++) {  
        for (j = 0; j < 4; j++) {  
            L = in[i].wordKey[j] >> 4;  
            R = in[i].wordKey[j] & 0x0f;  
            in[i].wordKey[j] = invSBox[L][R];  
        }  
    }  
}
```

通过简单移位进行直接替代提高执行速度

3. 操作流程

导入相关函数库

```
#include <iostream>  
#include <cstdlib>  
#include <stdio.h>
```

主函数调用如下

```

int main(int argc, char const* argv[])
{
    int i;
    //设置明文和密钥
    byte plaintext[16], keyword[16];
    for (i = 0; i < 16; i++) {
        plaintext[i] = byte(i + 2);
        keyword[i] = 0x01;
    }

    AES Aes;
    Aes.setPlainText(plaintext);
    Aes.setCipherKey(keyword);
    Aes.encryption();
    Aes.decryption();
    Aes.messageget();
    return 0;
}

```

解密函数如下

```

//解密
void AES::decryption() {
    int i, j, k;
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            deCipherText[i].wordKey[j] = cipherText[i].wordKey[j];
        }
    }

    addRoundKey(deCipherText, 10);
    for (i = 9; i > 0; i--) {
        invShiftRows(deCipherText);
        invSubByte(deCipherText);
        addRoundKey(deCipherText, i);
        invMixColumn(deCipherText);
    }

    invShiftRows(deCipherText);
    invSubByte(deCipherText);
    addRoundKey(deCipherText, 0);
}

```

4. 实现结果

```

plainText:
2 3 4 5 6 7 8 9 a b c d e f 10 11
wordKey:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 7c 7d 7d 7d 7d 7c 7c 7c 7c 7d 7d 7c 7c 7c 6e 6d 6d 82 13 11 11 fe 6f 6c 6c 83 12 1
0 10 ff a0 a7 7b 4b b3 b6 6a b5 dc da 6 36 ce ca 16 c9 dc 2c 6a c 6f 9a 0 b9 b3 40 6 8f 7d 8a 10 46 b2 2a fc 3f dd b0 fc
86 6e f0 fa 9 13 7a ea 4f 48 61 b4 8e 95 d1 48 8 fb 21 b2 1 e8 5b 58 4e 31 c7 57 d9 a4 16 1f d1 5f 37 ad d0 b7 6c f5 9e
e1 ed 90 bc 45 fb 8f 6d 1a cc 22 bd ad a0 d7 23 1a 2f 7a e5 5f d4 f5 88 45 18 d7 35 e8 b8 0 16 40 80 f1 b2 1f 54 4 3a 5
a 4c d3 f b2 f4 d3 19
cipherText:
f9 26 95 ce 16 8b b 99 99 87 b5 58 2 ed 8e d4
deCipherText:
2 3 4 5 6 7 8 9 a b c d e f 10 11

```

Project10:report on the application of this deduce technique in Ethereum with ECDSA

1:简介

椭圆曲线数字签名算法 (ECDSA)，主要用于对文件创建数字签名，可以在不破坏源文件的情况下进行验证等。ECDSA 处理过程：

1. 所有通信方使用相同的全局参数，用于定义椭圆曲线以及曲线上的基点。
2. 签名者生成一对公私钥。选择一个随机数或者伪随机数作为私钥，利用随机数和基点算出另一点，作为公钥。
3. 对消息计算 Hash 值，用私钥、全局参数和 Hash 值生成签名
4. 验证者用签名者的公钥、全局参数等验证。

以太坊（Ethereum）是由 Vitalik Buterin 所创建，一种允许智能合约和去中心化应用程序在其网络上运行的加密货币。

关键词：ECDSA Ethereum

3. 实现中用到的小技巧和创新点:

本实验进行了分子分母化简提高速率

```
def calculate_p_q(x1, y1, x2, y2, a, b, p):
    flag = 1 # 符号位
    # P = Q, 则k=[(3x1^2+a)/2*y1]mod p
    if x1 == x2 and y1 == y2:
        member = 3 * (x1**2) + a
        denominator = y1*2
    # P≠Q, 则k=(y2-y1)/(x2-x1) mod p
    else:
        member = y2 - y1
        denominator = x2 - x1
        if member * denominator < 0:
            flag = 0
            member = abs(member)
            denominator = abs(denominator)
    # 化简分子和分母
    gcd_value = get_gcd(member, denominator)
    member = member // gcd_value
    denominator = denominator // gcd_value
    # 求逆
    inverse = get_inverse(denominator, p)
    k = (member * inverse)
    if flag == 0:
        k = -k
    k = k % p
    # 计算x3, y3
    x3 = (k ** 2 - x1 - x2) % p
    y3 = (k * (x1 - x3) - y1) % p
    return [x3, y3]
```

3. 操作流程

主函数调用

```
print(write_ECDSA())
```

```
print(ECDSA())
```

求模逆

```
def get_inverse(a, m):#求模逆
    if get_gcd(a, m) != 1:
        return None
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return u1 % m
```

求 a 的 b 次方模 n

```
def pow_mod(a, b, n):
    a = a % n
    answer = 1
    # 因为分数没有取模运算，所以不需要考虑b小于0的情况
    while b != 0:
        if b & 1:
            ans = (ans * a) % n
        b = b - 1
        a = (a * a) % n
    return answer
```

判断是不是素数:

```
def is_prime(n):
    #检查是否为素数。
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n ** 0.5) + 1, 2):
        if n % i == 0:
            return False
    return True
```

签名和验证函数:

```
message="text202100210057"
def ECDSA():
    public=[0,0]
    def ECDSA_sign(msg):
        start_time=time.time()
        nonlocal public
        M = message.encode()
        private, public = get_key()
        k = random.randrange(1, n)
        R = calculate_np(Gx, Gy, k, a, b, p)
        r = R[0] % n
        e = int(shal(M).hexdigest(), 16)
        s = (get_inverse(k, n) * (e + private * r)) % n
        end_time=time.time()
        print("待签消息:", message)
        print("生成的签名为:")
        print(r, s)
        print(public)
        print(e)
        print("签名用的时间:", (end_time-start_time)*1000, "ms")
        return r, s

    def ECDSA_verif_sign(msg, sign):
        start_time = time.time()
        r, s = sign
        M = message.encode()
        e = int(shal(M).hexdigest(), 16)
        w = get_inverse(s, n)
        temp1 = calculate_np(Gx, Gy, e * w, a, b, p)
        temp2 = calculate_np(public[0], public[1], r * w, a, b, p)
        R, S = calculate_p_q(temp1[0], temp1[1], temp2[0], temp2[1], a, b, p)
        end_time= time.time()
        if (r == R):
            print("验证通过")
            print("验证用的时间:", (end_time - start_time)*1000, "ms")
            return True
```

4. 实现结果

```
True
待签消息: text202100210057
生成的签名为:
39375506681424162570071458265573171524585332988217490667860513871294082109004 34
2638131412653555662350725099854069255608276678735387075249975193309514286
[52952384766473377466571574773456020678951428990247574749002767803810159975073,
21921742080059634767981009846263899621436740124884923952925171495812499579019]
868576098922326651670958656163729832102617663271
签名用的时间: 41.96476936340332 ms
验证通过
验证用的时间: 78.4459114074707 ms
True
```

Project11: impl sm2 with RFC6979

1:简介

SM2 为非对称加密，基于 ECC。该算法已公开。由于该算法基于 ECC，故其签名速度与秘钥生成速度都快于 RSA。ECC 256 位（SM2 采用的就是 ECC 256 位的一种）安全强度比 RSA 2048 位高，但运算速度快于 RSA。SM2 算法定义了两条椭圆曲线，一条基于 F 上的素域曲线，一条基于 $F(2^m)$ 上的拓域曲线，目前使用最多的曲线为素域曲线，本文介绍的算法基于素域曲线上的运算，素域曲线方程定义如下：

$$y^2 - x^3 + ax + b$$

RFC6979(确定性签名算法)生产 k 的流程

关键词： RFC6979 SM2

2. 实现中用到的小技巧和 innovation 点：

本实验使用 sm3 哈希等来产生需要的 k

```
def RFC6979(num):  
    n=int(sm3.sm3_hash(list(num)), 16)  
    return n
```

3. 操作流程

调入相关函数库

```
import time
import random
from gmssl import sm3
```

相关参数

```
p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF
a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFFC
b = 0x28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93
n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7203DF6B21C6052B53BBF40939D54123
Gx = 0x32C4AE2C1F1981195F9904466A39C9948FE30BBFF2660BE1711D7C3A431D7C90
Gy = 0xD59796F4FE9EDD6B7D8B9C583CE2D3695A9E13641146433FBCC939DCE249296B
IDA=0x234C49481278122212567776555
IDB=0x66474383374646446474759
message="test202100210057"
```

RFC6979 相关函数

```
def RFC6979(num):
    n=int(sm3.sm3_hash(list(num)), 16)
    return n
```

求最大公因数和模逆

```
# 求最大公约数
def get_gcd(x, y):
    if y == 0:
        return x
    if x==0:
        return y
    while y!=0:
        x, y=y, x%y
    return x

def get_inverse(a, m):#求模逆
    if get_gcd(a, m)!=1:
        return None
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3!=0:
        q = u3//v3
        v1, v2, v3, u1, u2, u3 = (u1-q*v1), (u2-q*v2), (u3-q*v3), v1, v2, v3
    return u1%m
```

SM2 的相关实现和调用:

```
def signRFC6979(msg, ID):
    print("输入信息: ", message)
    r=0
    s=0
    dA, PA= key()
    M=message.encode()
    ENTL=8*get_bitsize(ID)
    data = ENTL.to_bytes(2,byteorder='little', signed=False)+int_to_bytes(ID)+in
    ZA = int(sm3.sm3_hash(list(data)), 16)
    M =int_to_bytes(ZA)+M
    e=int(sm3.sm3_hash(list(M)), 16)
    num=int_to_bytes(random.randrange(10000, 100000))+int_to_bytes(ID)+"mysm3RFC6

    #k的处理过程
    print("key的输入为: ")
    print(num)
    REALk=RFC6979(num)%n
    print("经过RFC6979的key为:")
    print(REALk)

    while True:
        x1,y1=calculate_np(Gx, Gy, REALk, a, b, p)
        r=(x1+e)%n
        sign=(get_inverse(1+dA, n)*(REALk-r*dA))%n
        if sign!=0 and r!=0 and r+REALk != n :
            REALk=(REALk+1)%n
            break
    print("签名为: ")
    return r, s
```

主函数调用:

```
begin=time.time()
print(signRFC6979(message, IDA))
end=time.time()
print("消耗时间: ", (end-begin)*1000, "ms")
```

Bit 和 int 的相关转换函数:

```
def bytes_to_int(bytes):
    n=int.from_bytes(bytes, byteorder='little')
    return n

def int_to_bytes(num):
    n=num.to_bytes(get_bitsize(num), byteorder='little', signed=False)
    return n
```

4. 实现结果

```
C:\Users\qcy12\Desktop\project_11\sm2.py
输入信息: test202100210057
key的输入为:
b'\xaa\xd9Uewg%!\x81'\x81\x94\xc4\x02mysm3RFC6979'
经过RFC6979的key为:
84801882358823959206110916530868317714199320188657202603686393079967562873525
签名为:
(95606354238633506007975074785374803246756857499065309256728770125247560966639,
0)
消耗时间: 147.6290225982666 ms
>
```


Project13: Implement the above ECMH scheme

1:简介

UTXO (Unspent Transaction Output)，是比特币交易中的基本单位。在比特币网络中，UTXO 代表了一定数量的比特币，可以被看作是比特币的零钱，可以用来支付交易。并且一旦某个 UTXO 被用作交易的输入，就会被消耗掉，同时意味着这 UTXO 中的比特币都被转移到了交易的输出。因此，UTXO 可以被视为比特币账户中的“余额”。

ECMH 把哈希值映射成椭圆曲线上的点，利用 ECC 运算，把多个数据的 hash 合并到一个 hash 中，并且支持删除等操作。这样节点维护一个 UTXO 的根 hash 的成本得到降低，只需要修改增量。然后只需要把 UTXO 根 hash 记录到区块，其他节点同步 UTXO 集合，可以验证该集合是否被篡改了。该方案的缺点是只能做全量验证，没办法验证单独一个 UTXO 是否存在。

关键词：UTXO ECMH

3. 实现中用到的小技巧和创新的点：

本实验在椭圆曲线上进行了相关操作

```
def T_add(P, Q):
    if (P == 0):
        return Q
    if (Q == 0):
        return P
    if P == Q:
        xxx = (3 * pow(P[0], 2) + a)
        yyy = gcd(2 * P[1], p)
        k = (aaa * bbb) % p
    else:
        xxx = (P[1] - Q[1])
        yyy = (P[0] - Q[0])
        k = (xxx * gcd(yyy, p)) % p

    Rx = (pow(k, 2) - P[0] - Q[0]) % p
    Ry = (k * (P[0] - Rx) - P[1]) % p
    R = [Rx, Ry]
    return R
```

3. 操作流程

相关参数

```
p = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc2f
a = 0
b = 7
n = 0xfffffffffffffffffffffffffffffebaedce6af48a03bbfd25e8cd0364141
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
```

求公因数

```
def gcd(x, y):
    if y == 0:
        return x
    if x==0:
        return y
    while y!=0:
        x, y=y, x%y
    return x
```

ECMH:

```
def ECMH(m):
    global p, a, b
    while (True):
        m = hash(m)
        tmp = m ** 3 + a * m + b
        tmp = tmp % p
        y = tonelli(tmp, p)
        if (y == -1):
            continue
        return tmp, y
```


主函数调用：

```
start=time.time()
r, s = ECMH('test')
print("哈希值:", r, ', ', s, ' ')
r1, s1 = ECMH_Group(['11311', '22522', '83933'])
end=time.time()
print("哈希值:", r1, ', ', s1, ' ')
time=(end-start)*1000
print("耗费时间为", time, "ms")
```

4. 实现结果

```
哈希值: ( 4794847758723081249320240162377303965802747149308594575 , 5408976720680
9146050305009044447366183547699428676754070799835026672662678747 )
哈希值: ( 11356508320721039895208142596264027905854458327813840244617835544475755
1332420 , 9744011922523356692426890677514664425123861922513600845976206985968661
5468194 )
耗费时间为 124.0394115447998 ms
```

Project18: send a tx on Bitcoin testnet, and parse the tx data down to every bit, better write script yourself

1:简介

比特币（Bitcoin）的概念最初的时候由中本聪在 2009 年的时候提出根据中本聪的思路设计发布的开源软件以及建构其上的 P2P 网络。比特币是一种 P2P 形式的数字货币点对点的传输意味一个去中心化的支付系统，比特币不依靠特定货币机构发行和特定的组织机构提供保障，依据特定算法，通过大量的计算生成比特币。比特币经济使用整个 P2P 网络中众多节点构成的分布式数据库来确认并记录所有的交易行为，并使用密码学的设计来确保货币流通各个环节安全性。P2P 的去中心化特性与算法本身可以确保无法通过大量制造比特币来人为操控币值。基于密码学的设计可以使比特币只能被真实的拥有者转移或支付。确保了货币所有权的匿名性，保障了用户的隐私权。

关键词：Bitcoin

2. 实现中用到的小技巧和创新的点：

本实验使用了网络发送请求

```
import requests
```

```
url='https://en.bitcoin.it/wiki/Testnet'  
tx={"1233672365448"}
```

3. 操作流程

主函数调用如下

```
url='https://en.bitcoin.it/wiki/Testnet'
tx={"1233672365448"}
print("发送的tx为:", tx)
post_html=requests.post(url, data=tx)
print("响应信息:")
print(post_html.text)
```

3. 实现结果

发送的tx为: {'1233672365448'}

响应信息:

```
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8" />
<title>Testnet - Bitcoin Wiki</title>
<script>document.documentElement.className.
replace( /(\s)client-nojs(\s|$)/, "$1client-js$2" );</script>
<script>(window.RLQ=window.RLQ||[]).push(function() {mw.config.set({ "wgCanonicalN
amespace": "", "wgCanonicalSpecialPageName": false, "wgNamespaceNumber": 0, "wgPageNam
e": "Testnet", "wgTitle": "Testnet", "wgCurRevisionId": 69794, "wgRevisionId": 69794, "w
gArticleId": 241, "wgIsArticle": true, "wgIsRedirect": false, "wgAction": "view", "wgUse
rName": null, "wgUserGroups": [ "*" ], "wgCategories": [ "Technical", "Developer", "Bitcoi
n Core documentation", "wgBreakFrames": false, "wgPageContentLanguage": "en", "wgPag
eContentModel": "wikitext", "wgSeparatorTransformTable": [ "", "" ], "wgDigitTransformT
able": [ "", "" ], "wgDefaultDateFormat": "dmy", "wgMonthNames": [ "", "January", "February
", "March", "April", "May", "June", "July", "August", "September", "October", "November",
"December"], "wgMonthNamesShort": [ "", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "A
ug", "Sep", "Oct", "Nov", "Dec"], "wgRelevantPageName": "Testnet", "wgRelevantArticleId
": 241, "wgRequestId": "6bb017e88371d42ecbc580b0", "wgIsProbablyEditable": false, "wgR
elevantPageIsProbablyEditable": false, "wgRestrictionEdit": [], "wgRestrictionMove":
[], "wgWikiEditorEnabledModules": { "toolbar": true, "preview": true, "publish": false } }
);mw.loader.state({"site.styles": "ready", "noscript": "ready", "user.styles": "ready
", "user": "ready", "user.options": "loading", "user.tokens": "loading", "mediawiki.leg
acy.shared": "ready", "mediawiki.legacy.commonPrint": "ready", "mediawiki.sectionAnc
hor": "ready", "mediawiki.skinning.interface": "ready", "skins.vector.styles": "ready
"});mw.loader.implement("user.options@0bhc5ha", function($, jQuery, require, module)
{mw.user.options.set([]);});mw.loader.implement("user.tokens@0ft0puh", function (
$. iQuerv. require. module ) {
```

Project19: forge a signature to pretend that you are Satoshi

1:简介

数字签名（digital signature）是只有信息的发送者才能产生的别人无法伪造的一段数字串，这段数字串同时也是对信息的发送者发送信息真实性的一个证明。不同于普通的物理签名，数字签名使用公钥加密领域的技术来保障安全性。一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。数字签名是非对称密钥加密技术与数字摘要技术的应用。

数字签名伪造，是形成数字签名可以通过验证。具有多种伪造方式对于不同的数字签名。

关键词：数字签名（digital signature） 数字签名伪造

2. 实现中用到的小技巧和创新的点：

3. 操作流程

引入相关模块

```
import hashlib
import gmssl
import random
import time
```


主函数调用如下：

```
begin=time.time()
print(forge_ECDSA())
end=time.time()
print("消耗的时间为", end-begin, "s")
```

Bytes 和 int 的相互转换

```
def int_to_bytes(num):
    return num.to_bytes(get_bitsize(num), byteorder='little', signed=False)
def bytes_to_int(bytes):
    return int.from_bytes(bytes, byteorder='little')
```

签名的伪造和验证

```
message= test_hello_world
print("消息:", message)
P=[12347259512020005462763638353364532312367891845761963173968514567546337027094, 123469442057811
r,s=65713732757593917641705955129814776632782548295209210156195240041086117167123, 54432546964026
def forge_ECDSA():
    def forge_sign():
        u = random.randrange(1, n)
        v = random.randrange(1, n)
        temp = calculate_np(Gx, Gy, u, a, b, p)
        temp2 = calculate_np(P[0], P[1], v, a, b, p)
        R = x, y = calculate_p_q(temp[0], temp[1], temp2[0], temp2[1], a, b, p)
        fr = x % n
        fs = (fr * get_inverse(v, n)) % n
        fe = (fr * u * get_inverse(v, n)) % n
        return fr, fs, fe
    r,s,e=forge_sign()
    print("伪造签名: ")
    print(r,s)
    def verify(r,s,e):
        k = get_inverse(s, n)
        temp = calculate_np(Gx, Gy, e * k, a, b, p)
        temp2 = calculate_np(P[0], P[1], r * k, a, b, p)
        R, S = calculate_p_q(temp[0], temp[1], temp2[0], temp2[1], a, b, p)
        if (R== r):
            return True
        else:
            return False
    return verify(r,s,e)
```

4. 实现结果

```
nature to pretend that you are Satoshi.py
消息: test_hello_world
伪造签名:
45769324867807848182693757708613419615958154744698171162035331352290111194079 99
81565651555957511831152984182693058110812090417463345129732174020748888006
False
消耗的时间为 0.15995359420776367 s
```

Project22: research report on MPT

1:简介

Merkle Patricia Tree (Merkle Patricia Trie) 是一种经过改良、融合默克尔树和前缀树两种树优点的数据结构，在太坊中用于组织管理哈希的重要数据结构。

MPT 树有以下几个作用：

1. 存储任意长的 key-value 键值对数据；
2. 提供了一种快速计算哈希标识的机制；
3. 提供快速状态回滚的机制；
4. 轻节点的扩展，进行验证

MPT 树中，树节点可以分为四类：空节点，分支节点，叶子节点，扩展节点。其中空节点表示空串。分支节点用来表示 MPT 树中所有拥有超过 1 个孩子节点以上的非叶子节点。在以太坊中，MPT 树的 key 值共有三种不同的编码方式，分别是，Raw 编码（原生的字符），Hex 编码（扩展的 16 进制编码），Hex-Prefix 编码（16 进制前缀编码）。

MPT 的操作有以下几种：Get（将需要查找 Key 的 Raw 编码转换成 Hex 编码，得到的内容称之为搜索路径）；Insert，Delete 等。MPT 树的提交过程是以根节点为入口，对根节点进行提交调用。

判断一个节点在内存中存在时间是否过长的依据是：该节点未被修改当前 MPT 的计数器减去节点的诞生标志超过了固定的上限；每当 MPT 调用一次 Commit 函数，MPT 的计数器发生自增。

轻节点的扩展的功能实现是通过默克尔证明实现的，默克尔证明指一个轻节点向一个全节点发起一次证明请求，询问全节点完整的默克尔树中，是否存在一个指定的节点；全节点向轻节点返回一个默克尔证明路径，由轻节点进行计算，验证存在性。

关键词：MPT 默克尔证明

2. 实现中用到的小技巧和创新的点：

本实验手动对 MPT 进行了模拟

```
void MPT()
{
    inode* temp = NULL;

    testMPT->publictitle = "46";
    testMPT->next_node[1] = new inode();
    testMPT->next_node[1]->pre = 2;
    testMPT->next_node[1]->leaf = 1;
    testMPT->next_node[1]->publictitle = "1355";
    testMPT->next_node[1]->wealth = 28;
    testMPT->next_node[15] = new inode();
    testMPT->next_node[15]->pre = 2;
    testMPT->next_node[15]->leaf = 9;
    testMPT->next_node[15]->publictitle = "8245";
    testMPT->next_node[15]->wealth = 2;
    testMPT->next_node[7] = new inode();
    temp = testMPT->next_node[7];

    temp->publictitle = "25";
```

3. 操作流程：

结点定义如下

```
struct inode
{
    int wealth = -1;
    bool leaf = 0;
    int pre = 0;
    string publictitle;
    inode* next_node[16];
};
```

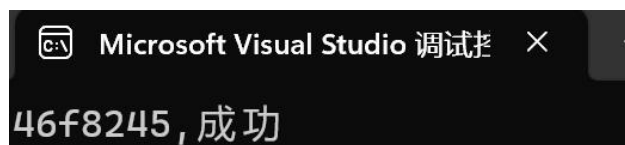
主函数调用如下：

```
int main()
{
    MPT();
    //测试结点为46f8245 2
    cout << "46f8245, ";

    if (check("46f8245") != 0) {
        cout << "成功";
    }
    else cout << "失败";

    return 0;
}
```

4. 实现结果



附录(参考文献)

密码学原理与实践

后记

经过本课程，我对网安的具体实践有了一定的了解，学习到了代码实现的一些技巧。