



网络空间安全创新创业实践

实验报告

山东大学网络空间安全学院

周宇帆 202100460108

目录

1	实验概况	5
2	Project1	6
2.1	Birthday attack 原理	6
2.2	数学推导	6
2.3	运行环境	8
2.4	实验内容	8
2.5	运行结果	8
3	Project2	8
3.1	实验原理	8
3.2	运行环境	9
3.3	实验内容	9
3.4	运行结果	9
4	Project3	10
4.1	实验原理	10
4.2	实验内容	11
4.3	运行环境	12
4.4	运行结果	12
4.5	参考资料	12
5	project4	12
5.1	运行结果	14
6	project5	14
6.1	实验内容	14
6.2	运行环境	14
6.3	运行结果	15
7	project9	15
7.1	实验环境	15
7.2	运行结果	15

8	project10	16
8.1	相关知识	16
8.1.1	椭圆曲线	16
8.1.2	ECDSA	16
8.1.2.1	公私钥生成	16
8.1.2.2	计算签名	16
8.2	ESDSA 签名恢复原理	17
8.3	ESDSA 签名恢复的过程	18
8.3.1	产生密钥 GenKey	18
8.3.2	签名算法 Sign	18
8.3.3	验证算法 Verify	18
8.3.4	恢复算法 Recover	19
8.4	以太网公钥恢复的意义	19
8.5	运行结果	19
9	Project11	20
9.1	运行结果	23
10	Project12	23
10.1	运行环境	23
10.2	运行结果	23
11	Project13	24
11.1	实验原理	24
11.2	实验环境	24
11.3	实验结果	25
12	Project14	25
12.1	实验环境	25
12.2	实验原理	25
12.3	实验内容	25
12.4	实验结果	28

13 Project15	28
13.1 实验原理	28
13.2 实验内容	28
13.3 实验指导	28
13.4 实验结果	29
14 Project16	29
14.1 实验原理	29
14.2 实验内容	29
14.3 实验指导	29
14.4 实验结果	30
15 Project17	30
16 Project18	32
17 Project19	33
17.1 实验原理	33
17.2 实验内容	33
17.3 实验指导	34
17.4 实验结果	34
18 Project21	34
18.1 实验结果	36
19 Project22	36

1 实验概况

所有 project 均为个人独立完成。

已完成的项目：

*Project1: implement the naïve birthday attack of reduced SM3

*Project2: implement the Rho method of reduced SM3

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

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

*Project5: Impl Merkle Tree following RFC6962

*Project9: AES / SM4 software implementation

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

*Project11: impl sm2 with RFC6979

*Project12: verify the above pitfalls with proof-of-concept code

*Project13: Implement the above ECMH scheme

*Project14: Implement a PGP scheme with SM2

*Project15: implement sm2 2P sign with real network communication

*Project16: implement sm2 2P decrypt with real network communication

*Project17: 比较 Firefox 和谷歌的记住密码插件的实现区别

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

*Project19: forge a signature to pretend that you are Satoshi

*Project20: ECMH PoC

*Project21: Schnorr Bacth

*Project22: research report on MPT

未完成的项目：

*Project6: impl this protocol with actual network communication

*Project7: Try to Implement this scheme

*Project8: AES impl with ARM instruction

2 Project1

2.1 Birthday attack 原理

生日攻击这个问题在数学上早有原型,叫做”[生日问题](https://en.wikipedia.org/wiki/Birthday_problem)

答案很出人意料。如果至少两个同学生日相同的概率不超过 5%, 那么这个班只能有 7 个人。事实上, 一个 23 人的班级有 50

这意味着, 如果哈希值的取值空间是 365, 只要计算 23 个哈希值, 就有 50% 的可能产生碰撞。也就是说, 哈希碰撞的可能性, 远比想象的高。实际上, 有一个近似的公式。

$$\sqrt{\frac{\pi}{2}N}$$

上面公式可以算出, 50% 的哈希碰撞概率所需要的计算次数, N 表示哈希的取值空间。生日问题的 N 就是 365, 算出来是 23.9。这个公式告诉我们, 哈希碰撞所需耗费的计算次数, 跟取值空间的平方根是一个数量级。

这种利用哈希空间不够大, 而制造碰撞的攻击方法, 就被称为生日攻击 (birthday attack)。

2.2 数学推导

这一节给出生日攻击的数学推导。

至少两个人生日相同的概率, 可以先算出所有人生日互不相同的概率, 再用 1 减去这个概率。

我们把这个问题设想成, 每个人排队依次进入一个房间。第一个进入房间的人, 与房间里已有的人 (0 人), 生日都不相同的概率是 ‘365/365’; 第二个进入房间的人, 生日独一无二的概率是 ‘364/365’; 第三个人是 ‘363/365’, 以此类推。

因此, 所有人的生日都不相同的概率, 就是下面的公式。

$$\bar{p}(n) = 1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{n-1}{365}\right)$$

上面公式的 n 表示进入房间的人数。可以看出, 进入房间的人越多, 生日互不相同的概率就越小。

这个公式可以推导成下面的形式。

$$\frac{365!}{365^n (365 - n)!}$$

那么，至少有两人生日相同的概率，就是 1 减去上面的公式。

$$p(n) = 1 - \bar{p}(n) = 1 - \frac{365!}{365^n (365 - n)!}$$

哈希碰撞的公式

上面的公式，可以进一步推导成一般性的、便于计算的形式。

根据泰勒公式，指数函数 e^x 可以用多项式展开。

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots$$

如果 x 是一个极小的值，那么上面的公式近似等于下面的形式。

$$e^x \approx 1 + x$$

现在把生日问题的 ‘ $1/365$ ’ 代入。

$$e^{-\frac{1}{365}} \approx 1 - \frac{1}{365}$$

因此，生日问题的概率公式，变成下面这样。

$$\begin{aligned} \bar{p}(n) &\approx 1 \cdot e^{-\frac{1}{365}} \cdot e^{-\frac{2}{365}} \cdot e^{-\frac{n-1}{365}} \\ &= e^{-\frac{1+2+\cdots+(n-1)}{365}} \\ &= e^{-\frac{n(n-1)/2}{365}} = e^{-\frac{n(n-1)}{730}} \\ p(n) &= 1 - \bar{p}(n) \approx 1 - e^{-\frac{n(n-1)}{730}} \end{aligned}$$

假设 d 为取值空间（生日问题里是 365），就得到了一般化公式。

$$p(n, d) \approx 1 - e^{-\frac{n(n-1)}{2d}}$$

上面就是哈希碰撞概率的公式。

2.3 运行环境

运行本代码前,必须安装 openssl,安装方法可以参考以下网址:<https://blog.csdn.net/zhizhengguar>
使用 visual studio 2019 运行得到结果.

2.4 实验内容

根据实验原理,运用 openssl 中的 sm3 函数、hash 函数等,用随机字符串碰撞已知字符串。创新点:利用 openssl 加快碰撞速度,利用预计算的方法,提前做表,减少时间。

2.5 运行结果

```
Find the collision!(16 bits)
Initial string:nFBWDV70gwlx4Yz
data(in hex) = 9161a9e809ec91693aad4fdbc325b08da6a353e4fd6b923d28531b20b3ed
find_data(in hex) = 6e4642574456373067776c7834597a3deff60d59ede0acdc6b692dd
H1 = 9161a9e809ec91693aad4fdbc325b08da6a353e4fd6b923d28531b20b3ed0971
H1_ = 9161fac55fd1f2642dee8f6e31d1b08236delc2f1d077f7b046elefefaa7cbcf
Running time = 0.148000 seconds
```

```
Find the collision!(24 bits)
Initial string:DZSvAIHR0FL5YOW
data(in hex) = 697e666c69d4b7d73d078fabb38d245d32f84a63f2aa434c6fe300c049ba072b
find_data(in hex) = 585544626345667470447451667551fb040edeef10bd19ead550ef84c18efd6
H1 = 697e666c69d4b7d73d078fabb38d245d32f84a63f2aa434c6fe300c049ba072b
H1_ = 697e66180d6b7a6b2874d3cb0419c193e0b3ac791203651e48971690793db438
Running time = 62.432000 seconds
```

3 Project2

3.1 实验原理

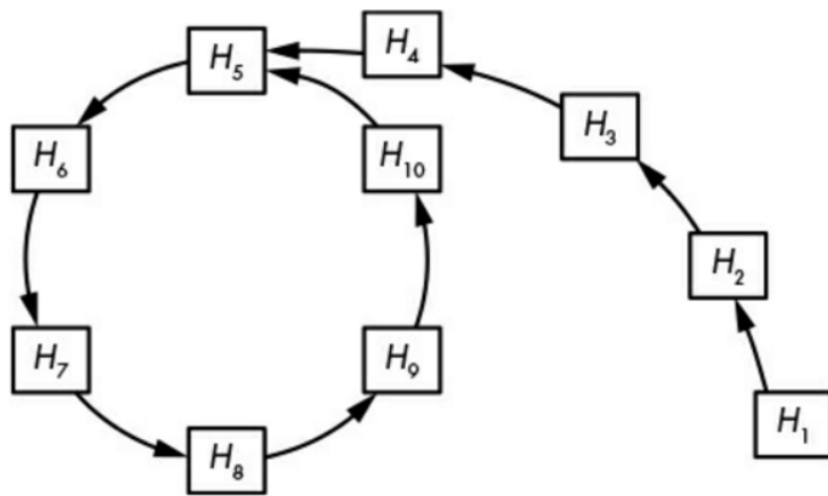
第二原像攻击:即给定消息 $M1$ 时,攻击者能够找到另一条消息 $M2$,其哈希值与 $M1$ 的哈希值相同。理论上复杂度为 $O(2^n)$

本实验利用 Pollard Rho 算法实现了 ** 第二原象攻击 **，即对于指定的字符串，找到与之哈希相同的字符串。最终，在可接受的时间里，实现了 32 比特的第二原象攻击。

Rho 攻击（来自 Pollard Rho 算法），流程如下

1. 给定具有 n 比特哈希值的哈希函数，选择一些随机哈希值 H_1 ，设 $H_1' = H_1$
2. 计算 $H_2 = Hash(H_1), H_2' = Hash(Hash(H_1'))$
3. 迭代该过程并计算 $H_{i+1} = Hash(H_i), H_{i+1}' = Hash(Hash(H_i'))$ ，直到有一个 i 可以满足 $H_{i+1} = H_{i+1}'$

对应的示意图如下



3.2 运行环境

visual studio 2019

需要提前安装 openssl 库，安装方法可以参考以下网址：

<https://blog.csdn.net/zhizhengguan/article/details/112846817>

3.3 实验内容

按照实验原理描述实现即可。

创新点：利用 openssl 库，提高运行速度。利用预处理。

3.4 运行结果

16 比特运行结果截图

```
Find the collision!(16 bits)
Initial string:2s0g40K3pYi70nwe      t
a(in hex) = 3eb605a5ceec811065640a18d2053a8705cb68528f27d11dffc03ff6887ea864
b(in hex) = 4cb759e381bf2b2fbb736e88d1fc1baa03c707cf082ecc314d037e14f87cc8b5
hash(b) (in hex) = 1653f841044d70dfe3d84e51fa93f86cf3eadfac4fec1b3f4161274462ba895f
H1 = hash(a) = 2fb8427368254e0ff0bdf8211e6f73e44cb52589e4ae55dd5faf28ac582ebf2
H1_ = hash(hash(b)) = 2fb84998d10ebd5d7cda41d488fcb106aadb2a7fd810ef17c987cc9dd71e61
Running time = 0.478000 seconds
```

24 比特运行结果截图

```
Find the collision!(24 bits)
Initial string:ybxexKN7Kv4Jy5IuMr7hPlMXj0qWE5301yu4Qk
a(in hex) = b9e3b455647ea03f60cb44682300ce89f40467ceefa8273bd759b98366332e71
b(in hex) = c01a7094611c8f20bab690f2fe0cbef3b763f35eef0d218cc85a2c08c53ba7d2
hash(b) (in hex) = c6369ec678efb48651d5e936153557804a2caa464176018079deb2734cf58cd6
H1 = hash(a) = 64bcf2caebc6ba648cc6c5b3d7ca5ab31f852c29f5baf08ed174077f1271d8a68
H1_ = hash(hash(b)) = 64bcf204b31fa424be835f75b49a1ab8f53a1a7531c9ab49652629ab8c400c04
Running time = 8.211000 seconds
```

4 Project3

4.1 实验原理

哈希长度扩展攻击 (Hash Length Extension Attacks)

当知道 $\text{hash}(\text{message})$ 的值及 message 长度的情况下, 可以推算出 $\text{hash}(\text{message}||\text{padding}||\text{m})$ 。在这里 m 是任意数据, $||$ 是连接符, 可以为空, padding 是 message 后的填充字节。hash 的 padding 字节包含整个消息的长度, 因此, 为了能准确计算出 padding 的值, message 的长度我们也是需要知道的。

当我们填充后, 服务器算出的原始 hash 值, 正好与我们添加扩展字符串并覆盖初始链变量所计算出来的一样。

SM3 算法过程

官方文档 [SM3 密码杂凑算法]

(<https://www.oscca.gov.cn/sca/xxgk/2010-12/17/1002389/files/302a3ada057c4a73830536d03e683>)

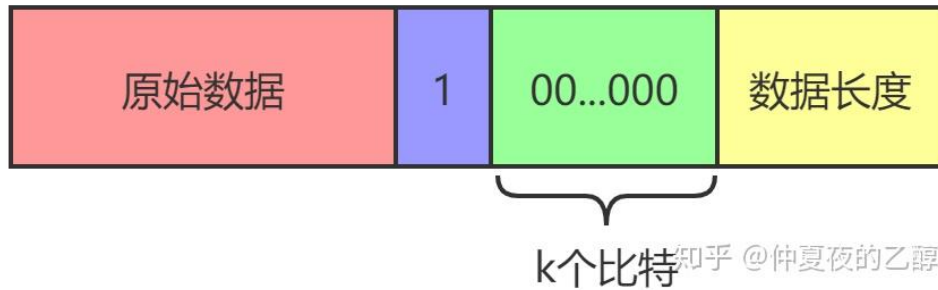
消息填充

SM3 的消息扩展步骤是以 512 位的数据分组作为输入的。因此, 我们需要在一开始就把数据长度填充至 512 位的倍数。数据填充规则和 MD5 一样, 具体步骤如下:

1. 先填充一个 “1”, 后面加上 k 个 “0”。其中 k 是满足 $(n+1+k) \bmod 512 = 448$ 的最小正整数。

2. 追加 64 位的数据长度 (bit 为单位, 大端序存放。观察算法标准原文附录 A 运算示例可以推知。)

填充完的数据大概长这样:



迭代过程

将填充后的消息 m 按 512 比特进行分组: $m = B^{(0)}B^{(1)} \cdots B^{(n-1)}$ $n = (l + k + 65)/512$

$$FOR i = 0 TO n - 1 \quad V^{(i+1)} = CF(V^{(i)}, B^{(i)}) \quad ENDFOR$$

其中 CF 是压缩函数, $V^{(0)}$ 为 256 比特初始值 IV , $B^{(i)}$ 为填充后的消息分组, 迭代压缩的结果为 $V^{(n)}$

消息扩展

SM3 的迭代压缩步骤没有直接使用数据分组进行运算, 而是使用这个步骤产生的 132 个消息字。(一个消息字的长度为 32 位) 概括来说, 先将一个 512 位数据分组划分为 16 个消息字, 并且作为生成的 132 个消息字的前 16 个。再用这 16 个消息字递推生成剩余的 116 个消息字。

在最终得到的 132 个消息字中, 前 68 个消息字构成数列 $\{W_j\}_{j=0}^{67}$ $\{W'_j\}_{j=0}^{67}$

压缩函数

令 A, B, C, D, E, F, G, H 为字寄存器, $SS1, SS2, TT1, TT2$ 为中间变量, 压缩函数 $V^{i+1} = CF(V^{(i)}, B^{(i)})$

4.2 实验内容

1. 随机生成一个消息 (secret), 用 SM3 函数算出 hash 值 (hash1)
2. 生成一个附加消息 (m)。首先用 hash1 推算出这一次加密结束后 8 个向量的值, 再以它们作为初始向量, 去加密 m, 得到另一个 hash 值 (hash2)
3. 计算 secret + padding + m 的 hash 值 (hash3), 如果攻击成功, hash2 应该和 hash3 相等

4.3 运行环境

visual studio 2019

4.4 运行结果

```

攻击的字符串: Zhou_Yu_Fan
输入字符串的ASCII码表示为: 5A686F75 5F59755F 46616E

填充后的消息为:
5A686F75 5F59755F 46616E80 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000058

杂凑值:
0F55B926 54FDCD0D 8BC4A86F 31E7C5AD 06D488C7 A6C76AE5 407ED0CC C57BE7BA

扩展的字符串: 202100460108
输入字符串的ASCII码表示为: 32303231 30303436 30313038

填充后的消息为:
32303231 30303436 30313038 80000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000060

杂凑值:
CD1D8D50 16B578B6 DFCDE031 F2FD55E3 508326C7 97710E4E 232ED3EE D0A4C61A

构造的字符串为(16进制形式):
5A686F75 5F59755F 46616E80 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000058
32303231 30303436 30313038 80000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000060

杂凑值:
CD1D8D50 16B578B6 DFCDE031 F2FD55E3 508326C7 97710E4E 232ED3EE D0A4C61A

```

可以看到第二次的哈希值和第三次构造的字符串的哈希值相同，攻击成功。

4.5 参考资料

SM3 的分布实现:

https://blog.csdn.net/nicai_hualuo/article/details/121555000

5 project4

本项目使用宏定义、SIMD 指令集、算法优化等方法实现了对 SM3 的优化方案。

代码说明

首先根据 sm3 说明文档，编写各个基本组件。一个重要的优化策略是利用 c 语言的宏定义代替函数，这样可以避免函数调用引起的开销，能够有效的提高运行速度。压缩函数

同样通过宏定义来实现。

同时，这里也有一个有效的优化措施：循环展开。

注意到，上图中所示的轮函数总共循环执行 64 次。我们以第一次为例，分析其展开的理论依据。

当第一次执行完毕后，有这样的关系：

$$A = TT1; B = A; C = B \ll 9; D = C; E = P_0(TT2); F = E; G = f \ll 19$$

当我们下一次迭代时，需要的参数仍然是 A H 按顺序排列。

在下一次迭代中有这样的关系：

$$Func(A', B', C', D', E', F', G', H')$$

等价于

$$Func(TT1, A, B \ll 9, C, P_0(TT2), E, F \ll 19, G)$$

可以看到，各个参数所关联的字寄存器仍然按照一定的规律排列，这里可以是 H, A, B, C, D, E, F, G 。

因此只需要让对应的寄存器满足对应位置的值，即

$$H = TT1; B = B \ll 9; F = F \ll 19; D = P_0(TT2)$$

这样，下一次调用

$$Func(TT1, A, B \ll 9, C, P_0(TT2), E, F \ll 19, G)$$

就等价于

$$Func(H, A, B, C, D, E, F, G)$$

看起来是将参数循环右移。

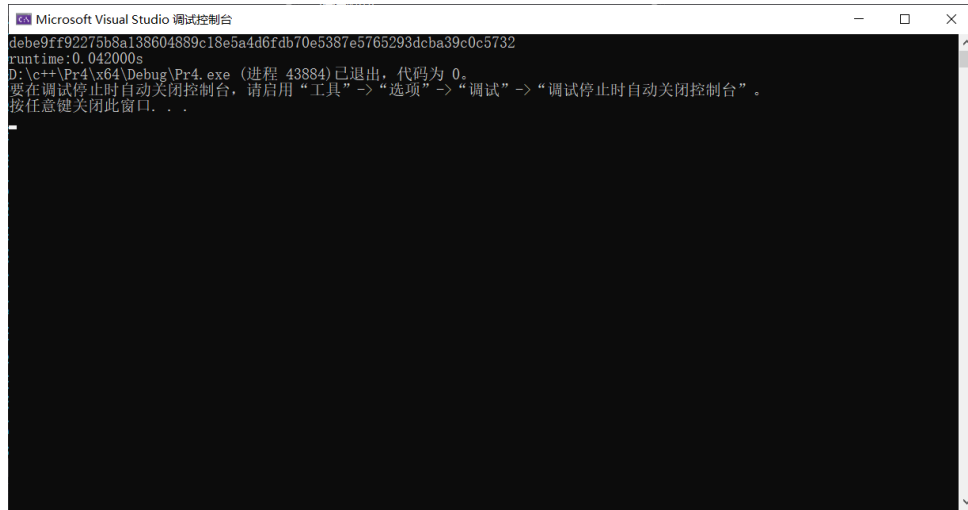
这样迭代 8 次后，就会回到最初的情况。

因此，可以将 8 次迭代作为一组，总共 8 组实现展开。

`define FF0(x,y,z) (x yz)defineFF1(x,y,z)((xy)|(xz)|(yz))`

`define GG0(x,y,z) (x yz)defineGG1(x,y,z)((xy)|((x)z))““`

5.1 运行结果



6 project5

依据 RFC6962, 创建一个 Merkle Tree, 并实现以下三点:

1. Construct a Merkle tree with 10w leaf nodes
2. Build inclusion proof for specified element
3. Build exclusion proof for specified element

6.1 实验内容

Merkle Tree 本质上是二叉树, 利用类这个数据结构, 建立起一个 Merkle Tree。其中, 哈希函数采用 sha256。

测试阶段, 随机产生了 100,000 个数据, 转化为 16 进制字符串作为叶子结点的值, 构建出 Merkle Tree。

测试存在性时, 分别测试了根节点、两个随机数、一个叶子结点, 最后结果为根节点和叶子结点均在 Merkle Tree 中, 两个随机数不在 Merkle Tree 中, 符合预期。

6.2 运行环境

Python 3.10

直接运行.py 文件, 建立 10w 个叶子结点的 Merkle Tree, 并测试了四个数据的存在性。

6.3 运行结果

```
Root of the Merkle Tree:
4117c98a9fb5ef9c1a5468b55d03a532e95cf5c577ef2df053f6be9bbadf849e
Inclusion and Exclusion Proof
Node: 4117c98a9fb5ef9c1a5468b55d03a532e95cf5c577ef2df053f6be9bbadf849e This node is in the Merkle Tree
Node: 25ac05d2eea45115842573be4f90cf67404bf21a49c01fc2bf870162ad0bb393 This node is not in the Merkle Tree
Node: 356fb16c30d2986ecadb7f68a9df00f36f43f688e9099b174fe2bb27f8e42ca9 This node is not in the Merkle Tree
Node: f755bc9cf5965e361f2c190d48a92738b7894f0a41e0b24d1d9557561aef3831 This node is in the Merkle Tree
```

(第一条为根节点，最后一条为叶子结点，中间两个数据是随机数，结果符合预期)

7 project9

SM4 算法是一种非对称 Feistel 结构的分组密码算法，其分组长度和密钥长度均为 128bits。

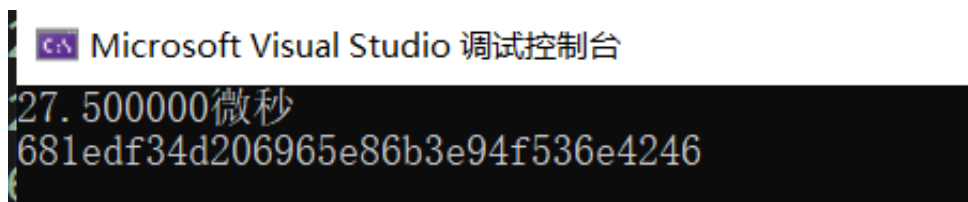
加密算法和密钥扩展算法迭代轮数均为 32 轮。SM4 加解密过程的算法相同但 SM4 的分组长度为 4 字，因此，其输入是 4 字的明文 (X_0, X_1, X_2, X_3)，输出是 4 字密文 (Y_0, Y_1, Y_2, Y_3)，经过 32 轮轮函数迭代，即每轮计算 $X_{i+4} = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus r_{ki})$

经过 32 轮得到 ($X_{32}, X_{33}, X_{34}, X_{35}$)，经过一次反序变换后，得到最终的密文 (Y_0, Y_1, Y_2, Y_3) = ($X_{35}, X_{34}, X_{33}, X_{32}$) SM4 的解密过程与加密过程完全相同，也包括 32 轮迭代和一次反序变换。只是在每轮迭代的时候，需要将轮密钥逆序使用。

7.1 实验环境

visual studio 2019

7.2 运行结果



```
Microsoft Visual Studio 调试控制台
27.500000 毫秒
681edf34d206965e86b3e94f536e4246
```

8 project10

8.1 相关知识

8.1.1 椭圆曲线

域 k (特征 0) 上的椭圆曲线可看成由下面方程的解全体再加上一个无穷远点: $y^2 = x^3 + ax + b, (x, y) \in k^2$, a, b 为常数, 并且判别式 $\Delta = -16(4a^3 + 27b^2) \neq 0$ 不等于 0。(即为了光滑性要求无重根)。

椭圆曲线上的点全体构成一个加法群, 点与点之间的“加法”运算。正因为椭圆曲线存在加法结构, 所以它包含了很多重要的数论信息。

对于密码学, 椭圆曲线是连续的, 并不适合用于加密。因此, 椭圆曲线密码学的第一要务就是把椭圆曲线定义在有限域上, (有限域 F_p , p 为素数), 并提出一条适于加密的曲线: $y^2 = x^3 + ax + b \pmod{p}$ 。

椭圆曲线通常与离散对数问题关联。椭圆曲线上的离散对数问题是指, 对于一个曲线上的点 $P = k * G$, G 为基点。已知 P 和 G , 计算 k 是困难的, 这就引出了公钥密码学的应用。

相比起在商业中被广泛采用的 RSA 加密算法, ECC 优势是可以使用更短的密钥, 来实现与 RSA 相当或更高级别的安全。通过下图我们清楚的发现, 160 位 ECC 加密安全性相当于 1024 位 RSA 加密, 而 210 位 ECC 加密安全性甚至相当于 2048 位 RSA 加密。

8.1.2 ECDSA

即椭圆曲线数字签名算法, 是 DSA 算法在椭圆曲线上的变形应用。

最原始的算法过程如下:

8.1.2.1 公私钥生成

随机取整数 $d_A \in [1, n - 1]$ 作为私钥。 n 为椭圆曲线群的阶。

计算 $Q = d_A * G$ 作为公钥, G 为基点。

8.1.2.2 计算签名

生成随机数 k 。

计算 $P = k * G$ ，得到曲线上一个点

取 P 的坐标 x ，令 $r = x \pmod n$

计算消息 m 的 hash 值 $H(m)$

计算 $s = k^{-1} * (H(m) + d_A * r) \pmod n$

输出 (r, s) 作为签名。

验证者需要得到签名对应的消息和公钥才可以验证签名合法性。

8.2 ESDSA 签名恢复原理

根据 ECDSA 签名算法中 s 的计算公式，利用适当的等式变形可以推导出公钥 P 的表达式：

$$s = k^{-1}(e + dr)$$

$$skG = eG + drG$$

$$sR = eG + rP$$

$$P = r^{-1}(sR - eG)$$

那么问题则从公钥表达式转移至还原 R ：

在 ECDSA 签名算法中， r 等于点 R 的横坐标；已知 r ，且点 R 在椭圆曲线：

$$y^2 = x^3 + Ax + B$$

上，我们可以通过横坐标 r 计算点 R 的纵坐标 y ，即求解二次剩余；

利用 Tonelli Shanks 算法，我们能够得到两个 y 值，即可计算点 R 与点 R' ，且 R 与 R' 关于 x 轴对称。

显然， R 与 R' 中，只有一个点与 ECDSA 签名时使用的 R 点相同。

那么问题从还原 R 转移至选择 R 点：

若得到签名 (r, s) 的同时得到签名前缀 v ，且 v 对于选择哪个 R 点有明确的指向性，那么就可以轻松地恢复 R ，并带入 式即可还原公钥 P ；

参考的资料中，对于签名前缀 v 没有具体的介绍，但如果签名前缀的作用仅是对恢复点 R 有明确指向性，那设计签名前缀 v 的算法是容易的：

首先，当 Tonelli Shanks 算法的输入一定时，它的输出即是固定的；

那么我们在 ECDSA 签名算法中，可以将 r 值代入 Tonelli Shanks 算法，判断一下关系式：

$$\text{tonellishanks}(r, P) == R_y$$

并设置一定的 v 值，使 v 与关系式是否成立有明确的联系；

恢复公钥时，一样是利用 Tonelli Shanks 算法计算纵坐标，并联系 v 值，选择：

1. Tonelli Shanks 算法的输出 y 所计算的点 $R:(x, y)$ ；

2. 选择另一个点 $R':(x, P - y)$ 。

8.3 ESDSA 签名恢复的过程

8.3.1 产生密钥 GenKey

选择一条椭圆曲线 $E_P(a, b)$ ，选择基点 G ， G 的阶数为 n

选择随机数 $d \in n$ 为私钥，计算公钥 $Q = d G$

8.3.2 签名算法 Sign

对消息 m 使用消息摘要算法，得到 $z = \text{hash}(m)$

生成随机数 $k \in n$ ，计算点 $(x, y) = k G$

取 $r = x \bmod n$ ，若 $r = 0$ 则重新选择随机数 k

计算 $s = k^{-1}(z + rd) \bmod n$ ，若 $s = 0$ 则重新选择随机数 k

上述 (r, s) 即为 ECDSA 签名

8.3.3 验证算法 Verify

使用公钥 Q 和消息 m ，对签名 (r, s) 进行验证。

验证 $r, s \in n$

计算 $z = \text{hash}(m)$

计算 $u_1 = zs^{-1} \bmod n$ $u_2 = rs^{-1} \bmod n$

计算 $(x, y) = u_1 G + u_2 Q \bmod n$

判断 $r == x$ ，若相等则签名验证成功

8.3.4 恢复算法 Recover

已知消息 m 和签名 (r,s) ，恢复计算出公钥 Q 。

验证 $r, s \in n$

计算 $R=(x, y)$ ，其中 $x=r, r+n, r+2n\cdots$ ，代入椭圆曲线方程计算获得 R

计算 $z = \text{hash}(m)$

计算 $u_1 = -zr^{-1} \bmod n$ $u_2 = sr^{-1} \bmod n$

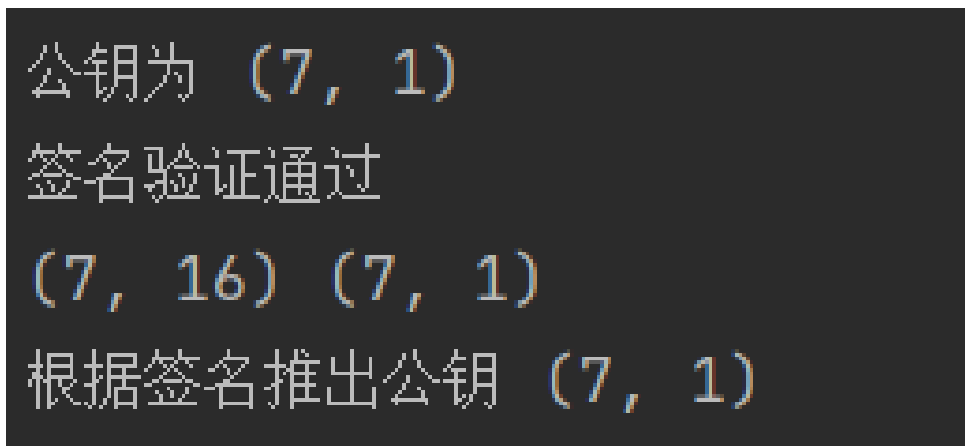
计算公钥 $Q = (x', y') = u_1 G + u_2 R$

8.4 以太网公钥恢复的意义

在区块链系统中，客户端对每笔交易进行签名，节点对交易签名进行验证。如果采用「验证算法 Verify」，那节点必须首先知道签发该交易所对应的公钥，因此需要在每笔交易中携带公钥，这需要消耗很大带宽和存储。如果采用「恢复算法 Recover」，并且在生成的签名中携带 recoveryID，就可以快速恢复出签发该交易对应的公钥，根据公钥计算出用户地址，然后在用户地址空间执行相应操作。

这里潜藏了一个区块链设计哲学，区块链上的资源（资产、合约）都是归属某个用户的，如果能够构造出符合该用户地址的签名，等同于掌握了该用户的私钥，因此节点无需事先确定用户公钥，仅从签名恢复出公钥，进而计算出用户地址，就可以执行这个用户地址空间的相应操作。

8.5 运行结果



```
公钥为 (7, 1)
签名验证通过
(7, 16) (7, 1)
根据签名推出公钥 (7, 1)
```

9 Project11

RFC6979

参照 RFC 文档中的实现方式，首先可以明确大致思路如下：

设置私钥为 pk ，私钥长度为 $qlen$ ，消息为 m

1. 计算 $h1 = H(m)$, H 即哈希函数，我选用的是 SHA256, $hlen$ 为哈希值的比特长度

2. 设置 $V = 0x01\ 0x01\ldots\ 0x01$ ，长度为 $hlen$

3. 设置 $K = 0x00\ 0x00\ldots\ 0x00$ ，长度为 $hlen$

4. 计算 $K = \text{HMAC}(V || 0x00 || \text{int2octets}(x) || \text{bits2octets}(h1))$

5. $V = \text{HMAC}(V)$

6. $K = \text{HMAC}(V || 0x01 || \text{int2octets}(x) || \text{bits2octets}(h1))$

7. $V = \text{HMAC}(V)$

8. 执行以下循环至找到合适的 K

* 设 T 为空序列， T 长度为 $tlen$ 个 bit

* $tlen < qlen$ 时执行

* $V = \text{HMAC}(V)$

* $T = T || V$

* 计算 $k = \text{bits2int}(T), k \in [1, q-1]$ ，则可输出，否则计算

* $K = \text{HMAC}(V || 0x00)$

* $V = \text{HMAC}(V)$

按照此流程初步实现了一种写法，但是在最后的循环部分有些小瑕疵，会出 bug，于是参考了 pybitcointools 中的处理方法，详见代码。

SM2 的优势之一在于采用随机数，因此同样的明文数据每一次加密结果都不一样，而使用 RFC6979 生成的 k 值由消息与私钥决定，因此可能会得到一样的结果，故在实现中同时采用 RFC6979 和随机数生成 k 值，并将二者相加从而使每次的加密结果不同，这样既保证泄露随机数种子也不能泄密，又能使同样的明文密钥能够得到不同的加密数值。

代码说明

RFC6979 文档中关于确定性产生 k 的方法如下：

“ Given the input message m , the following process is applied:

a. Process m through the hash function H , yielding:

$$h1 = H(m)$$

(h1 is a sequence of hlen bits).

b. Set:

$$V = 0x01\ 0x01\ 0x01\ \dots\ 0x01$$

such that the length of V, in bits, is equal to $8 \cdot \text{ceil}(\text{hlen}/8)$. For instance, on an octet-based system, if H is SHA-256, then V is set to a sequence of 32 octets of value 1. Note that in this step and all subsequent steps, we use the same H function as the one used in step 'a' to process the input message; this choice will be discussed in more detail in Section 3.6. c. Set:

$$K = 0x00\ 0x00\ 0x00\ \dots\ 0x00$$

such that the length of K, in bits, is equal to $8 \cdot \text{ceil}(\text{hlen}/8)$.

d. Set:

$$K = \text{HMAC}_K(V || 0x00 || \text{int2octets}(x) || \text{bits2octets}(h1))$$

where '||' denotes concatenation. In other words, we compute HMAC with key K, over the concatenation of the following, in order: the current value of V, a sequence of eight bits of value 0, the encoding of the (EC)DSA private key x, and the hashed message (possibly truncated and extended as specified by the bits2octets transform). The HMAC result is the new value of K. Note that the private key x is in the $[1, q-1]$ range, hence a proper input for int2octets, yielding rlen bits of output, i.e., an integral number of octets (rlen is a multiple of 8).

e. Set:

$$V = \text{HMAC}_K(V)$$

f. Set:

$$K = \text{HMAC}_K(V || 0x01 || \text{int2octets}(x) || \text{bits2octets}(h1))$$

Note that the "internal octet" is 0x01 this time.

g. Set:

$$V = \text{HMAC}_K(V)$$

h. Apply the following algorithm until a proper value is found for k:

1. Set T to the empty sequence. The length of T (in bits) is denoted tlen; thus, at that point, $\text{tlen} = 0$.

2. While $\text{tlen} < \text{qlen}$, do the following:

$$V = \text{HMAC}_K(V)$$

$T = T || V$ 3. Compute:

$k = \text{bits2int}(T)$

If that value of k is within the $[1, q-1]$ range, and is suitable for DSA or ECDSA (i.e., it results in an r value that is not 0; see Section 3.4), then the generation of k is finished.

The obtained value of k is used in DSA or ECDSA. Otherwise, compute:

$K = \text{HMAC}_K(V || 0x00)$

$V = \text{HMAC}_K(V)$

and loop (try to generate a new T , and so on).

Please note that when k is generated from T , the result of `bits2int` is compared to q , not reduced modulo q . If the value is not between 1 and $q-1$, the process loops. Performing a simple modular reduction would induce biases that would be detrimental to signature security. ““

根据此文档，使用 python，通过 `hmac`，`hashlib` 等库函数实现 k 的生成部分。

参考文档内容，编写生成 k 的函数：

```

1 def deterministic_generate_k(msghash, priv):
2     v = b'\x01' * 32
3     k = b'\x00' * 32
4     k = hmac.new(k, v+b'\x00'+priv+msghash, hashlib.sha256).
        digest()
5     v = hmac.new(k, v, hashlib.sha256).digest()
6     k = hmac.new(k, v+b'\x01'+priv+msghash, hashlib.sha256).
        digest()
7     v = hmac.new(k, v, hashlib.sha256).digest()
8     return bytes_to_int(hmac.new(k, v, hashlib.sha256).digest()
        )

```

得到 k 后，正常的构建 sm2 签名体系。

使用 `gmssl` 库中的 sm2 相关函数实现 sm2 的相关功能。并进行了加解密操作，验证了加解密的一致性，以及签名的正确性。

9.1 运行结果

```
k: 108913257062470343777232823561226136929544782759822580706321962097859599956788
加密结果 b'\xcb\t\xf4\x8b:\x15\xa5]]\xed\xbc\xf7\xe9\xeb\xfdqm\x7\x5\xe9\xdb?!\xf8\xd0\xf2\x8b\x98iu2R\xf6\xbe-\xbe\xcd\xca\xbd\x3'\xf5\t/\xad1xb3M0)
c|\x9b\xca\x17\xbc\x944\x0c\xec\x826(\xa3\x80u\x8f\xe7\x1e\xaa\x9c7w\x877\xf2\\LQ\x96\xda\x86\x9f\xec\x9f\x85\x16.\x86\x81\x9f\x1f\xbc0\xa8\x86\x95\x9a\x13;\xcfl\x99\xf1'
解密结果 b'282108468108'
解密一致:
验证签名成功
```

10 Project12

根据下图分别实现：

pitfalls	ECDSA	Schnorr	SM2-sig
Leaking k leads to leaking of d	✓	✓	✓
Reusing k leads to leaking of d	✓	✓	✓
Two users, using k leads to leaking of d , that is they can deduce each other's d	✓ RFC 6979	✓ RFC 6979	✓
Malleability, e.g. (r, s) and $(r, -s)$ are both valid signatures, lead to blockchain network split	✓	✓	$r = (e + x_1) \bmod n$ $e = \text{Hash}(Z_A M)$
Ambiguity of DER encode could lead to blockchain network split	✓	✓	----
One can forge signature if the verification does not check m	✓	✓	✓
Same d and k with ECDSA, leads to leaking of d	✓	✓	✓

10.1 运行环境

python 3.10

10.2 运行结果

由于运行结果过多，仅展示部分运行结果：

```
Success!
```

```
reuse k
ECDSA sig: reusing k leads to leaking of d is successful.
```

```
ECDSA sig: leaking k leads to leaking of d is successful.
```

11 Project13

11.1 实验原理

‘ECMH’ 用于 hash 单个元素, ‘ECMH_{set}’ hash $F_{2^n}^*$ 上的元素映射到椭圆曲线加法群上 G , 这样得到的 hash 就可以满足以下两条性质

1. $ECMH(A + B) = ECMH(A) + ECMH(B)$
2. $ECMH(A, B) = ECMH(B, A)$

ECMH 相关代码:

```

1 MultiSet Hash -> EC combine/add/remove
2 def add(ecmh, msg):
3     dot = msg_to_dot(msg)
4     tmp = EC_add(ecmh, dot)
5     return tmp
6 def single(msg):
7     return add(0, msg)
8 def remove(ecmh, msg):
9     dot = msg_to_dot(msg)
10    tmp = EC_sub(ecmh, dot)
11    return tmp
12 def combine(msg_set):
13    ans = single(msg_set[0])
14    num = len(msg_set) - 1
15    for i in range(num):
16        ans = add(ans, msg_set[i + 1])
17    return ans

```

11.2 实验环境

python 3.10 需要安装 hashlib 和 random 库

11.3 实验结果

```

===== duplicate elements=====
ECMH(m1):
(69420866617830924353956261676336861255436797791904662663853227576699199851903, 30196023516719035805058991859968778327986327316188247870164710850138412508315)
ECMH(m1 + m1):
(33594589687661287408579584432064823731359048625875210520931192727182495756811, 111086472205153975014491712669983250593467262384628734453844800962268384356391)
ECMH(m1) + ECMH(m1):
(33594589687661287408579584432064823731359048625875210520931192727182495756811, 111086472205153975014491712669983250593467262384628734453844800962268384356391)

===== add =====
ECMH(m1):
(69420866617830924353956261676336861255436797791904662663853227576699199851903, 30196023516719035805058991859968778327986327316188247870164710850138412508315)
ECMH(m2):
(253817273960805293598618269957231630465419176595282845729877904699774099676, 373918941673995795437143182616124599547920474398268823031511454828408785607)
ECMH(m1 + m2):
(53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020886922485844846866462680968691462886165646974260151587462)
ECMH(m1) + ECMH(m2):
(53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020886922485844846866462680968691462886165646974260151587462)

===== remove =====
ECMH(m1) + ECMH(m2) + ECMH(m3):
(5165596798266590968351747018273676682938584724389071889618322742629207592858, 18138801520186210651102645053859190544843239836415079925322159176636438992098)
ECMH(m1 + m2 + m3):
(5165596798266590968351747018273676682938584724389071889618322742629207592858, 18138801520186210651102645053859190544843239836415079925322159176636438992098)
ECMH(m1 + m2):
(53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020886922485844846866462680968691462886165646974260151587462)
ECMH(m1 + m2 + m3) - ECMH(m3):
(53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020886922485844846866462680968691462886165646974260151587462)

```

12 Project14

12.1 实验环境

python 3.10 需要安装 Crypto、gmssl、random 等库。

12.2 实验原理

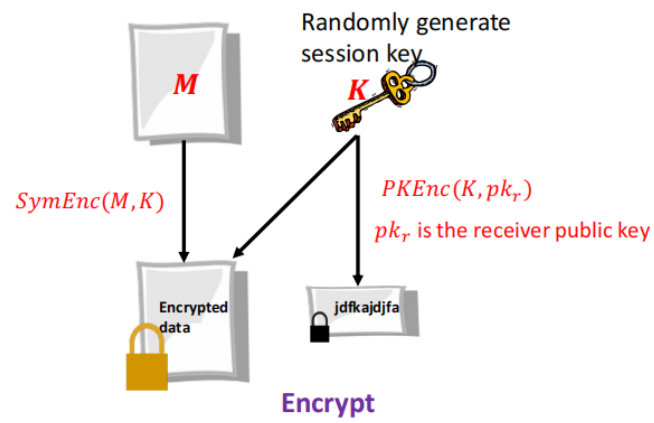
PGP(Pretty Good Privacy), 是一个基于 RSA 公钥和对称加密相结合的邮件加密软件。该系统能为电子邮件和文件存储应用过程提供认证业务和保密业务。

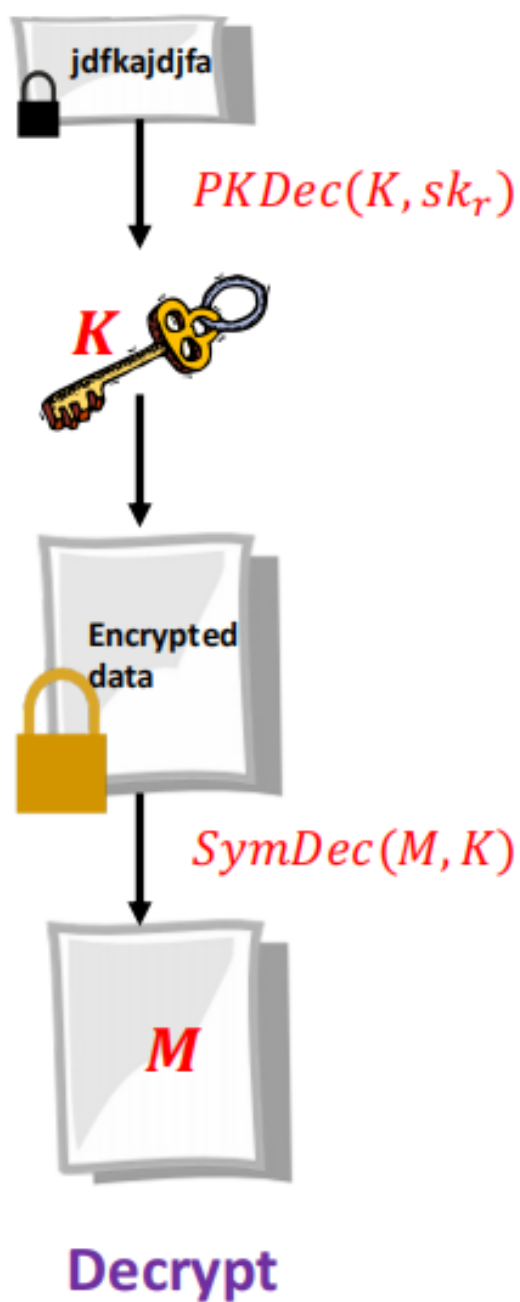
PGP 是个混合加密算法, 它由一个对称加密算法、一个非对称加密算法、与单向散列算法以及一个随机数产生器(从用户击键频率产生伪随机数序列的种子)组成。

12.3 实验内容

本次实验旨在实现一个简易 PGP, 调用 GMSSL 库中封装好的 SM2/SM4 加解密函数。

加密时使用对称加密算法 SM4 加密消息, 非对称加密算法 SM2 加密会话密钥; 解密时先使用 SM2 解密求得会话密钥, 再通过 SM4 和会话密钥求解原消息。加解密过程如下





部分代码说明

```

def epointmod(a, n)          a mod n
def epointmodmult(a, b, n)    a * b-1 mod n
def epointadd(P, Q, a, p)      P + Q
def epointmult(k, P, a, p)     k * P
def keygen(a, p, n, G) 生成 SM2 算法的公私钥对
def pgpenc(m, k)PGP

```

加密之前要先对消息进行填充, SM4 分组长度为 128 比特即 16 个字节, 填充完成后, 需要将消息 m 与密钥 k 转化为 bytes 类型。

调用 GMSSL 库中封装好的 SM4 加密函数对信息进行加密。

12.4 实验结果

```
Message,
b'32303231303034363031303040404040'

key,
b'3966383136393139336134663937313634356565376166386233383234643239'

Enc_message,
b'409B5E4E11C9ABF895C188C307BBAF9729027623091D334DF8E64701C900129A'

Enc_key,
b'
'4A1D7E8010B28F35B48712598E2098C4A3130A98A4D5C1A35CF2E67023F28BAE1393D433F753189848E083D23298B68F4DB4315E5868F400280AFA57A7A32A0230487BA3E630A497A712AFC688AA2495F94211C875AF
2ECDC15FB54028B278589B3E6088EB3A5366EA755B7C3674314FC9ED91E7A89CCECCED1E8CE2FA05C70D'

Dec_key,
b'3966383136393139336134663937313634356565376166386233383234643239'

Dec_message,
282108469188
```

13 Project15

13.1 实验原理

原理同上个实验大致相同。

13.2 实验内容

实现交互式协议

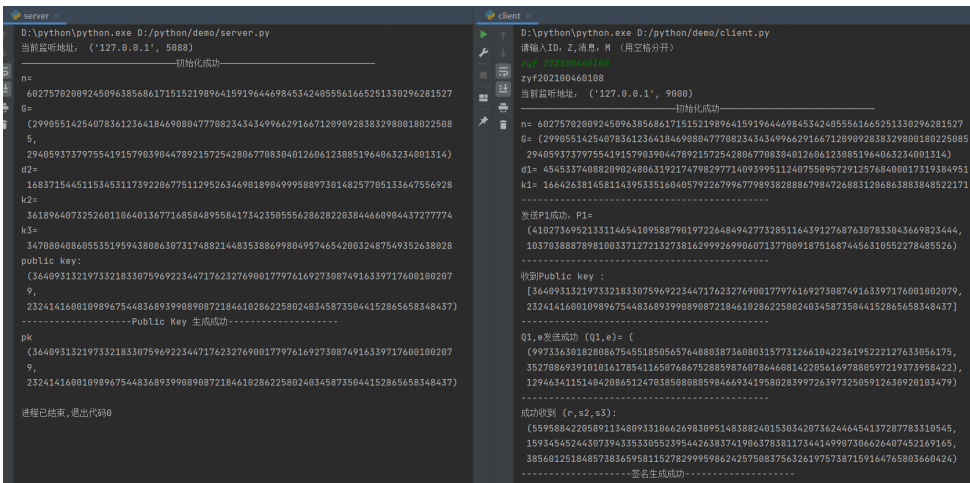
创新点: 运用 socket 实现网络交互。

13.3 实验指导

python 3.10

先运行 server.py 文件, 再运行 client.py 文件。

13.4 实验结果



14 Project16

14.1 实验原理

3.6 SM2 two-party decrypt

- Public key: $P = [(d_1 d_2)^{-1} - 1]G$
- Private key: $d = (d_1 d_2)^{-1} - 1$

(1) Generate sub private key $d_1 \in [1, n - 1]$

(2) get ciphertext $C = C_1 || C_2 || C_3$

- Check $C_1 \neq 0$
- Compute $T_1 = d_1^{-1} \cdot C_1$

(4) Recover plaintext M'

- Compute $T_2 - C_1 = (x_2, y_2) = [(d_1 d_2)^{-1} - 1] \cdot C_1 = kP$
- Compute $t = KDF(x_2 || y_2, klen)$
- Compute $M'' = C_2 \oplus t$
- Compute $u = Hash(x_2 || M' || y_2)$
- If $u = C_3$, output M'

(1) Generate sub private key $d_2 \in [1, n - 1]$

(3) compute $T_2 = d_2^{-1} \cdot T_1$

T_1

T_2

Encrypt:

- $C_1 = kG = (x_1, y_1)$ where $k \in [1, n - 1]$
- $kP = (x_2, y_2)$
- $t = KDF(x_2 || y_2, klen)$
- $C_2 = M \oplus t$
- $C_3 = H(x_2 || M || y_2)$

*Project: implement sm2 2P decrypt with real network communication

14.2 实验内容

实现交互式协议

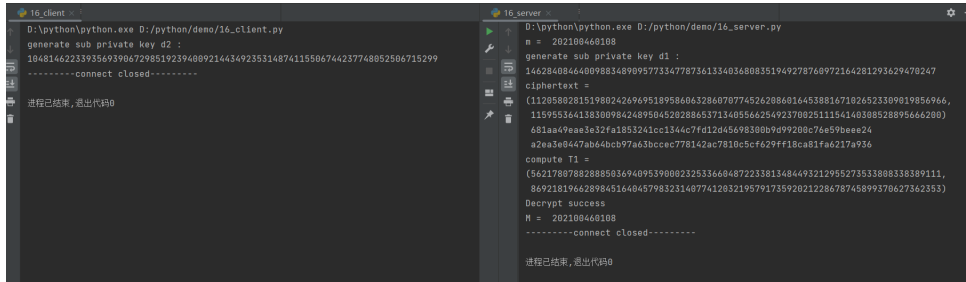
创新点：运用 socket 实现网络交互.

14.3 实验指导

python 3.10

先运行 server.py 文件，再运行 client.py 文件。

14.4 实验结果



```
16_client
D:\python\python.exe D:/python/demo/16_client.py
generate sub private key d2 :
10401462233935693906729851923940892144349235314874115506744237748852506715299
-----connect closed-----
进程已结束，退出代码0

16_server
D:\python\python.exe D:/python/demo/16_server.py
a = 20210846d108
generate sub private key d1 :
14428408444809883489095773347787361334036808351949278768972164281293629470247
ciphertext =
(112088028151980242696951895860632860707745262086016453881671026523309819856966,
11595536413830098424895045202886537134055662549237082511154140308528895666200)
681aa9eae1e32fa1853241cc1344c7fd12d4569830b9d99208c76e59bee24
a2ea1e0447ab64bcb97a63bccc778142ac7810c5cf629ff18ca81fa6217a936
compute T1 =
(5621788788288583694095390082325356604872233813484493212955273533808338389111,
86921819662898451640457983231487741283219579173592021228678745899370627362353)
Decrypt success
n = 20210846d108
-----connect closed-----
进程已结束，退出代码0
```

15 Project17

Chrome 记住密码实现

google 保存的密码数据存储在位于此处的 SQLite 数据库中：

AppDataData Local State

可以使用 SQLite 数据库浏览器打开此文件（文件名只是“登录数据”）并查看包含已保存密码的“登录”表。会注意到“password_oalue”

为了执行加密（在 Windows 上），Chrome 使用 Windows 提供的 API 函数，该函数使得加密数据只能由用于加密密码的 Windows 用户帐户解密。因此本质上，您的主密码就是您的 Windows 帐户密码。因此，一旦您使用帐户登录 Windows，Chrome 就可以解读此数据。

但是，由于您的 Windows 帐户密码是一个常量，因此对“主密码”的访问并非 Chrome 独有，因为外部实用程序也可以获取此数据并对其进行解密。使用 NirSoft 提供的免费实用程序 ChromePass，您可以查看所有保存的密码数据并轻松将其导出到纯文本文件。

因此，如果 ChromePass 实用程序可以访问这些数据，那么以相应用户身份运行的恶意软件也可以访问它，这是有道理的。当 ChromePass.exe 上传到 VirusTotal 时，超过一半的防病毒引擎将其标记为危险。虽然在这种情况下该实用程序是安全的，但令人有点放心的是，这种行为至少被许多 AV 软件包标记（尽管 Microsoft Security Essentials 不是报告其危险的 AV 引擎之一）

Google Chrome 为您提供了一个默认工具，无需安装即可保存您的登录凭据。它使用 AES 256 位 SSL/TLS 加密以及密码短语功能，为您的密码和个人信息提供额外的安

全性。除了生成和保存密码之外，您还可以通过 Chrome 的密码检查查看您的登录信息。该功能默认启用。Firefox 记住密码

Firefox 的同步是一个本地 AES-256-CBC 加密数据库，存储您的资料（可以包括密码），存储在 Mozilla 的服务器上。该密钥不会以未加密的形式离开您的浏览器，除了您之外，任何人都可以解密。但它最终会出现在您同步的每个 Firefox 浏览器上。

同步密钥存储在您本地的密码中。如果您没有 Firefox 主密码，则该密码不会在您的计算机上加密。如果您使用主密码，则从您将同步密钥输入 Firefox 的那一刻起，同步密钥就不会加密。

同步密钥可从您的浏览器获取。转到选项/Firefox 同步，单击“管理帐户”工具，选择“我的恢复密钥”，它将生成密钥的可打印版本。您可以在 Firefox 的任何其他实例中键入用户的电子邮件地址和该密钥，您将被纳入同步中，因此可以完全查看所有已同步的密码。

为满足开发者创建满足各种安全标准的应用程序，Mozilla 开发了一个叫做“Network Security Services”，或叫 NSS 的开源库。Firefox 使用其中一个叫做“Security Decoder Ring”，或叫 SDR 的 API 来帮助实现账号证书的加密和解密函数。firefox 使用它完成加密：

当一个 Firefox 配置文件被首次创建时，一个叫做 SDR 的随机 key 和一个 Salt（译者注：Salt，在密码学中，是指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符，这种过程称之为“加盐”）就会被创建并存储在一个名为“key3.db”的文件中。利用这个 key 和盐，使用 3DES 加密算法来加密用户名和密码。密文是 Base64 编码的，并存储在一个叫做 signons.sqlite 的 sqlite 数据库中。Signons.sqlite 和 key3.db 文件均位于

所以我们要做的就是得到 SDR 密钥。正如此处解释的，这个 key 被保存在一个叫 PKCS11 软件“令牌”的容器中。该令牌被封装进入内部编号为 PKCS11 的“槽位”中。因此需要访问该槽位来破译账户证书。

还有一个问题，这个 SDR 也是用 3DES(DES-EDE-CBC) 算法加密的。解密密钥是 Mozilla 叫做“主密码”的 hash 值，以及一个位于 key3.db 文件中对应的叫做“全局盐”的值。

Firefox 用户可以在浏览器的设置中设定主密码，但关键是好多用户不知道这个特性。正如我们看到的，用户整个账号证书的完整性链条依赖于安全设置中选择的密码，它是攻击者唯一不知道的值。如果用户使用一个强健的主密码，那么攻击者想要恢复存储的证书是不太可能的。

那么——如果用户没有设置主密码，空密码就会被使用。这意味着攻击者可以提取全局盐，获得它与空密码做 hash 运算结果，然后使用该结果破译 SDR 密钥。再用破译的 SDR 密钥危害用户证书。

比较

Firefox 和 Chrome 都有本机密码管理器，允许用户安全地存储其各种在线帐户的密码。Firefox 的密码管理器使用主密码来“解锁”您保存的其余密码，而 Chrome 只保存每个密码。要求主密码可以防止其他人在碰巧有权访问您的设备或浏览器时登录您的帐户，从而使 Firefox 的密码管理器更加安全。

相比于 chrome 浏览器，firefox 记住密码功能实现更复杂，安全性更高

参考:(<https://security.stackexchange.com/questions/41029/comparison-between-firefox-password-manager-and-chrome-password-manager>)

16 Project18

交易信息

打开一个比特币交易网站：<https://blockchair.com/bitcoin-cash/block/804701>

找到一次交易信息，如下图所示：

General info			
Mined on	Aug 4, 2023 8:26 AM UTC	Miner	AntPool
Transaction count	104	Fee per kB	0.00001807 BCH - ~0 USD
Input count	300	Output count	267
Input total	2,018.24 BCH - 456,607 USD	Output total	2,024.49 BCH - 458,021 USD
Fee total	0.0009984 BCH - 0 USD	Coindays destroyed	379.71
Generation	6.25 BCH - 1,414 USD	Reward	6.2509984 BCH - 1,414 USD
Size	55,558	Median time	Aug 4, 2023 7:45 AM UTC
Version	536870912 ₁₀ 20000000 ₁₆	Version [bits]	10000000000000000000000 ₂
Merkle root	29ad48...173023	Difficulty	402,516,518,316
Nonce	167,762,843	Bits	402,832,199
Chainwork	000000...a2d6aa		
Coinbase data	G Mined by AntPool18051 g0mm17,0p1j?id!0a 5S,#		

可以看到以下信息：

Hash 是这个区块的前一个区块的 hash 值。也就是矿工要进行计算的值。

时间戳用来标识这个区块挖出的时间

Height 指的是这个区块之前区块的数量

transaction count 这个区块内部交易的数量

Difficulty 衡量挖掘比特币区块的难度

Merkle root 指的是 merkle 树的根的 hash 值

Version 版本号

Bits 目标哈希的难度等级，表示解决 nonce 的难度

Nonce: 矿工必须解决的加密数字，以验证区块。nonce 用于验证块中包含的信息，生成一个随机数，将其附加到当前标头的散列中，重新散列该值，并将其与目标散列进行比较。

Transaction Volume 比特币的交易量

reward 比特币区块奖励是奖励给矿工的新比特币，

17 Project19

17.1 实验原理

ECDSA – Forge signature when the signed message is not checked

- Key Gen: $P = dG$, n is order
- Sign(m)
 - $k \leftarrow \mathbb{Z}_n^*$, $R = kG$
 - $r = R_x \bmod n$, $r \neq 0$
 - $e = \text{hash}(m)$
 - $s = k^{-1}(e + dr) \bmod n$
 - Signature is (r, s)
- Verify (r, s) of m with P
 - $e = \text{hash}(m)$
 - $w = s^{-1} \bmod n$
 - $(r', s') = e \cdot wG + r \cdot wP$
 - Check if $r' == r$
 - Holds for correct sig since
 - $es^{-1}G + rs^{-1}P = s^{-1}(eG + rP) =$
 - $k(e + dr)^{-1}(e + dr)G = kG = R$
- $\sigma = (r, s)$ is valid signature of m with secret key d
- If only the hash of the signed message is required
- Then anyone can forge signature $\sigma' = (r', s')$ for d
- (Anyone can pretend to be someone else)
- Ecdsa verification is to verify:
 - $s^{-1}(eG + rP) = (x', y') = R'$, $r' = x' \bmod n == r$?
 - To forge, choose $u, v \in \mathbb{F}_n^*$
 - Compute $R' = (x', y') = uG + vP$
 - Choose $r' = x' \bmod n$, to pass verification, we need
 - $s'^{-1}(e'G + r'P) = uG + vP$
 - $s'^{-1}e' = u \bmod n \rightarrow e' = r'u v^{-1} \bmod n$
 - $s'^{-1}r' = v \bmod n \rightarrow s' = r'v^{-1} \bmod n$
 - $\sigma' = (r', s')$ is a valid signature of e' with secret key d

*Project: forge a signature to pretend that you are Satoshi

17.2 实验内容

伪造签名过程 1、重新选择 a、b，计算: “ $(x_2, y_2) = a * G + b * Q_A$ ”

2、计算: “ $r_1 = x_2 \bmod n$ $b_1 = (b^{-1} - 1) \bmod n$ $e_1 = r_1 * a * b_1 \bmod n$ $s_1 = r_1 * b_1 \bmod n$ ” (r_1, s_1)

部分代码说明 def gcd(a,b) 辗转相除法求最大公因子

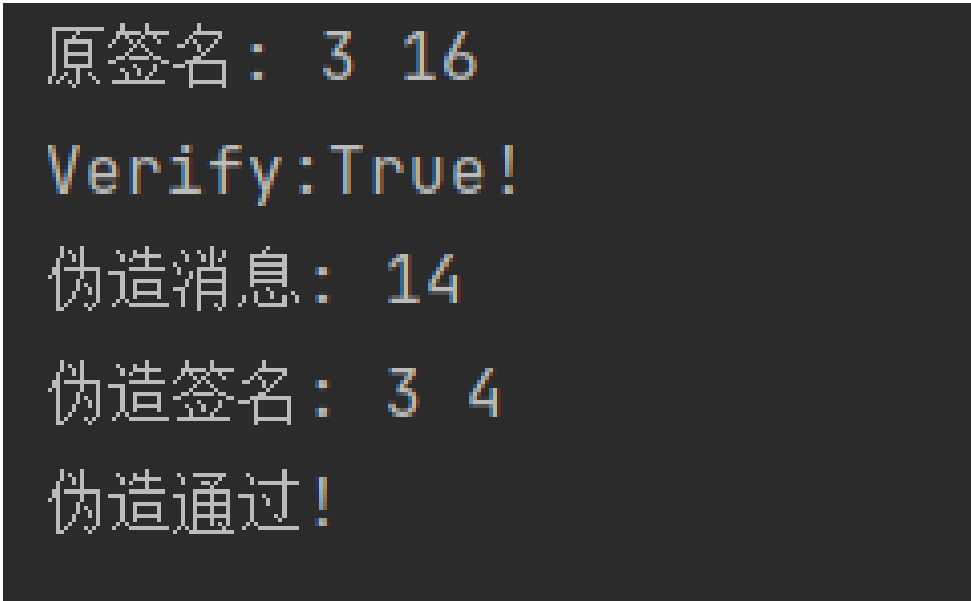
def xgcd(a, m) 扩展欧几里得算法求模逆

```
def epointadd(P, Q)
epointmul(k, g)
def signature(m) ECDSA 签名算法
def verify(r,s) ECDSA 验签算法
def forgesignature(r, s)
```

17.3 实验指导

python 3.10

17.4 实验结果



```
原签名: 3 16
Verify: True!
伪造消息: 14
伪造签名: 3 4
伪造通过!
```

18 Project21

Schnorr 数字签名算法简单流程

Setup:

```
x := random number    (aka private key)
G := common point
X := x * G    (aka public key)
```

Sign:

```
r := random number (aka nonce)
R := r * G    (aka commitment)
e := Hash(R, X, message) (aka challenge)
s := r + e * x    (aka response)
return (R, X, s, message)    ((s, e) aka signature)
```

Verify:

```
receive (R, X, s, message)
e := Hash(R, X, message)
s1 := R + e * X
s2 := s * G
return OK if S1 equals S2
```

schnorr batch verify

Schnorr Signature – Batch Verification

[Utilize the linear property of Schnorr signature's verification process](#)

- Recall Schnorr signature's verification: $sG = (k + ed)G = kG + edG = R + eP$
- Batch verification equation is :
 - $(\sum_{i=1}^n s_i) * G = (\sum_{i=1}^n R_i) + (\sum_{i=1}^n e_i * P_i)$
 - Attacker can forge signature to pass the batch verification
- Suppose attacker's public key $P_1 = x_1 * G$, to forge signature for public key $P_2 = x_2 * G$
 - x_2 is not known to attacker
 - Attacker randomly choose $r_2, s_2, R_2 = r_2 * G$, and computes $e_2 = h(P_2 || R_2 || m_2)$
 - Attacker set $R_1 = -(e_2 * P_2)$, and computes $e_1 = h(P_1 || R_1 || m_1)$
 - Then he derive $s_1 = r_2 + e_1 x_1 - s_2 \text{ mod } p$
 - It can be verified that signatures $(R_1, s_1), (R_2, s_2)$ pass the batch verification:
 - $(s_1 + s_2) * G = R_1 + R_2 + e_1 P_1 + e_2 P_2$
- Defense: randomly choose $a_i \in [0, p - 1], i \in [2, n]$ and verifies the following equation:
 - $(s_1 + \sum_{i=2}^n a_i s_i) * G = (R_1 + \sum_{i=2}^n a_i R_i) + (e_1 * P_1 + \sum_{i=2}^n (e_i a_i) * P_i)$

18.1 实验结果

```
原始方法验证结果:  
True  
True  
False  
原始方法的时间: 0.3010554313659668  
schnorr_batch验证结果:  
False  
schnorr_batch的时间: 0.19300532341003418
```

19 Project22

见文件夹的 md 文件