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** 6

2 Project1 7

2.1 Birthday attack 原理 . . . 7

2.2 数学推导 . . . 7

2.3 运行环境 . . . 9

2.4 实验内容 . . . 9

2.5 运行结果 . . . 9

3 Project2 10

3.1 实验原理 . . . 10

3.2 运行环境 . . . 10

3.3 实验内容 . . . 10

3.4 运行结果 . . . 11

4 Project3 11

4.1 实验原理 . . . 11

4.2 实验内容 . . . 12

4.3 运行环境 . . . 13

4.4 运行结果 . . . 13

4.5 参考资料 . . . 13

5 project4 13

5.1 实验内容 . . . 14

5.2 运行结果 . . . 16

5.3 运行指导 . . . 17

6 project5 17

6.1 实验内容 . . . 17

6.2 运行环境 . . . 17

6.3 运行结果 . . . 18

7 project9 18
7.1 实验原理 18
7.2 实验内容 19
7.3 实验环境 19
7.4 运行结果 19

8 project10 20
8.1 相关知识 20
8.1.1 椭圆曲线 20
8.1.2 ECDSA 20
8.1.2.1 公私钥生成 20
8.1.2.2 计算签名 20
8.2 ESDSA 签名恢复原理 21
8.3 ESDSA 签名恢复的过程 22
8.3.1 产生密钥 GenKey 22
8.3.2 签名算法 Sign 22
8.3.3 验证算法 Verify 22
8.3.4 恢复算法 Recover 23
8.4 以太网公钥恢复的意义 23
8.5 运行结果 23

9 Project11 24
9.1 实验环境 26
9.2 运行结果 27

10 Project12 27
10.1 运行环境 27
10.2 运行结果 27

11 Project13 28
11.1 实验原理 28
11.2 实验环境 29
11.3 实验结果 29

12 Project14 30
12.1 实验原理 30
12.2 实验内容 30
12.3 实验环境 32
12.4 实验结果 32

13 Project15 32
13.1 实验原理 32
13.2 实验内容 32
13.3 实验指导 32
13.4 实验结果 33

14 Project16 33
14.1 实验原理 33
14.2 实验内容 34
14.3 实验指导 34
14.4 实验结果 34

15 Project17 34

16 Project18 36

17 Project19 38
17.1 实验原理 38
17.2 实验内容 38
17.3 实验指导 38
17.4 实验结果 39

18 Project21 39
18.1 实验环境 40
18.2 实验结果 40

19 Project22 40
19.1 Trie 41
19.2 Patricia Trie 42

19.3 Merkle Tree 43
19.4 MPT 44
19.5 MPT 结构 45
19.6 安全的 MPT 48
19.7 总结 49

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]” (birthday problem)：一个班级需要有多少人，才能保证每个同学的生日都不一样？

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足够大，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，至少有两个人生日相同的概率，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 ex 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。

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

$$p(n)=1-\bar{p}(n)\approx 1-e^{-\frac{n(n-1)}{730}}.$$

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/zhizhengguan/article/details/112846817
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:nFBWDV70gw1x4Yz
data(in hex) = 9161a9e809ec91693aad4fdbc325b08da6a353e4fd6b923d28531b20b3ed
find_data(in hex) = 6e4642574456373067776c7834597a3deff60d59ede0acdc6b692dd
H1  = 9161a9e809ec91693aad4fdbc325b08da6a353e4fd6b923d28531b20b3ed0971
H1_  = 9161fac55fd1f2642dee8f6e31d1b08236de1c2f1d077f7b046e1efefaa7cbcf
Running time = 0.148000 seconds
```

```
Find the collision!(24 bits)
Initial string:DZSvAIHR0FL5YOW
data(in hex) = 697e666c69d4b7d73d078fabb38d245d32f84a63f2aa434c6fe300c049ba072b
find_data(in hex) = 5855446263456674704474516667551fb040edeef10bd19eead550ef84c18efd6
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。

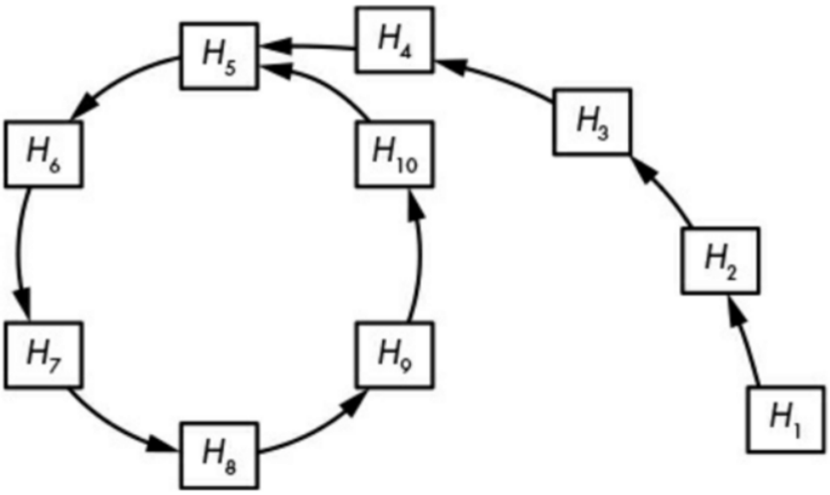
Rho 攻击（来自 Pollard Rho 算法），流程如下

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 H1, 设 $H1' = H1$

2. 计算 $H2 = Hash(H1), H2' = Hash(Hash(H1'))$

3. 迭代该过程并计算 $Hi + 1 = Hash(Hi)\ Hi + 1' = Hash(Hash(Hi'))$，直到有一个 i 可以满足 $Hi + 1 = Hi + 1'$

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

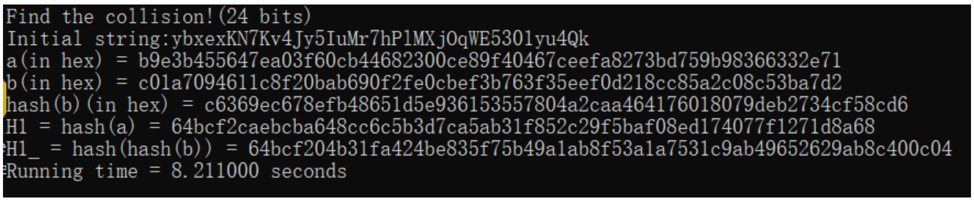
按照实验原理描述实现即可。

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

24 比特运行结果截图



4 Project3

4.1 实验原理

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

当知道 hash(message) 的值及 message 长度的情况下，可以推算出 hash(message||padding||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

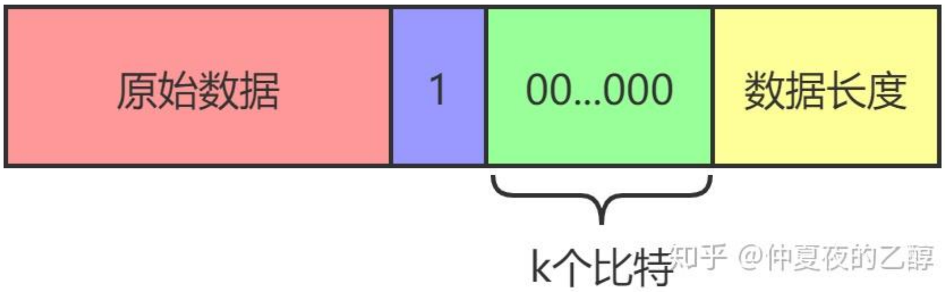
消息填充

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) mod 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：$m = B^{(0)}B^{(1)} \cdots B^{(n-1)}$ 其中 n=(l+k+65)/512

对 m 按下列方式迭代：

$$FORi = 0TOn - 1V^{(i+1)} = CF(V^{(i)}, B^{(i)})ENDFOR$$

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 $\{W_j\}$ 64 $\{W'_j\}$ j 0

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)

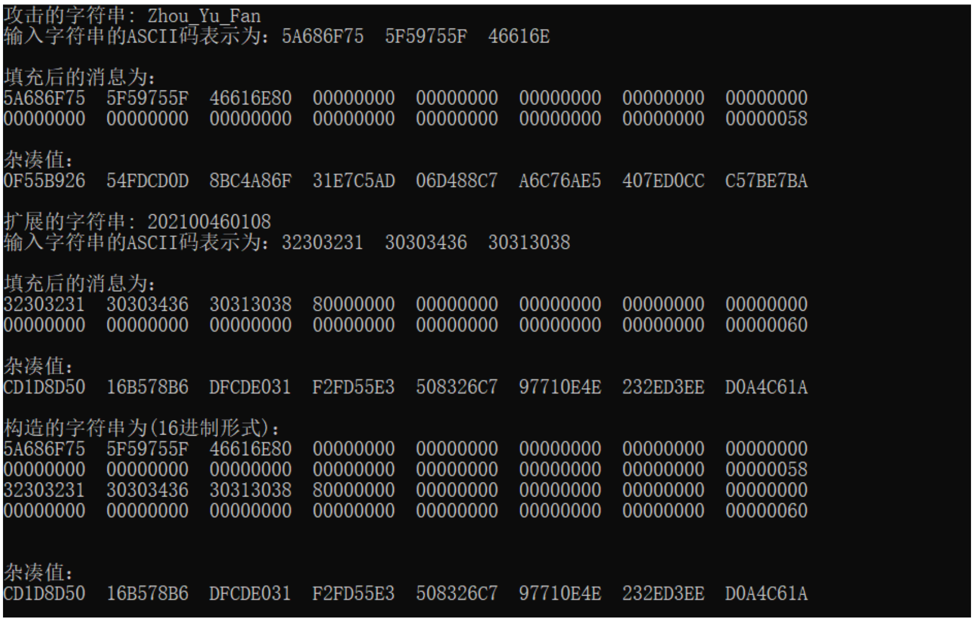
2. 生成一个附加消息 (m)。首先用 hash1 推算出这一次加密结束后 8 个向量的值，再以它们作为初始向量，去加密 m，得到另一个 hash 值 (hash2)

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



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$_{h}ualuo/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。

5.1 实验内容

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。

布尔函数

$$FF_j(X,Y,Z)=\begin{cases} X\oplus Y\oplus Z & 0\leqslant j\leqslant 15 \\ (X\wedge Y)\vee(X\wedge Z)\vee(Y\wedge Z) & 16\leqslant j\leqslant 63 \end{cases}$$

$$GG_j(X,Y,Z)=\begin{cases} X\oplus Y\oplus Z & 0\leqslant j\leqslant 15 \\ (X\wedge Y)\vee(\neg X\wedge Z) & 16\leqslant j\leqslant 63 \end{cases}$$

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。

```c
        #define FF0(x,y,z)  (x ^ y ^ z)
#define FF1(x,y,z)  ((x & y) | (x & z) | (y & z))

#define GG0(x,y,z)  (x ^ y ^ z)
#define GG1(x,y,z)  ((x & y) | ((~x) & z))
```

置换函数

$$P_0(X)=X\oplus(X<<<9)\oplus(X<<<17)$$
$$P_1(X)=X\oplus(X<<<15)\oplus(X<<<23)$$

消息扩展

5.3.2 消息扩展

将消息分组 $B^{(i)}$ 按以下方法扩展生成 132 个消息字 $W_0,W_1,\cdots W_{67},W'_0,W'_1,\cdots W'_{63}$，用于压缩函数 CF：

a) 将消息分组 $B^{(i)}$ 划分为 16 个字 $W_0,W_1,\cdots W_{15}$。

b) **FOR** $j=16$ **TO** 67

$W_j\leftarrow P_1(W_{j-16}\oplus W_{j-9}\oplus(W_{j-3}<<<15))\oplus(W_{j-13}<<<7)\oplus W_{j-6}$

ENDFOR

c) **FOR** $j=0$ **TO** 63

$W'_j=W_j\oplus W_{j+4}$

ENDFOR

注意到，上面的过程均是对于 32 位字进行独立的操作，相互之间并没有依赖关系，因此可以使用 SIMD 同时处理多组数据。

但重点在于，消息扩展中使用了 w[j-3]，w[j-6] ... 来计算 w[j]。跨度最小的是当前位置向前三个，这样的时序关系限制了最大的并行度。也就是说，由于需要使用前面的状态，最多只能同时处理三组数据。一旦超过三，则第四个 32 位字 w[j+3] 的计算就需要用到尚未得到的 w[j]，而这是不可行的。

因此使用 128 位的 SIMD 指令集是最合适的，每次同时处理三个 32 位字，只会空出一个字的位置。使用更大的宽度是没有意义的。

压缩函数

5.3.3 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 $V^{(i+1)}=CF(V^{(i)},B^{(i)})$，$0\leqslant i\leqslant n-1$。计算过程描述如下：

$ABCDEFGH \leftarrow V^{(i)}$

FOR $j=0$ **TO** 63

$SS1 \leftarrow ((A<<<12)+E+(T_j<<<(j \bmod 32)))<<<7$

$SS2 \leftarrow SS1 \oplus (A<<<12)$

$TT1 \leftarrow FF_j(A,B,C)+D+SS2+W'_j$

$TT2 \leftarrow GG_j(E,F,G)+H+SS1+W_j$

$D \leftarrow C$

$C \leftarrow B<<<9$

$B \leftarrow A$

$A \leftarrow TT1$

$H \leftarrow G$

$G \leftarrow F<<<19$

$F \leftarrow E$

$E \leftarrow P_0(TT2)$

ENDFOR

$V^{(i+1)} \leftarrow ABCDEFGH \oplus V^{(i)}$

其中，字的存储为 big-endian 格式，左边为高有效位，右边为低有效位。

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：

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

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

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

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

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

这样，下一次调用

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

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。

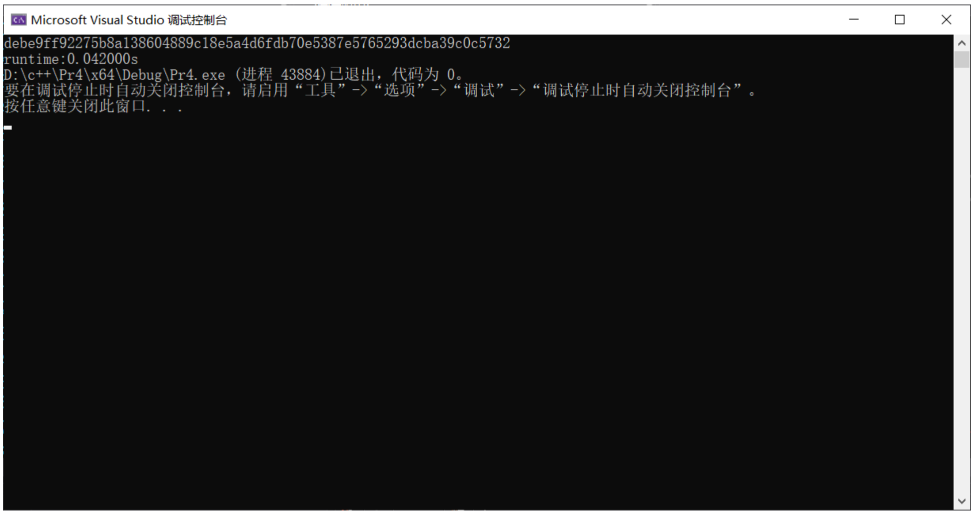
```
#define FF0(x,y,z)  (x ^ y ^ z)
#define FF1(x,y,z)  ((x & y) | (x & z) | (y & z))

#define GG0(x,y,z)  (x ^ y ^ z)
#define GG1(x,y,z)  ((x & y) | ((~x) & z))
```

5.2 运行结果

最终结果见下图。这里使用了国密局 SM3 文档中给出第二组的示例，对比确认了算法的正确性。

为了方便测算时间，重复进行了 10^6 次 sm3 运算，总计用时 2.853s。

平均每次计算 512bit（共 2 个分组，1024bit）消息需要的时间仅约 3 s 的时间，吞吐量达到了 21.39MB/s。



5.3 运行指导

其中项目默认可能是 64 位模式，如果不能运行，将上图位置改成 x86，即 32 位模式。

为了提高输出结果的速度，最终将测试次数降低至 10000 次，时间缩短了 100 倍。

6 project5

6.1 实验内容

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

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: 356fb16c30d2986ecadbf768a9df00f36f43f688e9099b174fe2bb27f8e42ca9   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

7.1 实验原理

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是轮密钥的使用顺序相反。以下是 SM4 的结构图：

SM4 的分组长度为 4 字，因此，其输入是 4 字的明文 (X0, X1, X2, X3)，输出是 4 字密文 (Y0, Y1, Y2, Y3), 经过 32 轮轮函数迭代，即每轮计算 $Xi+4 = Xi \oplus T(Xi+1 \oplus Xi+2 \oplus Xi+3 \oplus rki)$

经过 32 轮得到 (X32, X33, X34, X35)，经过一次反序变换后，得到最终的密文 (Y0, Y1, Y2, Y3) = (X35, X34, X33, X32) 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。

密钥扩展算法

将 128 比特的加密密钥 MK 扩展成 32 个 32 比特的轮子密钥，记为 rki(i = 0, ⋯, 31), 原始 4 字加密密钥为 MK = (MK0, MK1, MK2, MK3), 每个 MKi 都代表一个字

密钥扩展算法首先需要让原始密钥 MK 与系统参数 FK 异或，得到 k0, k1, k2, k3,, 每个 ki 都代表一个字，其中系统参数 FK 为 (0xA3B1BAC6, 0x56AA3350, 0x677D9197, 0xB27022DC)

之后，与加密过程类似，需要对这四个字进行 32 轮迭代，生成 32 个轮密钥。计算方法为：rki = ki+4 = ki $\oplus L\ S(ki{+}1{\oplus}ki{+}2{\oplus}ki{+}3{\oplus}cki)\quad L\ (B) = B{\oplus}(B\ 13){\oplus}(B\ 23), cki =$

$(cki,0,cki,1,cki,2,cki,3),cki,j=(4i+j)\times 7mod256$

解密时，轮密钥的生成方法与加密一致，只是需要逆序使用，因此不再赘述。

7.2 实验内容

按照实验原理中对 SM4 的概述，我们按照加解密过程、轮函数的实现、密钥扩展方案三部分来实现。并且在具体实现时参考了 Openssl 库的开源代码。SM4 的加密过程采用了 ECB 模式，即每次加密一个明文模块，将得到的密文模块组合得到最终的密文。采用 ECB 模式可以有效提高加密效率。我们将分组大小设为了 16 个字节，注意的是在划分最后一个分组后，要确定之后是否还有剩余的但不满足分组长度的明文，对于有不满足分组长度的明文分组要对其进行填充。之后便依次对每个分组进行 SM4 加密。SM4 的加密过程中共需要经过 32 轮迭代，我们将 4 轮迭代看作 1 轮，这样一次就能计算出 4 个加密的密文字。为了提高效率，我们在仅第一次迭代和最后一次迭代中使用非线性变换 S、线性变换 L 分开的方式，即 $SM4\_T\_slow$。中间的迭代，我们将 S 和 L 结合成 1 个模块 $SM4\_T$，通过查表运算同时完成非线性变换和线性变换。

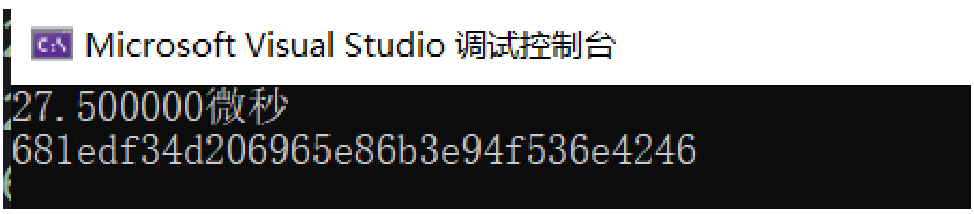
非线性变换 S 和线性变换 L 都是每次传入 1 个字，其中 $SM4\_T\_non\_lin\_sub$ 函数用于实现 S 盒，$SM4\_T\_slow$ 函数用于实现线性变换 L，rotl 函数用于实现循环左移.

在密钥扩展方案的实现中，我们用 $SM4_key_sub$ 函数实现非线性变换 S 和线性变换 L，用 $load\_u32\_b$ 函数表示取原始密钥的 1 个字，K 是新的密钥字，rk 是轮密钥，循环中每轮经过线性变换和非线性变换以及异或原始密钥 K[i] 得到新的密钥 K[i+4]，将其作为轮密钥。每次循环可生成 4 个轮密钥，而不是 1 次生成 1 个，加快了密钥扩展的速度。

7.3 实验环境

visual studio 2019

7.4 运行结果



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：$y^2 = x^3 + ax + b \ (mod\ p)$。

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 $r = x \ (mod \ n)$

计算消息 m 的 hash 值 H(m)

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

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：

$$tonellishanks(r, P) == R_y$$

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=hash(m)

生成随机数 $k \in n$，计算点 (x, y)=k G

取 r=x mod n，若 r=0 则重新选择随机数 k

计算 $s = k^{-1}(z + rd)modn$，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 $z = hash(m)$

计算 $u_1 = zs^{-1}modn$ $u_2 = rs^{-1}modn$

计算 $(x, y) = u1\ G + u2\ Qmodn$

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⋯，代入椭圆曲线方程计算获得 R

计算 z = hash(m)

计算 $u_1 = -zr^{-1} modn$ $u_2 = sr^{-1} modn$

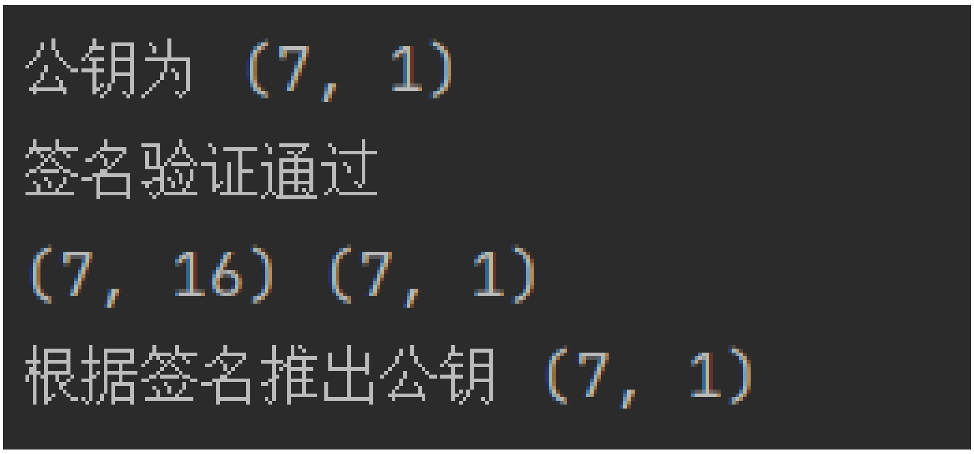
计算公钥 $Q = (x', y') = u_1\, G + u_2\, R$

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... 0x01，长度为 hlen

3. 设置 K = 0x00 0x00... 0x00，长度为 heln

4. 计算 K = HMAC(V||0x00||int2octets(x)||bits2octets(h1))

5. V = HMAC(V)

6. K = HMAC(V||0x01||int2octets(x)||bits2octets(h1))

7. V = 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 = HMAC(V)

* T = T||V

* 计算 k=bits2int(T),k [1,q-1]，则可输出，否则计算

* K = HMAC(V||0x00)

* V = 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 ... 0x01

such that the length of V, in bits, is equal to 8*ceil(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 ... 0x00

such that the length of K, in bits, is equal to 8*ceil(hlen/8).

d. Set:

K = HMAC$_K(V||0x00||int2octets(x)||bits2octets(h1))$

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 = HMAC$_K(V)$

f. Set:

K = HMAC$_K(V||0x01||int2octets(x)||bits2octets(h1))$

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

g. Set:

V = 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, tlen = 0.

2. While tlen < qlen, do the following:

V = HMAC$_K(V)$

T = T || V 3. Compute:

k = 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 = HMAC$_K(V||0x00)$

V = 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：

```
def deterministic_generate_k(msghash, priv):
    v = b'\x01' * 32
    k = b'\x00' * 32
    k = hmac.new(k, v+b'\x00'+priv+msghash, hashlib.sha256).
        digest()
    v = hmac.new(k, v, hashlib.sha256).digest()
    k = hmac.new(k, v+b'\x01'+priv+msghash, hashlib.sha256).
        digest()
    v = hmac.new(k, v, hashlib.sha256).digest()
    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 实验环境

python 3.10

需要提前安装 gmssl、hashlib 等库。

9.2 运行结果

```
k: 100513257062470343777232823561226136929544702759822580706321962097859599956780
加密结果 b'\xcb\t\xf4\x8b:\x15\xa5]\xed\x8c\xf7\xe9\xeb\xfdqm@\xc7\xc5\xe9\xdb?!\xfa\xd0\xf2\x8b\x98iuZR\xf6\xb6~\xbe\xc4\xca\xb1\xd3`\x9a_\xf5\t/\xad1\xb3MO)
 c|#\x9b\xca\x17\xbc\xd044\x0c\xec\xc8Z6(\xa3\x00u\x8f\xe7\x1e\xaa\x9c?w\x877\xf2\\LQ\x96\xda\xb6\x9f\xec\xa9\xb3\x16.\x86\x81\x9f\x1f\xbco\xa8\x86\x95\x9a\x13;\xcf\x99\xf1'
解密结果 b'202100460108'
加解密一致
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	✓	✓	$r=(e+x_1)\ mod\ n$ $e=Hash(Z_A\|\|M)$
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

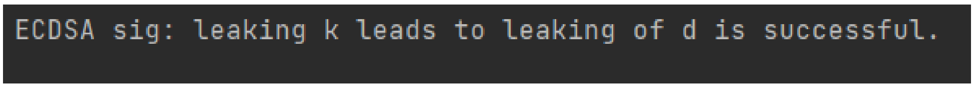
需要提前安装 gmssl 等库

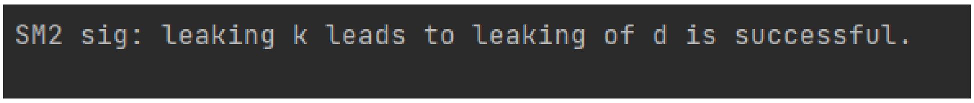
10.2 运行结果

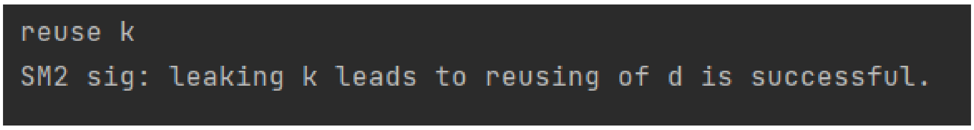
下面是 ECDSA、schnorr、sm2 的运行结果：

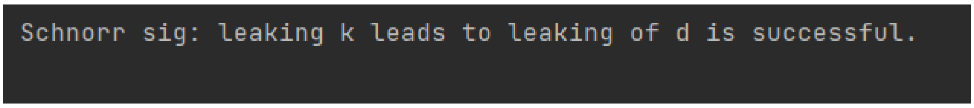
```
Success!
```

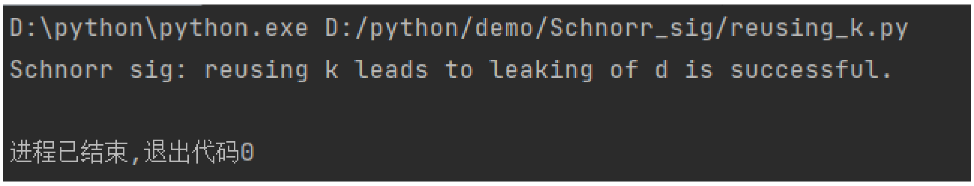
```
reuse k
ECDSA sig: reusing k leads to leaking of d is successful.
```











11 Project13

11.1 实验原理

ECMH 用于 hash 单个元素，‘ECMH$_s et$‘ $hash$ $\mathbb{F}_{2^n}^*$ 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 整体思路是：先把集合里的元素映射成椭圆曲线上的点，然后利用椭圆曲线上的加法求解哈希值。

为达到相同的安全性，ECMH 算法需要的密钥长度远远小于哈希求和算法，因而 ECMH 相较哈希求和算法更为安全。

ECMH 相关代码：

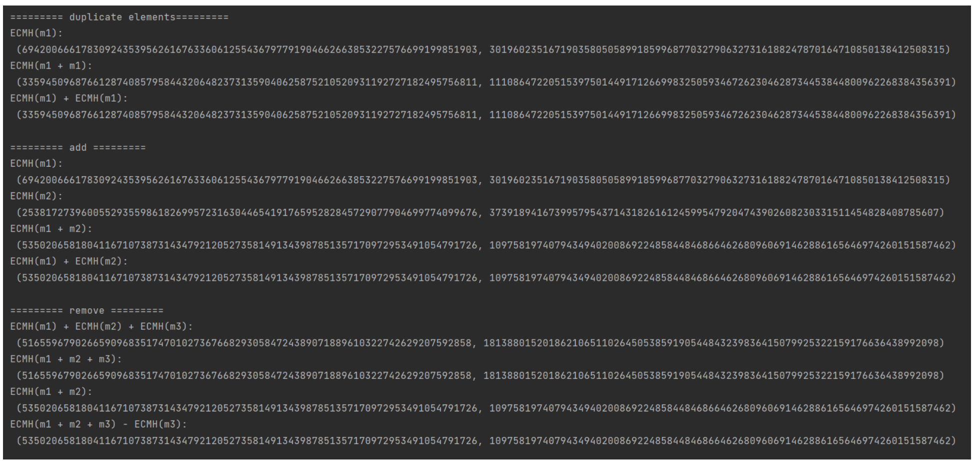
```
 MultiSet Hash -> EC combine/add/remove
def add(ecmh, msg):
    dot = msg_to_dot(msg)
```

```
    tmp = EC_add(ecmh, dot)
    return tmp
def single(msg):
    return add(0, msg)
def remove(ecmh, msg):
    dot = msg_to_dot(msg)
    tmp = EC_sub(ecmh, dot)
    return tmp
def combine(msg_set):
    ans = single(msg_set[0])
    num = len(msg_set) - 1
    for i in range(num):
        ans = add(ans, msg_set[i + 1])
    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):
 (694200666178309243539562616763360612554367977791904662663853227576699199851903, 30196023516719035805058991859968770327906327316188247870164710850138412508315)
ECMH(m1 + m1):
 (33594509687661287408579584432064823731359040625875210520931192727182495756811, 111086472205153975014491712669983250593467262304628734453844800962268384356391)
ECMH(m1) + ECMH(m1):
 (33594509687661287408579584432064823731359040625875210520931192727182495756811, 111086472205153975014491712669983250593467262304628734453844800962268384356391)

========= add =========
ECMH(m1):
 (694200666178309243539562616763360612554367977791904662663853227576699199851903, 30196023516719035805058991859968770327906327316188247870164710850138412508315)
ECMH(m2):
 (25381727396005529355986182699572316304465419176595282845729077904699774099676, 3739189416739957954371431826161245995479204743902608230331511454828408785607)
ECMH(m1 + m2):
 (53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020086922485844846866462680960691462886165646974260151587462)
ECMH(m1) + ECMH(m2):
 (53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020086922485844846866462680960691462886165646974260151587462)

========= remove =========
ECMH(m1) + ECMH(m2) + ECMH(m3):
 (5165596790266590968351747010273676682930584724389071889610322742629207592858, 18138801520186210651102645053859190544843239836415079925322159176636438992098)
ECMH(m1 + m2 + m3):
 (5165596790266590968351747010273676682930584724389071889610322742629207592858, 18138801520186210651102645053859190544843239836415079925322159176636438992098)
ECMH(m1 + m2):
 (53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020086922485844846866462680960691462886165646974260151587462)
ECMH(m1 + m2 + m3) - ECMH(m3):
 (53502065818041167107387314347921205273581491343987851357170972953491054791726, 109758197407943494020086922485844846866462680960691462886165646974260151587462)
```

12 Project14

12.1 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。

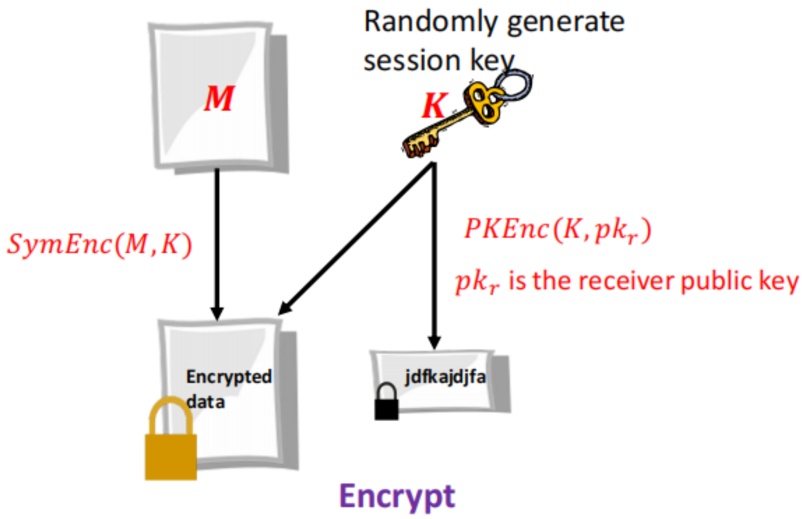
12.2 实验内容

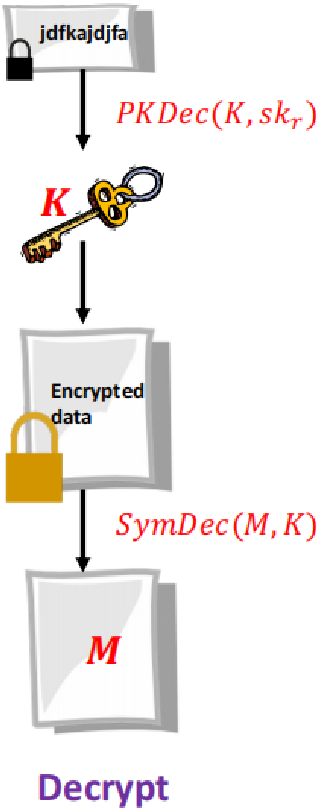
本次实验旨在实现一个简易 PGP，调用 GMSSL 库中封装好的 SM2/SM4 加解密函数。

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 $defepoint\_mod(a, n)$

定义椭圆曲线上的模运算，返回值等于 a mod n

$defepoint\_modmult(a, b, n)$

定义椭圆曲线上的模乘运算，返回值等于 $a * b^{-1} modn$

$defepoint\_add(P, Q, a, p)$

定义椭圆曲线上的加法运算，返回值等于 $P + Q$

$defepoint\_mult(k, P, a, p)$

定义椭圆曲线上的点乘运算，返回值等于 $k * P$

def keygen(a, p, n, G)

生成 SM2 算法的公私钥对

$defpgp\_enc(m, k)$

PGP 加密算法

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.

12.3 实验环境

python 3.10

需要安装 Crypto、gmssl、random 等库。

12.4 实验结果

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。

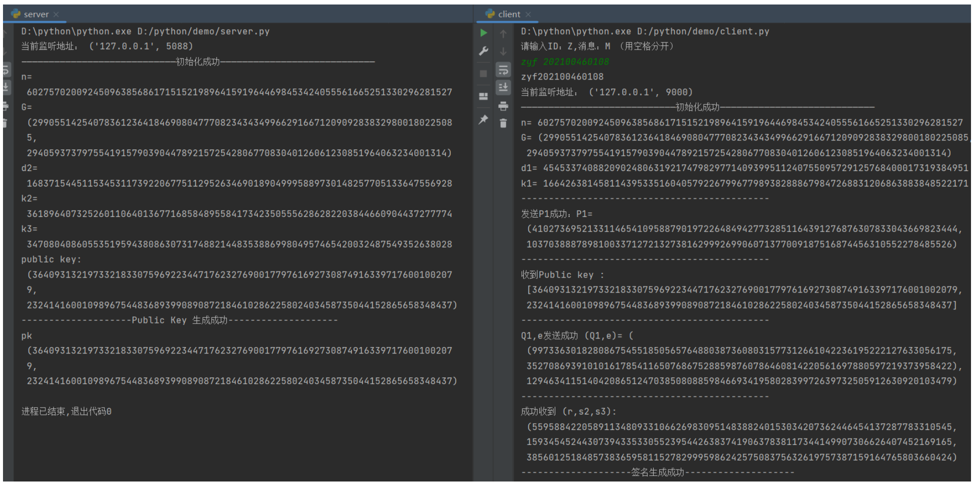
三个 python 文件要放在同一目录下.

ECC.py 中是参数定义和基本函数。

server.py 负责接收 client 的请求，生成并保存子私钥（sub private key）之一的 d2，完成一些辅助 client 签名的计算，并向其发送辅助的数据；

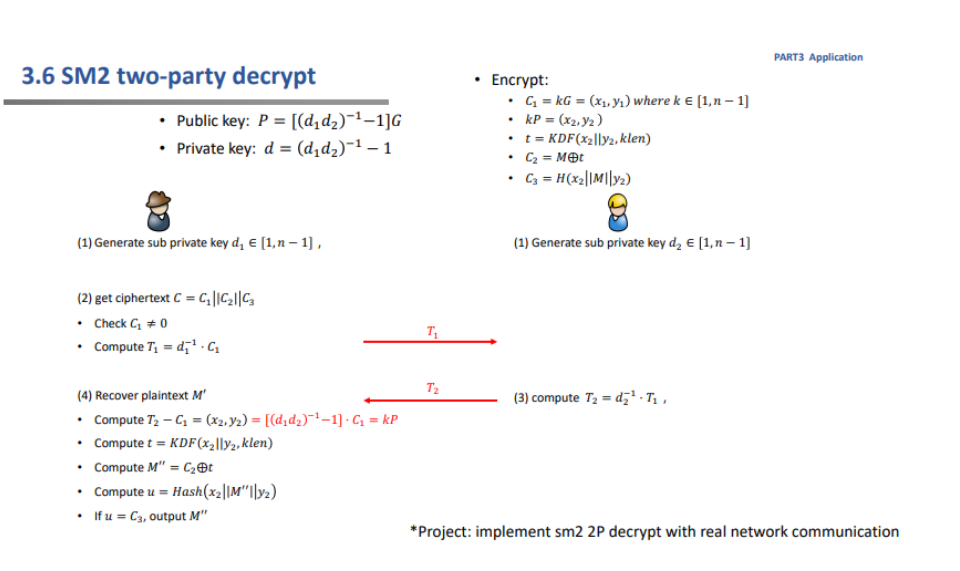
client.py ：生成并保存子私钥（sub private key）之一的 d1，请求 server 完成需要的辅助工作，并针对消息 M 完成 SM2 2P sign；

13.4 实验结果



14 Project16

14.1 实验原理



首先 A 需要与 B 协商共同的公钥，此阶段称为 prepare。

由 A 发起一系列运算，生成自身的私钥，计算 P_1 发送给 B。

B 根据 mode 指定的需求做出应答。

首先接收 P_1。计算自己的私钥，生成 P_2，计算 $P = d_2^{-1}P_1 + nG\ (mod\ p)$，将 P 发送给 A。

该阶段完成后，双方各自计算得到了同样的公钥 P。

然后，A 使用公钥 P 加密一段信息，得到密文

为了解密，A 需要向 B 提出合作解密的请求。同理，B 做出回复。最终，A 可以获得明文 M。

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

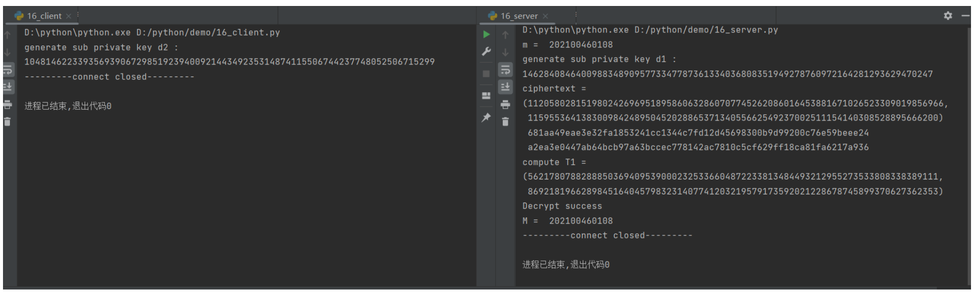
python 3.10

先运行 server.py 文件，再运行 client.py 文件。

server.py ：负责接收 client 的请求，生成并保存子私钥之一的 d2，计算并公布双方子私钥对应的公钥，完成一些辅助 client 解密的计算，并向其发送辅助的数据

client.py ：生成并保存子私钥之一的 d1，请求 server 完成需要的辅助工作，并针对使用 server 公布的公钥加密的密文完成 SM2 2P decrypt；

14.4 实验结果



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$_value$”

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 PCKS11 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

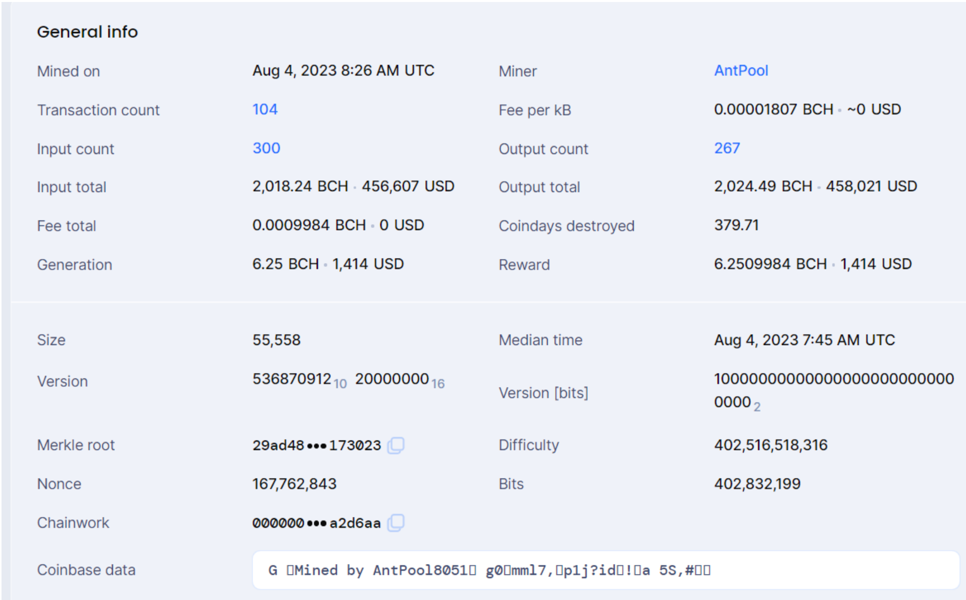
参考：(https://security.stackexchange.com/questions/41029/comparison-between-firefox-password-manager-and-chrome-password-manager)

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]	$1000000000000000000000000000000_2$
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 AntPool8051 g0mml7,p1j?id!a 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

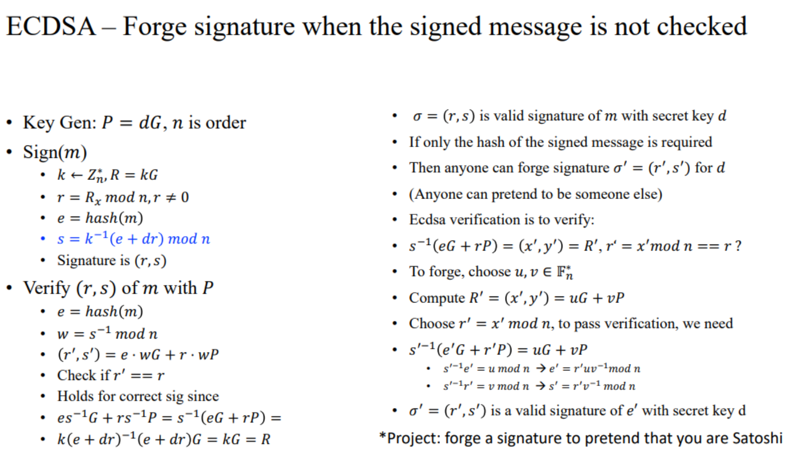
Nonce：矿工必须解决的加密数字，以验证区块。nonce 用于验证块中包含的信息，生成一个随机数，将其附加到当前标头的散列中，重新散列该值，并将其与目标散列进行比较。

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



17.2 实验内容

伪造签名过程

1、重新选择 a、b，计算:

$$(x2, y2) = a * G + b * Q_A$$

2、计算:

$$r1 = x2modnb1 = b^{-1}modne1 = r1 * a * b1modns1 = r1 * b1modn$$

(r1,s1) 即为伪造的签名。此外，该算法为概率算法，成功概率与选取的随机数有关。

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 epoint$_a dd(P, Q)$

epoint$_m ul(k, g)$

def signature(m) ECDSA 签名算法

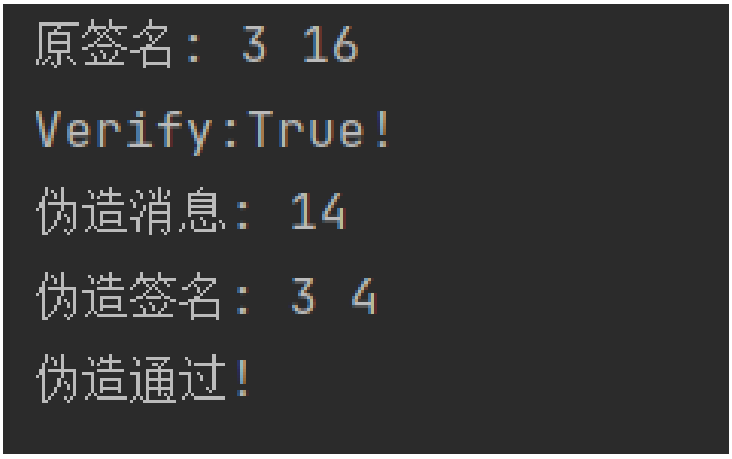
def verify(r,s) ECDSA 验签算法

def forge$_s ignature(r, s)$

17.3 实验指导

python 3.10

17.4 实验结果



18 Project21

Schnorr 数字签名算法简单流程

Setup:

$$
\begin{aligned}
&\mathrm{x} := \text{ random number} \qquad \text{(aka private key)}\\
&\mathrm{G} := \text{ common point}\\
&\mathrm{X} := \mathrm{x}^{*}\mathrm{G} \qquad \text{(aka public key)}
\end{aligned}
$$

Sign:

$$
\begin{aligned}
&r := \mathit{random} \text{ number (aka nonce)}\\
&R := r * G \qquad \text{(aka commitment)}\\
&e := \text{ Hash } R, X, \text{message)(aka challenge)}\\
&\mathrm{s} := r + e * x \qquad \text{(aka response)}\\
&\text{return } (R, X, s, \text{message}) \qquad ((s, e) \text{ aka signature})
\end{aligned}
$$

Verify:

$$
\begin{aligned}
&\text{receive } (R, X, s, \text{ message })\\
&\mathrm{e} \ := \text{Hash}(R, X, \text{message })\\
&\mathrm{s1} := R + e * X\\
&\mathrm{s2} := \mathrm{s} * \mathrm{G}\\
&\text{return OK if S1 qeuals S2}
\end{aligned}
$$

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 $s_1 = r_2 + e_1 x_1 - s_2 \; mod \; p$
  - 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 实验环境

python 3.10

18.2 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

MPT (Merkle Patricia Tries) 是以太坊存储数据的核心数据结构，它是由 Merkle Tree 和 Patricia Tree 结合的一种树形结构，理解 MPT 有助于我们更好的理解以太坊的数据存储。

在了解 MPT 数据结构之前，需要先介绍基本的 Tree 结构和 Merkle Tree、Patricia Tree。

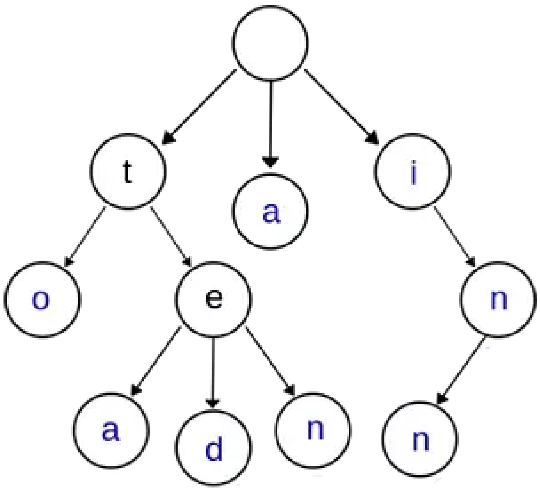
19.1 Trie

Trie 树，又称前缀树或字典树，是一种有序树，用于保存关联数组，其中的键通常是字符串。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。Trie 的核心思想就是用空间换时间，利用公共前缀缩小要比较的范围来达到快速查找的目的。

相比于哈希表，使用前缀树来进行查询拥有共同前缀 key 的数据时十分高效，例如在字典中查找前缀为 pre 的单词，对于哈希表来说，需要遍历整个表，时间效率为 O(n)，然而对于前缀树来说，只需要在树中找到前缀为 pre 的节点，且遍历以这个节点为根节点的子树即可。

但是对于最差的情况（前缀为空串），时间效率为 O(n)，仍然需要遍历整棵树，此时效率与哈希表相同。

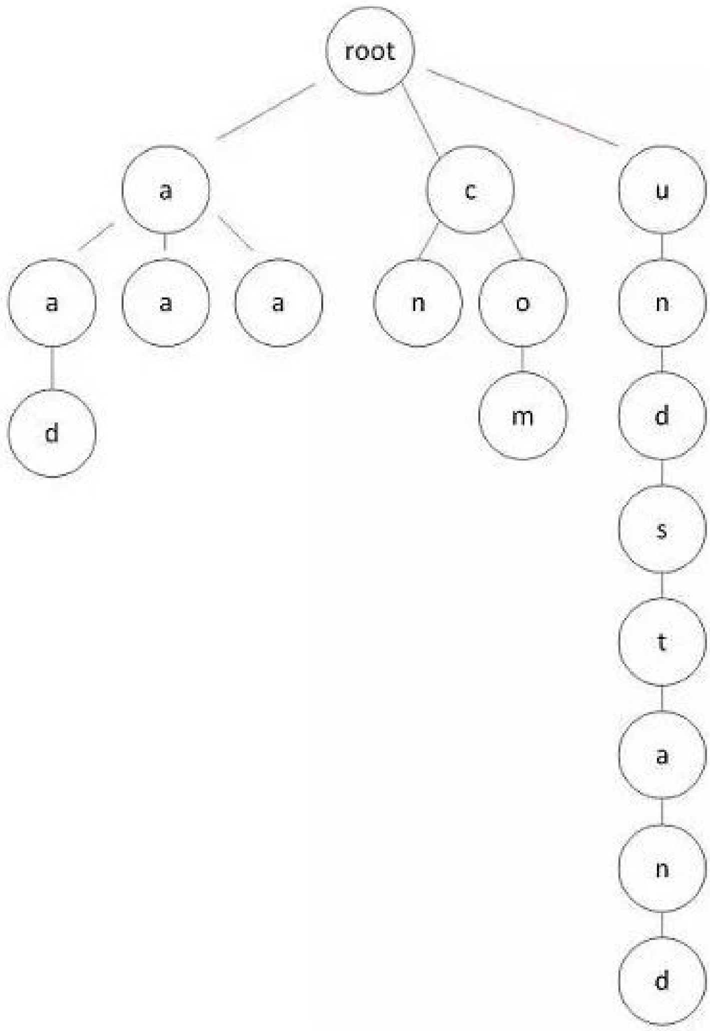
相比于哈希表，在前缀树不会存在哈希冲突的问题。



上图是一棵 Trie 树，表示了字符串集合“a”，“to”，“tea”，“ted”，“ten”，“i”，“in”，“inn”，从上图中我们可以看出 Trie 树的特点：

- 根节点不包含字符，除根节点外的每一个子节点都包含一个字符。- 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。- 每个节点的所有子节点包含的字符互不相同。

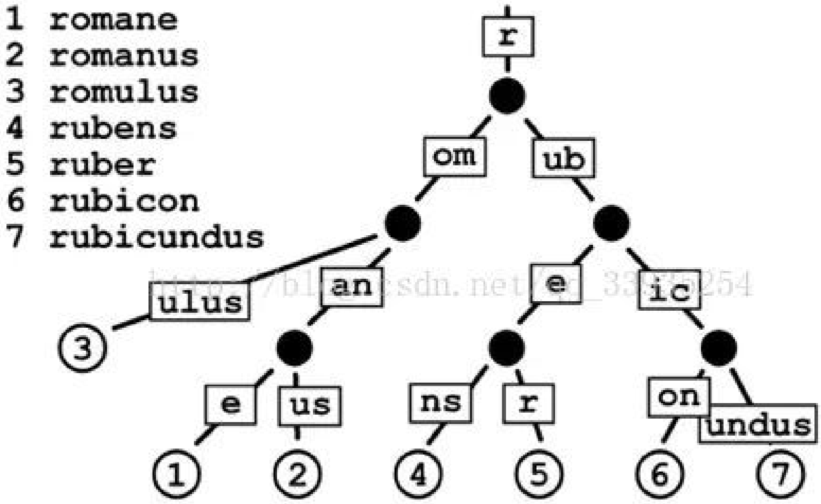
然而，这样的结构存在缺陷。当存在少量的长字符串，且其某个较长前缀下只有本身一个元素时，树的高度会很大，且一条长路径上只有一个叶节点。这样极大地浪费了存储空间，且应用起来效率也不高。下图展示了这样的状态：



为了解决这个问题，产生了 Patricia 树 (压缩前缀树)。

19.2 Patricia Trie

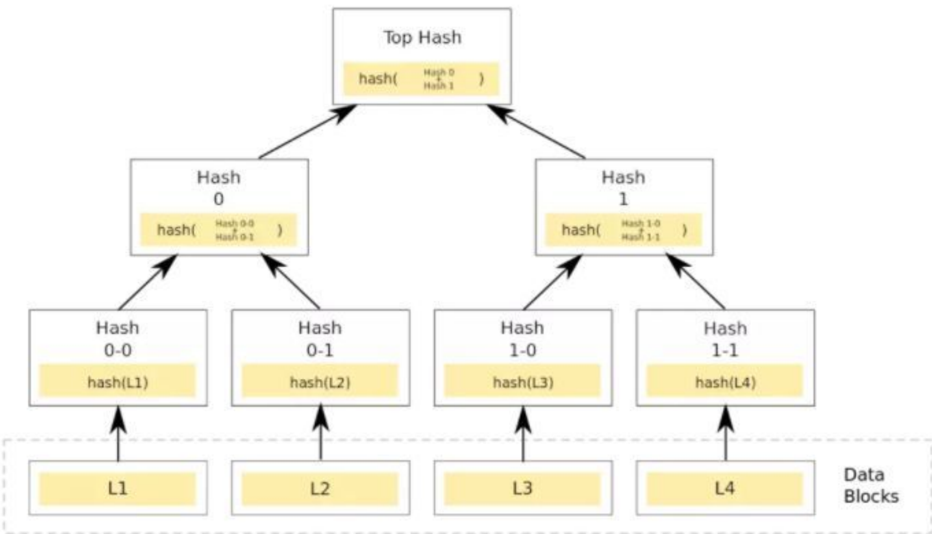
压缩前缀树，是一种更节省空间的 Trie。对于树的每个节点，如果该节点是唯一的子节点，就和父节点合并。



19.3 Merkle Tree

Merkle 树，主要用于数据集较大时的文件校验。其主要特点为：

- 叶节点存储数据块的 Hash（如：文件块、一段数据集）- 非叶子节点 (包括中间节点和根节点) 存储着对应子节点 Hash 值串联字符串之后的 Hash 值。



从上图中可以看出：

- 在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应地哈希和它对应；- 往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希。- 最终到达根节点，生成了可以代表整棵树的 hash 值，该节

点成为叫做 Merkle Root。

该数据结构的应用可以通过一个实例理解。

在 P2P 下载中，文件会切分成很多小的数据块，每个数据块从不同的来源上下载，这些来源可能是不稳定或不可信的。文件下载完成后，需要校验文件完整性。

以往一般的做法是对于整个文件计算 hash 值并与正确的值进行对比。然而这需要完整地下载整个文件，并且一旦产生错误，也很难找出是哪一个部分出现问题，这使得下载的可靠性和效率都很差。而 Merkle Tree 就可以解决这个问题。

在 P2P 网络下传输文件之前，先从可信的源获得文件的 Merkle Root，然后就可以从其他不可信的源获取 Merkle tree。通过可信的 Merkle Root 来验证 Merkle tree 是否损坏或者是虚假的。文件合并之后，计算小数据块的 Hash 并最终计算根 Hash，对比与可信的根 Hash 是否一致。这样就避免了对整个文件进行 Hash 计算。

了解了这些基本概念，下面可以开始讨论报告的主题：MPT。

19.4 MPT

MPT，即 Merkle Patricia Tree。是一种经过改良的、融合了默克尔树和前缀树两种树结构优点的数据结构，是以太坊中用来组织管理账户数据、生成交易集合哈希的重要数据结构。

MPT 树有以下几个作用：

存储任意长度的 key-value 键值对数据，符合以太坊的 state 模型；

提供了一种快速计算所维护数据集哈希标识的机制；

提供了快速状态回滚的机制；

提供了默克尔证明，进行轻节点的扩展，实现简单支付验证；

MPT 树中的节点包括空节点、叶子节点、扩展结点和分支节点。

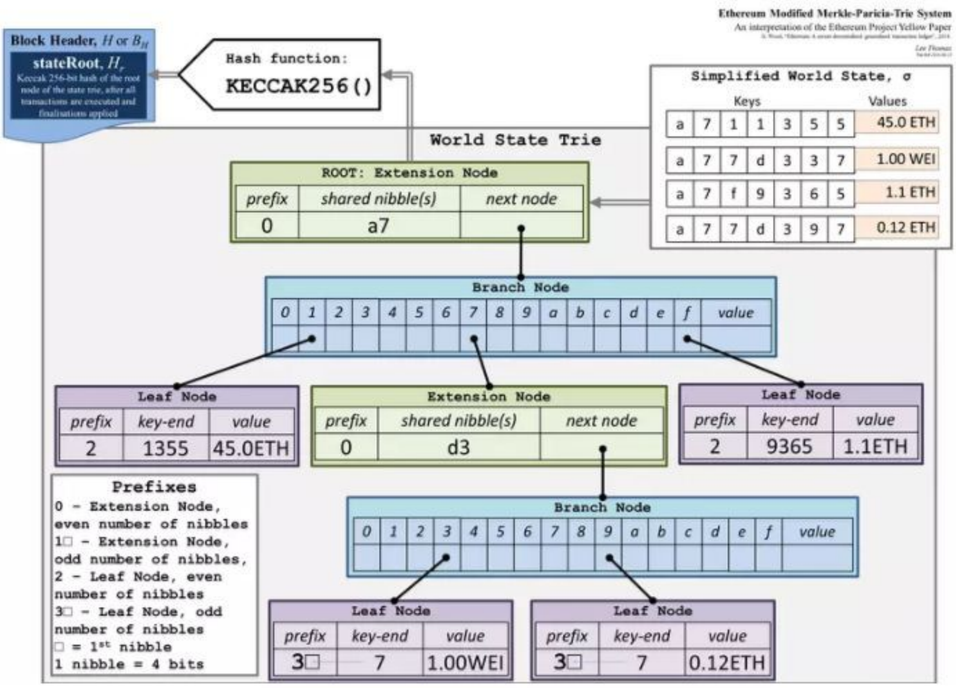
1、空节点（NULL）：表示为空，在代码中是一个空串。

2、叶子节点（leaf）：表示为 [key,value] 的一个键值对，其中 key 是 key 的一种特殊十六进制编码 (MP 编码)，value 是 value 的 RLP 编码。

3、拓展节点（extension）：也是 [key,value], 但是这里的 value 是其他节点的 hash，通过这个 hash 链接到其他节点。

4、分支节点（branch）：MPT 中的 key 被序列化成一种特殊的 16 进制编码，加上最后的 value，所以分支节点是一个长度为 17 的 list，前 16 个元素对应着 key 中的 16 个可能的十六进制字符。如果有一个 [key,value] 对在这个分支节点终止，则最后一个元素代表一个值 (例如，有三个 key，分别是（abc，abd，ab），则该节点的第 17 个字段存

储了 ab 节点的值）即分支节点既可以是搜索路径的终止也可以是路径的中间节点。对应于 Trie 中的关键路径节点。



MPT 中另外一个重要的概念是一个特殊的十六进制前缀 (hex-prefix, HP) 编码，用来对 key 进行编码。这种编码使用十六进制字符串，因此前缀的字母表也是 16 进制，每个节点可能有 16 个子节点。

HEX-Prefix 十六进制前缀编码：

若输入 key 结尾为 0x10，则去掉这个终止符。

key 之前填充一个二进制四元组，第 0 位区分奇偶信息，第 1 位区分节点类型。

如果输入 key 的长度是偶数，则再添加一个四元组 0x0 在 flag 四元组后。

将原来的 key 内容压缩，将分离的两个 hex 以高四位低四位进行合并成一个字节。

此外，注意到叶节点和扩展节点的存储结构完全相同。为了将 MPT 树存储到数据库中，同时还可以把 MPT 树从数据库中恢复出来，对于 Extension 和 Leaf 的节点类型做了特殊的定义：** 如果是一个扩展节点，那么前缀为 0，这个 0 加在 key 前面。如果是一个叶子节点，那么前缀就是 1**。

19.5 MPT 结构

MPT 树的特点如下：

叶子节点和分支节点可以保存 value, 扩展节点保存 key；

没有公共的 key 就成为 2 个叶子节点；key1=[1,2,3] key2=[2,2,3]

有公共的 key 需要提取为一个扩展节点；key1=[1,2,3] key2=[1,3,3] => ex-node=[1], 下一级分支 node 的 key

如果公共的 key 也是一个完整的 key，数据保存到下一级的分支节点中；key1=[1,2] key2=[1,2,3] =>ex-node=[1,2], 下一级分支 node 的 key; 下一级分支 =[3], 上一级 key 对应的 value

使用 MPT 的主要原因在于一个问题：设计在扩展节点的 Val 字段有可能存储一串哈希值作为孩子节点的索引，这一点符合 Trie 的特性。

在以太坊中，该哈希代表着另外一个节点在数据库中索引，即根据这个哈希值作为数据库中的索引，可以从数据库中读取出另外一个节点的内容。

这种设计的目的是：

（1）当整棵树被持久化到数据库中时，保持节点间的关联关系；

（2）从数据库中读取节点时，尽量避免不必要的 IO 开销；

在内存中，父节点与子节点之间关联关系可以通过引用、指针等编程手段实现，但是当树节点持久化到数据库是，父节点中会存储一个子节点在数据库中的索引值，以此保持关联关系。

同样，从数据库中读取节点时，本着最小 IO 开销的原则，仅需要读取那些需要用到的节点数据即可，因此若目前该节点已经包含所需要查找的信息时，便无须将其子节点再读取出来；反之，则根据子节点的哈希索引递归读取子节点，直至读取到所需要的信息。

节点分类

MPT 树中，树节点可以分为以下几类：空节点、分支节点、叶子节点、扩展节点

空节点

空节点用来表示空串。

分支节点

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

与前缀树相同，MPT 同样是把 key-value 数据项的 key 编码在树的路径中，但是 key 的每一个字节值的范围太大（[0-127]），因此在以太坊中，在进行树操作之前，首先会进行一个 key 编码的转换，将一个字节的高低四位内容分拆成两个字节存储。通过编码转换，key’的每一位的值范围都在 [0, 15] 内。因此，一个分支节点的孩子至多只有 16 个。以太坊通过这种方式，减小了每个分支节点的容量，但是在一定程度上增加了树高。

分支节点的孩子列表中，最后一个元素是用来存储自身的内容。

此外，每个分支节点会有一个附带的字段 nodeFlag，记录了一些辅助数据：

（1）节点哈希：若该字段不为空，则当需要进行哈希计算时，可以跳过计算过程而直接使用上次计算的结果（当节点变脏时，该字段被置空）；

（2）脏标志：当一个节点被修改时，该标志位被置为 1；

（3）诞生标志：当该节点第一次被载入内存中（或被修改时），会被赋予一个计数值作为诞生标志，该标志会被作为节点驱除的依据，清除内存中“太老”的未被修改的节点，防止占用的内存空间过多。

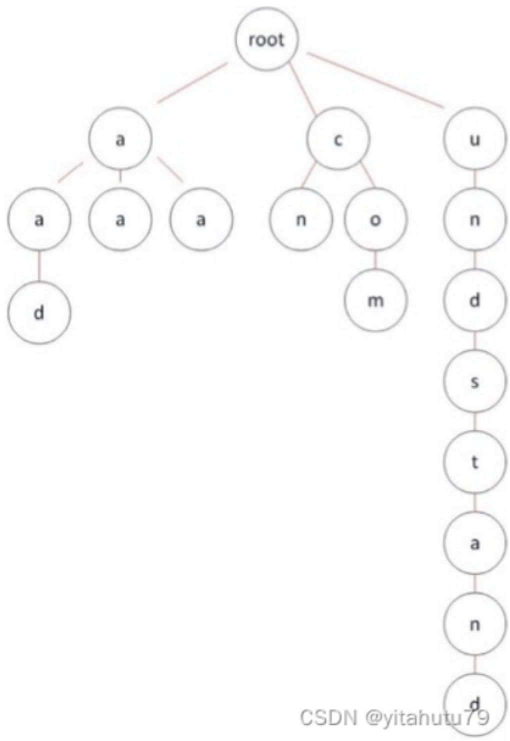
叶子节点和扩展节点

关键的字段为：

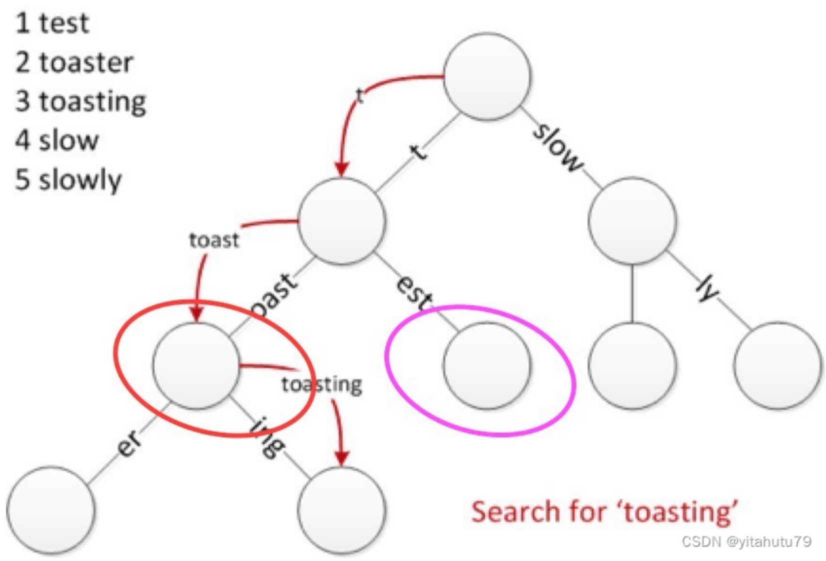
（1）Key：用来存储属于该节点范围的 key，这是 MPT 树实现树高压缩的关键；

（2）Val：用来存储该节点的内容；

如之前所提及的，前缀树中会出现严重的存储空间浪费的情况，如下图：



针对这种情况，MPT 树对此进行了优化：当 MPT 试图插入一个节点，插入过程中发现目前没有与该节点 Key 拥有相同前缀的路径。此时 MPT 把剩余的 Key 存储在叶子 / 扩展节点的 Key 字段中，充当一个” Shortcut “。



例如图中我们将红线所圈的节点称为 node1，将紫色线所圈的节点称为 node2。node1 与 node2 共享路径前缀 t，但是 node1 在插入时，树中没有与 oast 有共同前缀的路径，因此 node1 的 key 为 oast，实现了编码路径的压缩。

这种做法可以提高节点的查找效率，避免过多的磁盘访问，并且减少存储空间浪费，避免存储无用的节点。

19.6 安全的 MPT

以上介绍的 MPT 树，可以用来存储内容为任何长度的 key-value 数据项。倘若数据项的 key 长度没有限制时，当树中维护的数据量较大时，仍然会造成整棵树的深度变得越来越深，会造成以下影响：

（1）查询一个节点可能会需要许多次 IO 读取，效率低下；

（2）系统易遭受 Dos 攻击，攻击者可以通过在合约中存储特定的数据，“构造”一棵拥有一条很长路径的树，然后不断地调用 SLOAD 指令读取该树节点的内容，造成系统执行效率极度下降；

（3）所有的 key 其实是一种明文的形式进行存储。

为了解决以上问题，在以太坊中对 MPT 再进行了一次封装，对数据项的 key 进行了一次哈希计算，因此最终作为参数传入到 MPT 接口的数据项其实是 (sha3(key), value)

优势:

1、传入 MPT 接口的 key 是固定长度的（32 字节），可以避免出现树中出现长度很长的路径；

劣势:

1、每次树操作需要增加一次哈希计算;

2、需要在数据库中存储额外的 sha3(key) 与 key 之间的对应关系。

19.7 总结

MPT 结合了 Merkle Tree 和 Patricia Trie 的优势，概括如下：

可以实现快速重哈希，当树节点内容发生变化时，能够在前一次哈希计算的基础上，仅仅将被修改的树节点进行哈希重计算，便能得到一个新的根哈希用来代表整棵树的状态。

仍然可以进行轻节点扩展，验证一条交易只需要验证包含该交易的路径即可，并不需要把所有交易的 Hash 全部重新算一遍。

能够实现快速状态回滚。区块链公链的环境下，可能会造成分叉而导致区块链状态需要进行回滚，由于出块时间短，这种分叉的几率很大，区块链状态回滚的现象很频繁。所谓的状态回滚指的是：（1）区块链内容发生了重组织，链头发生切换（2）区块链的世界状态（账户信息）需要进行回滚，即对之前的操作进行撤销。

每个节点在数据库中的存储都是值驱动的。当一个节点的内容发生了变化，其哈希相应改变，而 MPT 将哈希作为数据库中的索引，也就实现了对于每一个值，在数据库中都有一条确定的记录。而 MPT 是根据节点哈希来关联父子节点的，因此每当一个节点的内容发生变化，最终对于父节点来说，改变的只是一个哈希索引值；父节点的内容也由此改变，产生了一个新的父节点，递归地将这种影响传递到根节点。最终，一次改变对应创建了一条从被改节点到根节点的新路径，而旧节点依然可以根据旧根节点通过旧路径访问得到。所以，在以太坊中，发生分叉而进行世界状态回滚时，只需要用旧的 MPT 根节点作为入口，就能完成一次“状态回滚”。