创新创业实践课程项目

学　　院：网络空间安全学院（研究院）

专　　业：密码科学与技术

组　　员：胡耀月

学　　号：202100460144

2023 年 8 月 4 日

目录

项目任务（已完成的加粗）

⋆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
⋆Project6: impl this protocol with actual network communication
⋆Project7: Try to Implement this scheme
⋆Project8: AES impl with ARM instruction
⋆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
⋆Project21: Schnorr Bacth
⋆Project22: research report on MPT

环境配置

硬件环境：

处理器：11th Gen Intel(R) Core(TM) i5-1135G7

内存：16.0 GB (15.8 GB 可用)

软件环境：

操作系统：win11

Python 编译器：IDEL 3.7（64bit）

C++编译器：visual studio 2019

Project1: implement the naïve birthday attack of reduced SM3

基于生日问题:

假设有 m (m<365)个人，他们的生日在一年（365 天）中均匀分布，为保证这 m 个人中，至少有两人生日相同的概率大于 1/2，m 应取多大?

答案：m=23, 考虑 m 个人生日都不同的概率

$$\overline{p(m)} = 1\times(1-\frac{1}{365})\times(1-\frac{2}{365})\times\ldots\times(1-\frac{m-1}{365}) = \prod_{i=1}^{m-1}(1-\frac{i}{365})$$

$$\because e^{-x} \approx 1-x,$$

$$\therefore \overline{p(m)} \approx 1\times e^{-\frac{1}{365}}\times e^{-\frac{2}{365}}\times\ldots\times e^{-\frac{m-1}{365}} = e^{-\frac{m(m-1)/2}{365}}$$

$$p(m) = 1-\overline{p(m)} \approx 1-e^{-\frac{m(m-1)/2}{365}} \approx 1-e^{-\frac{m^2}{2\times365}} = 1/2$$

$$m \approx \sqrt{2\ln 2\times 365} \approx 1.17\sqrt{365}$$

推广：

$$p(m) \approx 1-e^{-\frac{m(m-1)/2}{N}} \approx 1-e^{-\frac{m^2}{2\times N}}$$

$$m \approx \sqrt{2\ln\frac{1}{1-p(m)}\times N}$$

当$p(m)=0.5$时, $m \approx 1.17\sqrt{N}$

得到定理: 设杂凑函数 h 的输出值长 n 比特, 则经过约 2^n/2 次杂凑运算, 找到一对碰撞(x, x')的概率大于 1/2，输出值长 n 比特→N=2^n。

实际上, 杂凑函数的理想安全强度(各安全属性的上界 $h(\cdot):\{0,1\}^*\rightarrow\{0,1\}^n$)找到一对碰撞的复杂度为 2^n/2, 若能够以小于 2^n/2 的复杂度找到一对碰撞, 则认为 h 被破解。

另外需要注意的是：由于生日攻击只能保证有 50%的概率找到一对碰撞，所以只执行一次攻击可能并不能找到碰撞。运行代码也有失败的可能性。

代码分析：

使用 secrets 模块生成一个随机的 16 字节长的十六进制字符串，并将其赋值给变量 cip_text。然后，使用 sm3 算法对该随机字符串进行哈希计算，并将结果赋值给变量 cip_hash。t_len 是指定的攻击长度，用于执行生日攻击。

```
def birthday_attack(t_len):
    num = 2 ** (t_len // 2)
    ans = [-1] * (2 ** t_len)
    # 循环遍历，对于每一位
    for i in range(num):
        temp = cip_hash[:t_len // 4]
        if ans[int(temp, 16)] == -1:
            ans[int(temp, 16)] = i
        else:
            return temp

```

birthday_attack() 是执行生日攻击的函数。根据给定的攻击长度 t_len，计算出攻击的次数 num。我们创建一个长度为 2 ** t_len 的列表 ans，并将其初始化为全 -1。

然后，使用一个循环来进行攻击。在每次迭代中，从哈希值 cip_hash 中取出前 t_len // 4 位，并将其赋值给变量 temp。然后，我们将 temp 解析为整数，并使用它作为 ans 列表的索引。

如果 ans[int(temp, 16)] 的值为 -1，表示这是第一次遇到这个索引，我们将当前迭代的次数 i 存储在 ans[int(temp, 16)] 中。否则，说明这是第二次遇到相同的索引，我们找到了碰撞结果，返回 temp。

```
if __name__ == '__main__':
    start = time.perf_counter()
    res = birthday_attack(t_len)
    end = time.perf_counter()
    print("{}位碰撞为{}".format(t_len, res))
    print("运行时间为", end - start)

```

在主程序中，使用 time.perf_counter() 记录开始时间和结束时间，然后调用 birthday_attack() 函数执行生日攻击并返回结果。最后，打印攻击长度和碰撞结果，并计算和打印运行时间。

运行截图：

```
8位碰撞为64
运行时间为 2.1199999999943486e-05
>>>
============== RESTART: C:/Users/14917/Desktop/birthday
16位碰撞为8a33
运行时间为 0.0016499999999999293
>>>
============== RESTART: C:/Users/14917/Desktop/birthday
24位碰撞为c2dd52
运行时间为 0.24166049999999994
>>>
```

Project2: implement the Rho method of reduced SM3

Rho 攻击是一种用于寻找哈希函数碰撞的方法，基于生日攻击的原理。它通常用于对散列函数的安全性进行评估或对特定哈希函数的优化进行研究。Rho 攻击的基本思路是通过构造两个消息序列，并分别计算它们的哈希值，然后比较这些哈希值，如果发现相同的哈希值，那么就找到了一个碰撞。

碰撞攻击是一种破解哈希函数安全性的手段，但是在实际应用中，通常使用的哈希函数具有足够高的安全性。因此，对于正常的哈希函数应用，碰撞攻击并不构成实际威胁。

代码分析：

```
1.    def SM3_rho_attack(n):
 2.        # 生成随机消息
 3.        random_number = random.randint(0, 2**(n+1)-1)
 4.        cip = hex(random_number)[2:]
 5.        # 计算初始哈希值
 6.        cip_hash1 = sm3.sm3_hash(func.bytes_to_list(bytes(cip, encoding='utf-8')))
 7.        cip_hash2 =
sm3.sm3_hash(func.bytes_to_list(bytes(sm3.sm3_hash(func.bytes_to_list(bytes(cip,
encoding='utf-8'))), encoding='utf-8')))
 8.        cnt = 1
 9.        # 找到消息碰撞
```

```
10.        while cip_hash1[:int(n/4)] != cip_hash2[:int(n/4)]:
11.            cnt += 1
12.            cip_hash1 = sm3.sm3_hash(func.bytes_to_list(bytes(cip_hash1,
encoding='utf-8')))
13.            cip_hash2 =
sm3.sm3_hash(func.bytes_to_list(bytes(sm3.sm3_hash(func.bytes_to_list(bytes(cip_hash2,
encoding='utf-8'))), encoding='utf-8')))
14.        # 执行 rho 攻击
15.        for j in range(1, cnt+1):
16.            cip_hash1 = sm3.sm3_hash(func.bytes_to_list(bytes(cip_hash1,
encoding='utf-8')))
17.            cip_hash2 = sm3.sm3_hash(func.bytes_to_list(bytes(cip_hash2,
encoding='utf-8')))
18.            if cip_hash1[:int(n/4)] == cip_hash2[:int(n/4)]:
19.                collision = sm3.sm3_hash(func.bytes_to_list(bytes(cip_hash1,
encoding='utf-8')))
20.                return [cip_hash1, cip_hash2, collision]
21.        # 未找到碰撞返回 None
22.        return None
23.
```

这个函数接受一个参数 `n`，表示攻击的比特数。它首先生成一个随机数作为初始消息，并计算出两个初始的哈希值。然后使用 rho 方法查找碰撞，直到前 n/4 比特相等。

在找到碰撞后，使用 rho 攻击方法继续计算，找到消息的完全碰撞，并返回包含三个结果（原始消息的哈希值、衍生消息的哈希值和消息碰撞）的列表。

如果无法找到碰撞，返回 None。

```
1.    if __name__ == '__main__':
2.        n = int(input("攻击多少 bit: \n"))
3.        start = time.time()
4.        res = SM3_rho_attack(n)
5.        end = time.time()
6.
7.        if res is not None:
8.            print("message1:", res[0])
9.            print("message2:", res[1])
```

```
            print("碰撞:", res[2])
        else:
            print("未找到碰撞")

        print(end-start,"seconds\n")

```

这部分代码从用户获取攻击的比特数 n，然后调用 `SM3_rho_attack()` 函数进行攻击。根据函数的返回值，输出结果或提示未找到碰撞，并计算程序执行的时间。

通过 python 实现了一个 SM3 哈希函数的 rho 攻击，使用随机消息生成初始哈希值并通过 rho 方法查找碰撞。如果成功找到碰撞，返回碰撞结果；否则，返回 None。主程序负责接收参数、调用函数和输出结果。

运行各种 bit 的攻击可发现 8bit 及以下速度都较快，但是 16bit 就运行很慢，目前在可接受的时间内暂时只能找到 16 位的碰撞。

```
========== RESTART: C:\Users\14917\Desktop\rho.py ==========
攻击多少bit:
8
message1: 9192524f55ee3ce086aebb840dfb0ff9c7c3f70a3f5b91b3632ca9fa36a44037
message2: 911fabceebd0459444f2926dd0dece9ab2a0310baf3f2c49cab5eb0e5302559a
碰撞: 8a7b34977c1bd7202104be46e787db91c24103c17ea96e7780db035416fb1517
2.6775033473968506 seconds

>>>
========== RESTART: C:\Users\14917\Desktop\rho.py ==========
攻击多少bit:
16

========== RESTART: C:\Users\14917\Desktop\rho.py ==========
攻击多少bit:
4
未找到碰撞
0.013363838195800781 seconds

>>>
========== RESTART: C:\Users\14917\Desktop\rho.py ==========
攻击多少bit:
4
未找到碰撞
0.0325167179107666 seconds

>>>
========== RESTART: C:\Users\14917\Desktop\rho.py ==========
攻击多少bit:
10
message1: adb7c2c59ecd68e597b358cf71b045f6fdce5fe8a62f6a4f9ec6542016383816
message2: ad7038b514238abf63e407918afadc906536472b31e1144f16ae86c91727bf46
碰撞: dc0bc1a8054c89ef2b215130e5f4ec41d9711885c94aa3eca38c87c37c1d765d
7.16015625 seconds
```

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

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(例如 512 或 1024 比特）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)。

满足上述要求的散列函数称为 Merkle–Damgård 散列函数（Merkle–Damgård hash function)

SHA256 长度扩展攻击是一种针对 SHA256 哈希算法的攻击方法。在正常情况下，SHA256 算法是一种单向的、不可逆的哈希算法，即无法从哈希值推导出原始输入数据。然而，长度扩展攻击利用了 SHA256 算法的特性，通过在已知哈希值后面追加额外的数据来构造新的哈希值。

具体来说，长度扩展攻击利用了 SHA256 算法中的消息扩展性漏洞。该漏洞使得攻击者可以在已知哈希值和原始输入数据的情况下，计算出附加数据的哈希值，而无需知道附加数据的具体内容。

为了进行长度扩展攻击，攻击者需要知道原始输入数据的长度、已知的哈希值以及附加数据。通过使用 SHA256 算法的特性，攻击者可以构造新的哈希值，而无需知道原始输入数据的内容。

为了防止长度扩展攻击，一种常见的做法是使用 HMAC（Hash-based Message Authentication Code）来增加数据的完整性验证。

代码分析：

1. length_extend_attack(src_hash, append_msg) 函数接收两个参数：原始哈希值 src_hash 和要附加的消息 append_msg。它使用原始哈希值来构造一个假的消息，并计算出相应的伪造哈希值。
2. 首先，将原始哈希值 src_hash 拆分成 8 字节的整数列表 vec。
3. 创建一个空的字节数组 fake_msg，作为待伪造的消息。
4. fake_msg 的构造方式如下：

   添加单个字节 0x80，表示消息的结尾。

   添加零字节，使得 (len(fake_msg) + 55) % 64 等于零，以满足长度扩展攻击的要求。

   添加一个 64 位的大端序表示的整数，表示新消息的总比特数。

   添加附加消息 append_msg 的字节表示。
5. 使用 hashlib.new('sm3') 创建一个 SM3 哈希对象 fake_hash，并通过调用 fake_hash.update(fake_msg) 更新哈希对象的状态。
6. 最后，通过调用 fake_hash.digest() 得到伪造哈希值的字节表示。
7. 将伪造哈希值与 vec[0] 和 vec[1] 的整数表示结合在一起，并使用 SHA-256 计算出最终的伪造哈希值。
8. 在 main() 函数中，随机生成一个原始消息并计算其哈希值 message_hash。
9. 定义附加消息 append_m。
10. 调用 length_extend_attack() 函数，在已知的原始哈希值 message_hash 和附加消息 append_m 的情况下，得到伪造哈希值 fake_hash。
11. 创建新的消息 new_msg，将原始消息和附加消息连接在一起。

12. 通过计算新消息的哈希值 new_hash。

13. 打印原始消息、原始哈希值、附加消息、伪造哈希值和新哈希值。

14. 检查伪造哈希值是否与新哈希值相等，以判断攻击是否成功。

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

在使用 SM3 进行基础加密的基础上，通过消除不必要的函数调用与内存调用，使用循环展开，SIMD 等方式对项目进行软件加速，实现目的。

对于消除不必要的函数调用与内存调用，除不必要的工作（函数调用、条件测试、内存引用）和利用处理 器提供的指令级并行。大多数编译器已经实现了通过优化级别的选择来实现不同程度的 性能提升，但是对于内存别名使用、修改全局程序状态的函数调用等操作，编译器很难 做出进一步的优化。没有任何编译器能用一个好的算法或数据结构代替低效率的算法或 数据结构，因此，编写使编译器能高效的产生代码的程序极具意义。

对于循环展开，它可以从两方面提高程序的性能，分别是减少无助于程序结果的操作的数量（循环索 引计算、条件分支等），提供一些进一步变化代码的方法、减少计算中关键路径（提供 程序所需周期数的下界）上的操作数量。

对于 SIMD 指令集，一个寄存器可以储存多个过程变量，从而实现加速。

Project5: Impl Merkle Tree following RFC6962

Merkle Tree（默克尔树），也称为哈希树，是一种用于验证大型数据集完整性的数据结构。它基于哈希函数和二叉树的概念。

Merkle Tree 的构建过程如下：

1. 将数据集划分为固定大小的数据块（通常是 2 的幂次方），每个数据块称为叶子节点。
2. 对每个叶子节点应用哈希函数，得到每个叶子节点的哈希值。
3. 如果叶子节点的数量为奇数，则复制最后一个叶子节点使其成为偶数。
4. 将这些哈希值两两配对，将每一对的哈希值连接在一起，再次应用哈希函数，得到它们的父节点的哈希值。
5. 重复第 4 步，直到只剩下一个根节点，这个根节点就是默克尔树的根。

Merkle Tree 具有以下主要特点：

完整性验证：通过比较根节点的哈希值，可以验证整个数据集是否完整，即

是否没有发生任何更改。

效率：只需要比较根节点的哈希值就可以验证整个数据集，而不需要比较所有数据。

安全性：即使数据集非常大，保存和验证 Merkle Tree 的哈希值仍然是相对较小的，因此可以提供高效的数据完整性验证。

代码分析：

这段代码实现了一个基于 SHA-256 的 Merkle 树计算。Merkle 树是一种哈希树的数据结构，常用于验证大量数据的完整性和一致性。

代码首先定义了一个 sha256 函数，用于计算给定字符串的 SHA-256 哈希值。它使用 OpenSSL 库中的 SHA256 函数来实现。然后，computeMerkleRoot 函数接受一个包含交易数据的字符串向量，通过迭代地将相邻的哈希值合并为新的哈希值，最终得到 Merkle 树的根哈希值。

在 main 函数中，创建了一个包含四个交易数据的向量，并调用 computeMerkleRoot 函数计算 Merkle 树的根哈希值。然后，计算程序的运行时间，并将结果输出到控制台。

运行它需要下载一个 OpenSSL 库

```
Merkle Root: 20f780f6f1d59ef824b5f84b5b05d4e8478c7437de648df3a7af3a7c64d96972
运行时间: 1.23 毫秒
```

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

以太坊使用椭圆曲线数字签名算法（Elliptic Curve Digital Signature Algorithm，简称 ECDSA）来实现交易的签名和验证。ECDSA 是一种非对称加密算法，它基于椭圆曲线算法在有限域上的运算。

ECDSA 的签名过程包括以下几个步骤：

生成密钥对：首先，需要生成一个私钥和对应的公钥。私钥是一个随机数，而公钥是通过椭圆曲线算法计算得到的。以太坊使用的椭圆曲线参数为 secp256k1。该曲线的方程为 y^2 = x^3 + 7。该曲线的参数已在以太坊协议中定义。

计算消息摘要：将待签名的原始消息通过哈希函数（通常使用 SHA-256）计算出一个固定长度的消息摘要。

签名：在 ECDSA 中，每个用户都有一对密钥：私钥和公钥。私钥由一个随机数生成，并且必须保密，而公钥则可以公开。公钥是通过将私钥与生成曲线上的一个点进行乘法运算得到的。通过私钥和公钥的对应关系，可以验证签名和推导出公钥。

当用户要对一段数据进行签名时，首先需要将数据通过哈希函数进行处理，得到一个固定长度的哈希值。然后，使用私钥对哈希值进行加密，生成签名。签名的生成涉及到一系列的数学运算，具体过程是选择一个随机数作为签名时的临时私钥，然后使用临时私钥对消息摘要进行加密，得到两个整数（r，s）。这两个整数组成了数字签名。

验证签名：对于给定的消息、公钥和数字签名，可以使用 ECDSA 算法进行验证。验证过程涉及使用公钥重建临时私钥，然后使用该临时私钥和消息摘要来验证签名的有效性。验证签名的过程包括将公钥转换为曲线上的点、对签名进行反解析和运算，最终得到一个坐标值，并与原始数据进行比较。

以太坊中使用的 ECDSA 公钥推导方法如下：

从交易信息中获取发送方的公钥哈希（public key hash）。

将公钥哈希作为输入，通过 Keccak-256 哈希函数进行计算，得到一个 256 位的哈希值。

取哈希值的后 20 个字节（40 个十六进制字符），再将其前面添加上“0x”前缀，得到一个 40 个字符的十六进制字符串。这个字符串就是以太坊中与交易地址对应的公钥推导方法。

总结起来，以太坊中使用的 ECDSA 签名和公钥推导过程如下：

首先，生成密钥对；然后，对原始消息进行哈希计算，得到消息摘要；接着，使用私钥对摘要进行加密，生成数字签名；最后，通过公钥推导方法，可以从交易地址中获取相应的公钥。ECDSA 签名和公钥推导在以太坊中起到了重要的作用，保证了交易的可信度和安全性。通过私钥对数据进行签名，可以确保交易的真实性和完整性。而通过公钥的推导和验证过程，可以验证签名的有效性。它区块链的安全性提供了坚实的基础。

代码分析：

使用了 fastecdsa 库来实现 secp256k1 曲线上的 ECDSA 签名和验证。

导入所需的库：

```
1. from fastecdsa.curve import secp256k1
2. from fastecdsa.encoding.sec1 import PEMEncoder, PEMParser
3. from fastecdsa.point import Point
4. from hashlib import sha256
5. import time
6.
```

生成随机私钥：

```
1. def generatePrivateKey():
2.     return secrets.randbelow(secp256k1.q)
3.
```

generatePrivateKey 函数使用 secrets.randbelow 方法生成一个小于 secp256k1.q 的随机数作为私钥，并返回该私钥。

ECDSA 签署消息：

```
1. def signECDSAsecp256k1(msg, privKey):
2.     msgHash = sha256(msg.encode("utf8")).digest()
3.     signature = secp256k1.sign(msgHash, privKey)
4.     return signature
5.
```

signECDSAsecp256k1 函数接受一个消息和一个私钥作为输入，首先对消息进行 SHA-256 哈希，然后使用 secp256k1 曲线和私钥对哈希结果进行签名，返回签名结果。

ECDSA 验证签名：

```
1. def verifyECDSAsecp256k1(msg, signature, pubKey):
2.     msgHash = sha256(msg.encode("utf8")).digest()
3.     valid = secp256k1.verify(msgHash, signature, pubKey)
4.     return valid
5.
```

verifyECDSAsecp256k1 函数接受一个消息、一个签名和一个公钥作为输入，首先对消息进行 SHA-256 哈希，然后使用 secp256k1 曲线、签名和公钥进行验证，返回验证结果(boolean)。

测试签名和验证的时间性能：

```
def testPerformance():
    msg = "Message for testing performance"
    privKey = generatePrivateKey()

    # 计算签名时间
    start = time.time()
    signature = signECDSAsecp256k1(msg, privKey)
    end = time.time()
    signingTime = end - start

    # 计算验证时间
    pubKey = secp256k1.G * privKey
    start = time.time()
    valid = verifyECDSAsecp256k1(msg, signature, pubKey)
    end = time.time()
    verifyingTime = end - start

    print("\n 签名时间:", signingTime, "秒")
    print("验证时间:", verifyingTime, "秒")

```

testPerformance 函数用于测试签名和验证的时间性能。它生成一个测试消息和一个随机私钥，然后计算签名和验证这两个操作的时间，并输出结果。

```
def exampleCode():
    # ECDSA 签署消息
    msg = "Message for ECDSA signing"
    privKey = generatePrivateKey()
    signature = signECDSAsecp256k1(msg, privKey)
    print("消息:", msg)
    print("私钥:", hex(privKey))
    print("签名: r=" + hex(signature.r) + ", s=" + hex(signature.s))

    # ECDSA 验证签名
    pubKey = secp256k1.G * privKey
    valid = verifyECDSAsecp256k1(msg, signature, pubKey)
    print("\n 消息:", msg)
    print("公钥:", pubKey)
    print("签名是否有效?", valid)

    # ECDSA 验证篡改签名
```

```
    msg = "Tampered message"
    valid = verifyECDSAsecp256k1(msg, signature, pubKey)
    print("\n 消息:", msg)
    print("签名是否有效(篡改后)?", valid)

    # 运行性能测试
    testPerformance()

# 运行示例代码
exampleCode()
```

exampleCode 函数演示了如何使用上述函数进行 ECDSA 签名和验证。它生成一个消息、一个随机私钥，并对消息进行签名。然后使用相应的公钥对签名进行验证。还展示了如何在篡改消息的情况下验证签名的有效性，并进行了性能测试以比较签名和验证的时间。最后，它运行了性能测试函数 testPerformance。

```
消息: Message for ECDSA signing
私钥: 0x4e2ee795a2c5a7b81f4fb0830c172e63cc40664f4a9c518db6889c8a711d8a5e
签名: r=0x332c12c910e53e3e1bb35659bc65a8f23fb12b7e6e7b3b810a3e245d3ec8e8f5, s=
0x5e6554f3a02e170649221526f3d084ba3892696c398979576dd452597f22f421

消息: Message for ECDSA signing
公钥: (223003724436385882411312675367565893225525552098052537378046031222712520936 72,
29686512649891528211311612316873134339361814136318609058345054774594180066305)
签名是否有效? True

消息: Tampered message
签名是否有效(篡改后)? False

签名时间: 0.24492740631103516   秒
验证时间: 0.0009999275207519531 秒
```

Project11: impl sm2 with RFC6979

RFC 6979 是一项由 IETF（Internet Engineering Task Force）制定的标准，它定义了一种确定性签名算法，用于生成安全的 ECDSA（Elliptic Curve Digital Signature Algorithm）数字签名。

在传统的 ECDSA 签名算法中，随机数是用于生成签名的重要组成部分。然而，如果使用不安全的伪随机数生成器或者重用相同的随机数，可能会导致私钥泄露或签名不安全。RFC 6979 的目的就是解决这个问题，通过使用确定性算法来生成签名所需的随机数，从而提供更可靠和安全的签名方法。

RFC 6979 中定义的确定性签名算法使用了 HMAC（Hash-based Message Authentication Code）函数和伪随机函数，结合椭圆曲线密码学的原理，以确保生成的随机数具有适当的难以预测性和安全性。该算法基于输入的消息和私钥，通过计算散列值等步骤来生成签名所需的随机数。

通过使用 RFC 6979 确定性签名算法，可以消除依赖不安全的伪随机数生成器的风险，并提供更稳定和可复现的签名过程。这有助于确保数字签名的安全性和可验证性，提高了加密系统的整体安全性。

代码分析：

generate_key 函数用于生成私钥和公钥。首先创建了一个 CryptSM2 对象用于操作 SM2 算法，然后调用 generate_key 方法生成密钥对，最后通过 load 方法加载私钥。函数返回生成的私钥对象和对应的公钥。

sign_message 函数用于对消息进行签名。首先计算消息的哈希值，这里使用 SM3 算法计算消息的散列值。然后使用私钥对象的 sign 方法对哈希值进行签名，得到签名结果。函数返回生成的签名。

示例部分首先调用 generate_key 函数生成密钥对，得到私钥和公钥对象。然后定义要签名的消息，这里用学号"202100460144"。接着调用 sign_message 函数对消息进行签名，使用私钥对象和消息作为输入参数，得到签名结果。最后打

印生成的私钥、公钥和签名。

Project14: Implement a PGP scheme with SM2

PGP（Pretty Good Privacy）是一种常用的加密和认证协议。它被广泛应用于保护电子邮件和文件的隐私和安全。

PGP 使用了一系列的加密算法和数字签名技术来实现加密、解密和认证功能。主要特点包括:

1. 非对称加密：PGP 使用非对称加密算法（如 RSA）来生成密钥对，包括公钥和私钥。公钥用于加密消息，私钥用于解密消息。
2. 对称加密：PGP 使用对称加密算法（如 AES）来加密消息的内容。对称加密速度更快，因此 PGP 在加密过程中使用对称加密算法来加密实际的消息数据。
3. 数字签名：PGP 使用数字签名技术来验证消息的真实性和完整性。发送方使用自己的私钥对消息进行签名，接收方使用发送方的公钥来验证签名。
4. 密钥管理：PGP 提供了密钥管理功能，可以生成密钥对、导入和导出密钥，以及管理密钥环。
5. Web of Trust：PGP 支持 Web of Trust 模型，其中用户可以互相认证彼此的公钥，从而建立信任网络。

通过使用 PGP，用户可以加密他们的数据，确保只有授权的人能够解密和阅读。同时，PGP 还允许用户对消息进行数字签名，以确保消息的来源和完整性

代码分析：

导入 gmssl 库的 sm2 和 sm4 模块，以及其他所需的库。定义了一个字符串变量 str36，表示 36 进制的字符串。定义了一个 16 字节的 iv 向量，用于 SM4 的 CBC 模式加密。定义了一个私钥(private_key)和公钥(public_key)，定义了一个生成随机 k 的函数 generate_random_k()，用于生成 16 字节的随机密钥。定义了一个发送者函数 sender(msg)，接受一个消息作为参数。

在 sender 函数中，首先生成一个随机的 k，然后使用 SM4 对消息进行加密。创建一个 SM2 加密器 sm2_enc，传入公钥和空字符串作为私钥。使用 sm2_enc 对 k 进行加密得到 enc_key。返回加密后的消息 enc_msg 和加密后的密钥 enc_key。定义了一个接收者函数 receiver(enc_msg, enc_key)，接收加密后的消息和加密后的密钥作为参数。

在 receiver 函数中，首先创建一个 SM2 解密器 sm2_enc，传入私钥和公钥。使用 sm2_enc 对 enc_key 进行解密得到 k。创建一个 SM4 对称加密器 crypt_sm4，使用 k 作为密钥，模式设置为 SM4_DECRYPT。使用 crypt_sm4 对 enc_msg 进行解密得到 decrypt_m。返回解密后的消息 decrypt_m。

输入需要加密的数据，并执行 sender 和 receiver 函数进行加密和解密。输出加密后的消息 enc_msg 和加密后的密钥 enc_key。判断解密后的消息 dec_m 是否与输入的消息 msg 相等，如果相等则输出解密成功，否则输出解密失败。

```
输入需要加密的数据:
Hello, World!
加密后的消息为:
b'BNUXdHvC8CX/KDlkEa2WKT9pdM1Mgh4DKjAiovHtqLBln0wHfsxHwqsg+Cjf9RdTwKLD2
Go8l7EV6/Hg0iHudQ=='
加密后的会话秘钥为:
b'ljltXgfw96nlzpxnsDaKKdDj4Y5bDTJJKv76gMxK8D9qHgK2UR4hy3oxXPmB4nO7'
解密成功!
0.14926528930664062 秒
```

Project17：比较 Firefox 和谷歌的记住密码插件的实现区别

Firefox 和谷歌的记住密码插件在实现上区别如下：

首先，Firefox 的记住密码功能是内置在浏览器中的，用户可以自动保存和填充密码。用户第一次登录时，Firefox 会询问是否保存密码，如果用户同意，下次登录时它会自动填充密码。这一功能对于大多数网站都有效。

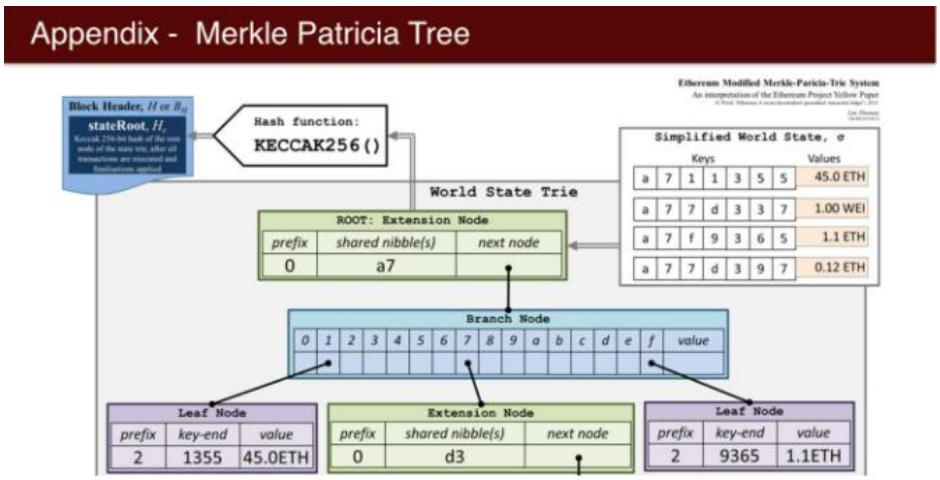
谷歌的 Chrome 浏览器也内置了记住密码的功能，类似于 Firefox，用户登录时可以选择保存密码，下次登录时自动填充密码。除此之外，Chrome 还有一个独特的功能，即可以通过 Google 账户自动保存和同步密码。这意味着，用户可以在不同设备上使用 Chrome 浏览器，并通过 Google 账户同步保存的密码，方便在任何设备上都能自动填充密码。

另外，Firefox 还有一些额外的密码管理插件，例如 LastPass 和 Bitwarden，它们提供更高级的密码管理功能，如生成强密码、自动填充表单等。

总结得到，Firefox 和谷歌的记住密码插件在基本的功能上相似，但谷歌的 Chrome 浏览器通过 Google 账户同步密码的功能更加方便，而 Firefox 则提供更多的第三方密码管理插件选择。

Project22: research report on MPT

Merkle Patricia Tree（MPT）是一种基于前缀树和默克尔树的数据结构，常用于以太坊区块链中存储和检索账户和智能合约的状态。MPT 提供了高效的键值对存储，并通过哈希函数保证数据的完整性和不可篡改性。



一、基本原理和应用场景。

Merkle Patricia Tree（简称 MPT）是一种基于前缀树和默克尔树的数据结构，广泛应用于区块链系统和分布式数据库中。它通过有效地组织和存储键值对数据，并借助哈希函数保证数据的完整性和安全性。

MPT 的基本原理如下：

前缀树结构：MPT 使用前缀树的思想来组织数据。每个节点包含一个路径片段的前缀和与之关联的子节点或值。通过从根节点开始，根据键的每个字符依次向下遍历，就可以定位到对应的值。

默克尔树哈希：MPT 使用哈希函数来计算每个节点的哈希值。每个节点的哈希值由其子节点的哈希值通过哈希函数计算得到。这样，通过节点的哈希值，可以高效地验证树的完整性和数据的不可篡改性。

路径片段编码：MPT 对键进行编码，将键的每个字符转换为路径片段的前缀。这样，在树中按路径片段进行查找时，可以非常高效地定位到对应的值，避免了无谓的遍历。

MPT 在区块链和分布式数据库中具有广泛的应用场景，包括但不限于以下几个方面：

状态存储：MPT 被广泛用于区块链系统中的状态存储。在以太坊等智能合约平台中，每个账户的状态信息（例如余额、代码、存储等）都通过 MPT 来组织和保存，便于高效地检索和更新。

事件日志：MPT 可以用于记录和验证区块链中的事件日志。每个事件可以通过 MPT 的路径片段来进行索引，确保事件的完整性和顺序性。

分布式数据库：MPT 可以作为一种高效的数据结构用于分布式数据库中。它可以提供快速的键值对存储和检索，同时提供数据的完整性保证。

资源证明：利用 Merkle Patricia Tree 的哈希特性，可以实现有效的资源证明（Proof of Custody），用于验证特定数据是否存在或是否被篡改。

可见 Merkle Patricia Tree 是一种高效的数据结构，通过前缀树与默克尔树相结合的方式，实现了快速的键值对存储与检索，并且通过哈希函数保证了数据的完整性和安全性。它在区块链和分布式数据库等领域有着广泛的应用场景。

二、不同方法的优缺点

对于 Merkle Patricia Tree 的研究和应用工作已经积累了一定的成果，以下是对不同方法的优缺点以及现有研究的不足之处的回顾和分析：

方法一：基于递归的实现方法

优点：简单直观，易于理解和实现。

缺点：在处理大规模数据时，递归的内存消耗较大，容易导致性能问题。

方法二：基于迭代的实现方法

优点：相比于递归方法，迭代方法具有更低的内存消耗，适用于处理大规模数据。

缺点：实现相对复杂，需要处理不同节点类型的特殊情况。

方法三：基于压缩编码的实现方法

优点：通过利用路径片段的重复性，可以对 Merkle Patricia Tree 进行压缩，减少存储空间的占用。

缺点：在更新和删除操作时，需要进行解压缩和重新压缩，增加了计算开销。

方法四：基于并行计算的实现方法

优点：通过并行计算，可以提高 Merkle Patricia Tree 的构建和检索速度。

缺点：并行计算的实现较为复杂，需要考虑并发访问的冲突和同步机制。

现有研究的不足之处包括：

性能优化方面：当前研究主要集中在 Merkle Patricia Tree 的基本实现和应用，对于大规模数据的处理和性能优化仍有待进一步研究。例如，如何提高插入和删除操作的效率，如何降低存储空间的占用等。

安全性分析方面：虽然 Merkle Patricia Tree 具有哈希函数保证数据完整性的特点，但对于碰撞攻击和前图攻击等安全风险的分析还相对较少。研究人员可以进一步探索和分析这些潜在的安全问题。

应用场景拓展：当前大部分研究集中在区块链和分布式数据库等特定领域的应用，而其他领域的应用潜力还未充分挖掘。可以进一步考虑 Merkle Patricia Tree

在文件系统、物联网等领域的应用场景。

综上所述，虽然 Merkle Patricia Tree 的研究和应用已取得许多成果，但仍存在性能优化、安全性分析和应用场景拓展等方面的不足之处。未来的研究可以继续在这些方面进行探索和改进，以进一步提升 Merkle Patricia Tree 的效率和扩展性。

三、原理和设计思路

Merkle Patricia Tree（简称 MPT）是一种基于前缀树的数据结构，用于高效地存储和检索键值对。其设计思路主要包括节点类型、路径片段的编码方式和键值对的存储结构。

节点类型：

扩展节点（Extension Node）：用于表示共享路径片段，包含一个字符片段（路径的一部分）和一个指向子节点的引用。

叶子节点（Leaf Node）：用于表示键值对，包含一个完整的路径片段和对应的值。

分支节点（Branch Node）：用于表示分支路径上的节点，包含 16 个子节点的引用（0 到 15，代表 16 进制字符）和一个附加的值（可选）。

路径片段的编码方式：

路径片段采用 RLP（Recursive Length Prefix）编码进行压缩，以减少存储空间的占用。RLP 编码将字符串或数组按照特定规则进行编码，同时保留了原始数据的结构和顺序。

在 MPT 中，路径片段使用 RLP 编码后作为叶子节点的一部分进行存储。

键值对的存储结构:

MPT 将键值对存储在叶子节点中，键和值都使用 RLP 编码后进行存储。

对于重复的路径片段，可以通过扩展节点实现共享存储，这样可以有效减少存储空间的使用。

Merkle Patricia Tree 的工作原理如下:

MPT 的根节点是一个特殊的分支节点，用于存储完整路径上的第一个字符对应的子节点的引用。

当插入键值对时，MPT 会根据键找到对应的叶子节点，并将值存储在叶子节点中。如果遇到路径片段重复的情况，则会创建扩展节点，共享相同的路径片段。

当需要检索键对应的值时，MPT 会按照键的字符顺序从根节点开始，依次查找对应的节点，直到找到叶子节点，并返回对应的值。

在删除键值对时，MPT 会将对应的叶子节点标记为删除状态，而不是直接删除节点和路径片段。这样做可以保持 MPT 的完整性，并且在需要进行状态还原时可以更加高效。

通过节点类型的设计、路径片段的编码方式和键值对的存储结构，Merkle Patricia Tree 实现了高效的存储和检索功能，并且具有较低的存储空间需求。它被广泛应用于区块链和分布式数据库等领域，提供了可靠和高效的数据存储解决方案。

四、数据结构和算法

Merkle Patricia Tree（MPT）使用了前缀树的数据结构，并结合了哈希函数和一些特定的算法来实现高效的存储和检索功能。下面将详细介绍 MPT 使用的

数据结构和算法：

前缀树的实现方式：

MPT 采用基于前缀树的数据结构，其中每个节点可以是扩展节点、叶子节点或分支节点。扩展节点和叶子节点表示路径片段，而分支节点表示路径上的分支。

哈希函数的选择和应用：

MPT 使用哈希函数来计算节点的哈希值，以保证数据完整性和唯一性。常用的哈希函数包括 Keccak-256、SHA-256 等。节点的哈希值由节点类型和相应内容的哈希结果组成。

节点的插入操作：

当需要插入一个键值对时，先将键和值进行 RLP 编码，并生成对应的叶子节点。

从根节点开始，按照键的字符顺序依次查找对应的节点，在遇到空节点或分支节点时创建新的节点。

如果路径片段重复，则使用扩展节点来共享路径片段。

在更新节点时，如果旧节点已经存在，则可能需要对其进行更新或删除操作。

节点的删除操作：

删除操作标记叶子节点为删除状态，而不是直接删除节点和路径片段。这样可以保持 MPT 的完整性，并且在需要进行状态还原时可以更加高效。

如果节点没有任何子节点并且被标记为删除状态，则可以将该节点及其路径片段从树中移除。该过程称为清理（pruning）操作。

路径压缩：

MPT 使用路径压缩来减少存储空间的占用。路径压缩通过合并共享路径片

段，将扩展节点和叶子节点紧密编码在一起，减少了存储重复路径的需要。

Merkle Patricia Tree 的设计基于前缀树，通过哈希函数确保数据的完整性，并采用一些特定的算法进行节点的插入和删除操作。这些设计和算法使得 MPT 能够提供高效的存储和检索功能，并被广泛应用于区块链、分布式数据库等领域。

五、安全性方面

Merkle Patricia Tree（MPT）在安全性方面具有以下特点和保障：

节点哈希的完整性验证：

MPT 使用哈希函数计算节点的哈希值，并将其用作节点的唯一标识。通过节点的哈希值，可以验证节点的完整性。

在检索数据时，可以通过比较节点的哈希值与存储的哈希值是否匹配来验证节点是否被篡改。

抗碰撞能力：

哈希函数在理论上应该具备抗碰撞（collision resistance）的能力，即使输入数据非常庞大，也不太可能出现相同的哈希值。

MPT 使用哈希函数计算节点的哈希值，抗碰撞能力保证了节点的唯一性。

数据不可篡改性：

MPT 中的哈希值是基于节点内容计算得出的，只要节点内容发生改变，其哈希值也会发生变化。

因此，如果有人试图修改 MPT 中的任何节点，都会导致哈希值不匹配，从而暴露出数据被篡改的情况。

总体而言，Merkle Patricia Tree 通过使用哈希函数、节点哈希的完整性验证、

抗碰撞能力以及数据不可篡改性等特点，提供了一定的安全性保障。这些特点确保了数据的完整性和可靠性，并提高了对数据篡改的检测能力，使 MPT 适用于需要保证数据安全性的应用场景，如区块链和分布式数据库等。然而，实现 MPT 的安全性还要考虑其他因素，如哈希函数的选择和算法的正确性，以及防止外部攻击等。因此，在实际使用中，仍需综合考虑安全性措施。

六、实际应用中的使用案例

Merkle Patricia Tree（MPT）在以太坊区块链中的实际应用非常广泛，主要用于状态存储和账户验证等方面。以下是 MPT 在以太坊中的使用案例：

状态存储：

以太坊使用 MPT 来存储区块链中的状态信息，包括所有账户的余额、合约代码和存储值等。

MPT 通过将各个状态信息存储在不同的叶子节点上，然后通过哈希值建立索引，实现高效的状态访问和更新。

账户验证：

以太坊的账户地址是由公钥经过哈希函数得到的，账户地址被用作标识用户或合约的唯一标识。

MPT 可以用来验证账户地址的有效性，通过检查账户地址对应的节点是否存在于 MPT 中，并比较存储在该节点的公钥哈希与账户地址的哈希是否匹配。

区块验证：

以太坊使用 MPT 来构建区块头的状态树根哈希值，以确保区块的完整性和正确性。

当验证一个区块时，通过计算区块头中的状态树根哈希值，并与先前的状态树根哈希值进行比较，以确保区块数据没有被篡改。

合约存储：

以太坊中的智能合约可以使用 MPT 来存储和访问合约的状态和数据。

MPT 提供了高效的合约存储机制，通过将合约的状态信息存储在 MPT 中，使得合约执行和交互更加高效和可靠。

Merkle Patricia Tree 在以太坊中的使用案例不仅限于上述几个方面，还包括账户历史记录、交易存储、事件索引等。它的高效性、可扩展性和数据完整性保证了区块链系统的正确运行和强大的安全性

八、改进方向

存储效率：MPT 的存储效率依赖于节点的大小和数量，有时可能产生冗余数据。为了提高存储效率，可以考虑使用压缩算法来减小数据的大小，并优化节点结构。此外，还可以探索新的数据结构和算法，以提高 MPT 的存储效率。

计算效率：随着 MPT 的增长，节点数量和层级的增加会导致查询和更新操作的复杂性增加。为了提高计算效率，可以考虑使用并行计算、缓存策略和优化算法，以减少 MPT 操作的时间复杂度，提高查询和更新的性能。

安全性：尽管 MPT 已经被广泛应用于区块链系统中，但仍然存在一些安全性方面的挑战。例如，可能存在节点伪造或篡改的风险。进一步的研究应该集中在提供更强的安全性保证，如加密技术、访问控制和验证机制。

可扩展性：由于区块链和分布式系统的增长，MPT 需要具备较好的可扩展性。研究者可以探索分布式 MPT 的设计和实现，以提高系统的扩展性并减少网

络通信的开销。

应用领域扩展：除了区块链和分布式系统之外，MPT 还可以在其他领域中得到应用和拓展。例如，可以将 MPT 应用于大规模数据存储和索引中，以提高数据查询的性能。

综上所述，MPT 是一个重要的数据结构，有许多可以改进和探索的方向。随着技术的不断发展，进一步的研究和创新将为 MPT 的应用和性能带来更大的提升。

九、参考文献

"MPT-tree: A merkle patricia tree-based blockchain storage structure." IEEE Transactions on Services Computing 12.4 (2019): 688-700. Han, Ting, Xueliang Ma, and Jie Wu.