**创新创业实践课project报告**

靳立伟 202100460079

处理器：AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz

\*Project1: implement the naïve birthday attack of reduced SM3

方式：

SM3哈希函数的输出长度为256位，因此生日攻击的目标是在2^128的时间复杂度内找到碰撞。但是现有的技术是达不到的，因此我们将长度缩短来尝试生日攻击。

生日攻击需要存储大量的哈希值和相应的输入。为了找到碰撞，需要建立一个哈希表，以存储已经计算过的哈希值和对应的输入。

开始进行生日攻击前，需要生成大量随机的输入。可以使用伪随机数生成器来创建输入。

计算哈希并查找碰撞：对于每个生成的随机输入，使用SM3哈希函数计算其哈希值，并将该哈希值与哈希表中已有的哈希值进行比较。如果找到了相同的哈希值，那么就找到了一个碰撞。

攻击完成后，需要检查已经找到的碰撞是否满足攻击的要求。在生日攻击中，通常需要多次运行攻击来找到满足要求的碰撞。

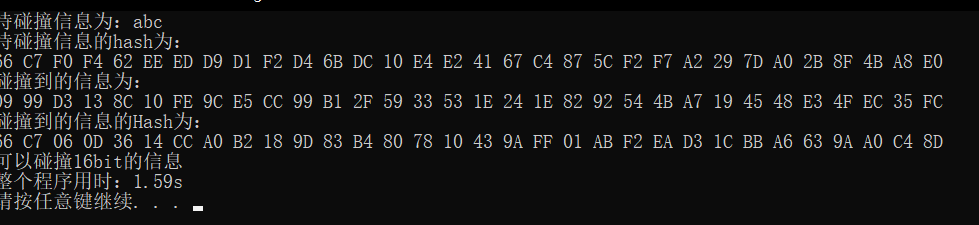
效果：109.88872194290161 s

\*Project2: implement the Rho method of reduced SM3

方式：

按照rho方法，将输出作为下一步的输入，之后迭代循环，直到hash值发生碰撞，因sm3的碰撞开销很大，所以将长度缩短减少时间。

效果：



用时1.59s

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

方式：

在长度扩展攻击中，攻击者利用哈希函数的特性，在已知哈希值H(M)和消息M的情况下，构造另一个消息M'，使得哈希值H(M')可以被计算出来，而不需要知道M'的实际内容。SM3的长度扩展攻击是基于Merkle-Damgård结构的哈希函数特性。

伪代码：

Initialize IV (Initialization Vector) with some constants

Divide the input message M into blocks: M = M1 || M2 || ... || Mn

for each block Mi:

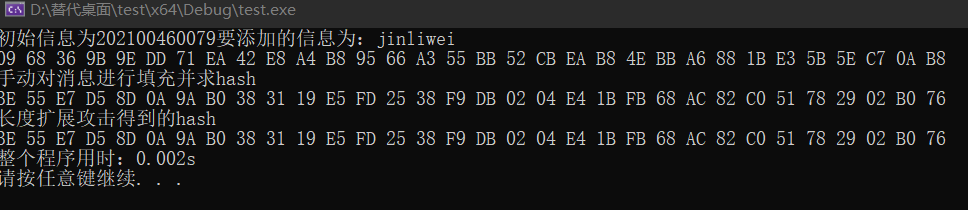
Compress(IV, Mi) # Compress the IV and the block Mi to update IV

Output the final value of IV as the hash value H(M)

Compress函数是压缩函数，它将当前的IV和一个消息块Mi作为输入，输出新的IV。在长度扩展攻击中，攻击者知道消息M的哈希值H(M)，并希望构造新消息M'，使得H(M')可以被计算出来。

在构造新的消息块M'时，需要根据原始消息的长度来合理填充数据，使得新消息的长度满足哈希函数的块长度要求。

效果：



用时：0.002s

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

SM3适用于商用密码应用中的数字签名和验证，是在SHA-256基础上改进实现的一种算法，其安全性和SHA-256相当。 SM3和MD5的迭代过程类似，也采用Merkle-Damgard结构。消息分组长度为512位，摘要值长度为256位。

整个算法的执行过程可以概括成四个步骤：消息填充、消息扩展、迭代压缩、输出结果。

由于其M-D结构，SM3无法利用多线程将消息分组同时加密，因为前者的结果会在后面形成依赖。于是采用预计算64常量、多线程多次加密不同文件，并且循环展开关键步骤、然后同时调用多个SM3模块进行多线程优化，提高吞吐量。

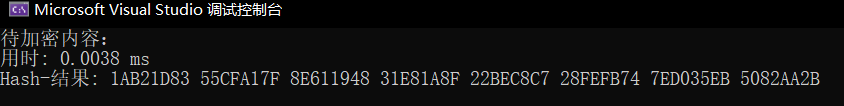
使用unroll，现代 CPU 采用流水线设计 (可以类比汽车生产流水线，分多阶段)，把一条机器指令的执行分成多个阶段，然后每个时钟周期尽可能多地执行不同指令的不同阶段，为了让流水线更好地流懂，我们可以在写程序时可以地减少程序相邻行的数据相关，于是我们通过循环展开即增加每次迭代计算元素的数量，减少循环迭代次数，更好地增加指令的并行性

采用多个线程，每个线程完成多次sm3，计算多次加密不同文件，然后同时调用多个SM3模块进行多线程优化，能够达到提高吞吐量的效果。

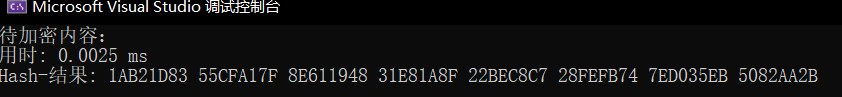
预计算64个常量并存储。优化后，虽然会牺牲256字节的存储空间，但可以避免每个消息分组都去进行常数的移位操作。且运算时间几乎是和消息大小成正比的，消息越大，优化效果越明显，即缩短的运算时间就越多，性价比越高。

效果：

未优化：



优化后：



优化前用时0.0038ms，优化后用时0.0025ms

\*Project5: Impl Merkle Tree following RFC6962

方式：

创建：本次实验中，我们先用random.randint生成10万的数据作为区块数据，再据此生成merkle tree。

检查：当我们检查或者证明某个节点是否存在时，只需给出该节点的相邻兄弟节点和父节点即可，具体方式是对于待证明的节点，我们只需要确切找出它的hash值是否与树中某节点hash值相同以及该节点在树中的位置，可以求解其在list中对应的下标n；

我们需要讨论待求是偶节点还是奇节点。根据其下标，如果是奇数：首先检查是否是最后一个节点。 如果是，则不操作，进入下一个循环。

（即本层没有该节点的兄弟节点，即在Merkle Tree的逻辑表示中，该节点并不位于本层，而是在更高的层。）

找到该节点之后，将其右节点（下标+1的节点）与该节点一起写入列表。 如果是偶数：将其左节点（下标为-1的节点）与该节点一起写入列表；之后我们就可反复地调用上面的程序，最后将根节点放入树中。

验证是否其父节点的hash值等于hash（子节点的hash值之和），然后再照此验证其祖父节点。此外我们还应该检查该树是否符合merkle tree的定义结构。

效果：

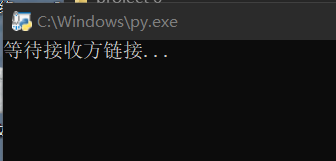


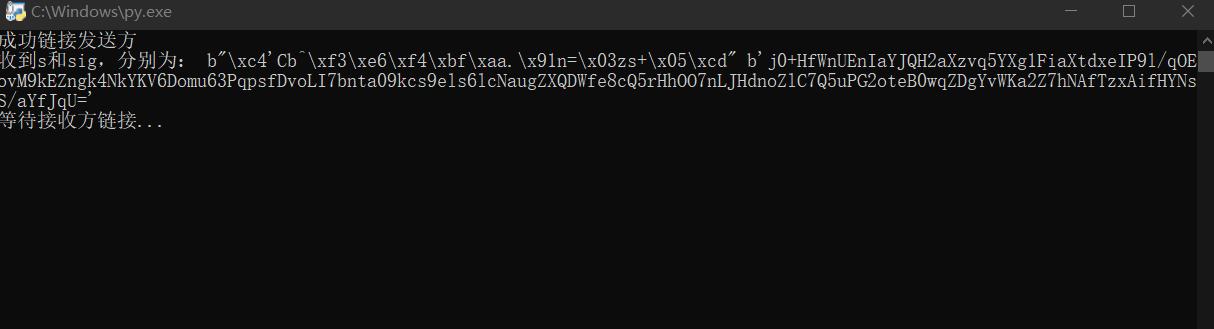
用时：103023us

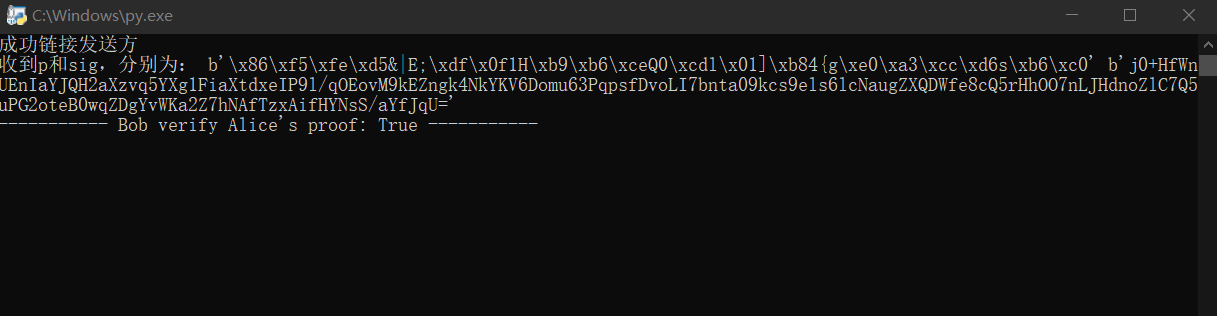
\*Project6: impl this protocol with actual network communication

实现方式：该例子主要是应用到了hash-chain和签名。hash-chain即对某一对象多次hash，在本实验实现中，因hash函数接受的是字节流，所以对hash后的结果进行.digest()即可得到字节流。本实验的两个hash函数分别为SHA1和SHA256，签名为RSA签名。该年龄问题根据ppt中的算法实现即可。在实际的网络通信中，我们通过主机和端口号，套子节实现。本实验需要注意的是hash函数的输入和输出的类型，因此在输入为整数时，需转化为字符串，再转化为字节流。而输出是一个class，需取digest得到字节流。签名和验证的输入输出也都要注意类型。

效果：

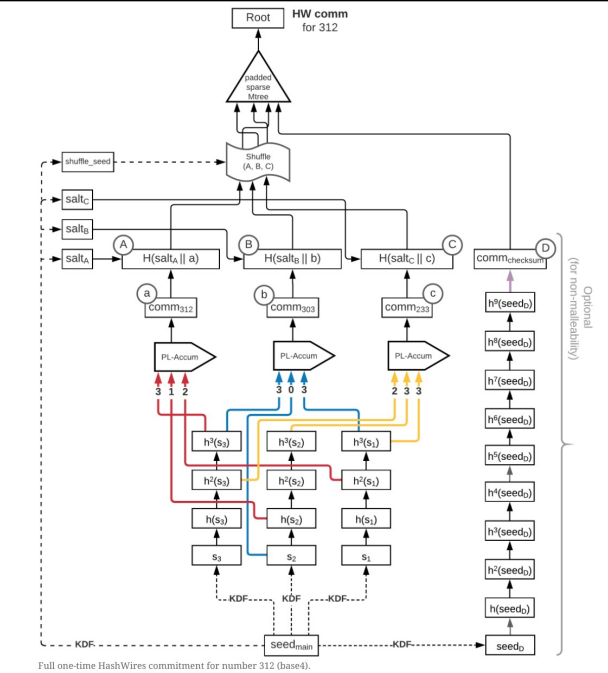






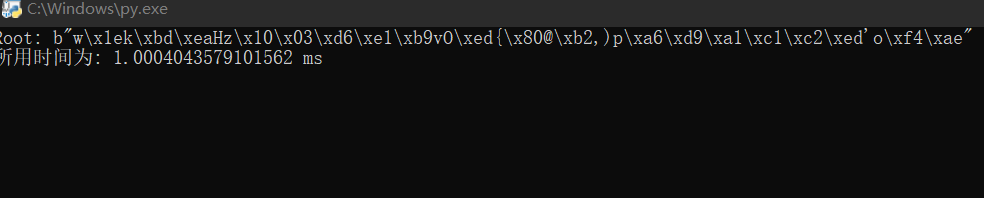
\*Project7: Try to Implement this scheme

方式：



根据图示书写代码，其中主要还是使用了与上一个project相同的hash-chain。在开始时，还需要使用一个KDF，本实验是根据线性同余法编写自己的KDF。还有一个shuffle()函数，根据一个shuffle种子，然后初始化随机数生成器，再使用random.shuffle()对哈希值列表进行shuffle即可。本实验的KDF在编写时花费了很多时间，应该学会使用python中有的KDF。

效果：



用时：1ms

\*Project9: AES / SM4 software implementation

方式：

选择AES的密钥长度：AES支持128位、192位和256位密钥长度。根据需要，选择合适的密钥长度。

根据选择的密钥长度，对输入的密钥进行密钥扩展，生成轮密钥。

将要加密的数据按照128位（16字节）为一组进行划分。

将密钥与数据的第一组进行异或运算。

对数据的每一组进行若干轮的迭代加密，每一轮包括以下步骤：

字节代换：将数据块中的每个字节进行字节代换，使用S-Box（Substitution Box）进行替换。

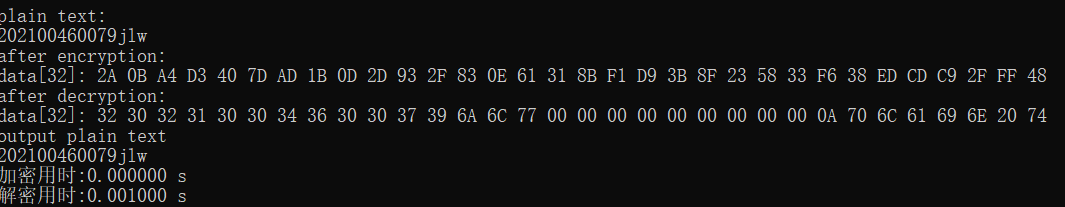
行移位：对数据块的每一行进行循环左移操作。

列混淆：对数据块进行列混淆操作，通过矩阵乘法实现。

轮密钥加：将当前轮的轮密钥与数据块进行异或运算。

最后一轮：执行最后一轮迭代加密，不包括列混淆步骤。

效果：



加密和解密所用时间都很少，有时在该计时函数下加解密的的用时都为0，可见效率之快。

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

方式：

首先需要选择合适的椭圆曲线参数，包括椭圆曲线的方程、基点、椭圆曲线的域参数等。这些参数的选择直接影响到ECDSA的安全性和性能。

使用椭圆曲线参数生成一对公钥和私钥。公钥是由基点乘上私钥得到的椭圆曲线上的点，私钥是一个随机的大整数。

使用私钥对消息进行签名。签名过程包括以下步骤：

计算消息的哈希值：通常使用哈希函数（如SHA-256）对消息进行哈希，生成消息的摘要。

选择随机数：选择一个随机数作为签名过程的辅助参数。

计算签名：通过一系列椭圆曲线运算，将哈希值、私钥和随机数结合在一起，生成签名。

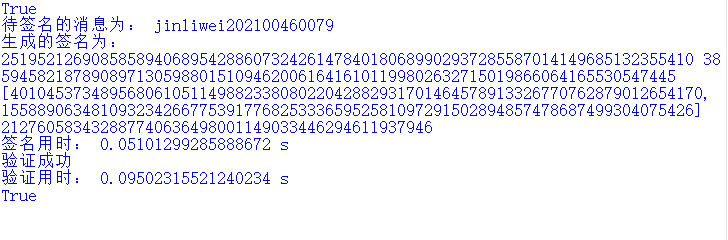
验证：使用公钥和签名来验证消息的真实性。验证过程包括以下步骤：

计算消息的哈希值：与签名时使用的哈希函数相同，生成消息的摘要。

计算验证参数：通过一系列椭圆曲线运算，将公钥、哈希值和签名结合在一起，计算出验证参数。

比较验证参数：将计算得到的验证参数与签名中的一部分进行比较，如果相等，则说明签名有效。

效果：



\*Project11: impl sm2 with RFC6979

方式：

SM2是中国国家密码管理局发布的椭圆曲线公钥密码算法，RFC6979是用于确定性签名的随机数生成的标准。结合SM2和RFC6979，可以实现SM2的确定性签名，即使用确定性的方法生成签名过程中所需的随机数，从而提高签名的安全性。

以下是SM2与RFC6979结合的简要实现方式：

首先需要选择合适的椭圆曲线参数，包括椭圆曲线的方程、基点（Base Point）、椭圆曲线的域参数等。这些参数的选择直接影响到SM2的安全性和性能。

使用SM2的椭圆曲线参数生成一对公钥和私钥。

使用私钥对消息进行签名。签名过程包括以下步骤：

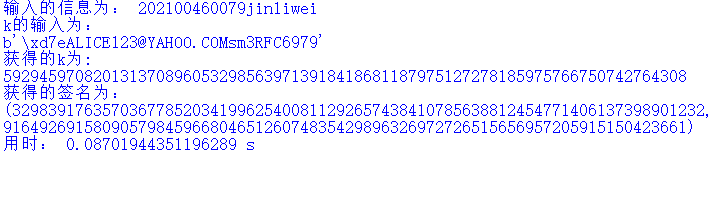
使用哈希函数对消息进行哈希，生成消息的摘要。

使用RFC6979生成确定性随机数：RFC6979规定了一种生成签名过程中所需的确定性随机数的方法，该随机数是基于消息、私钥和哈希值计算得到的，从而避免了使用真正的随机数，减少了潜在的安全风险。

计算签名：通过一系列椭圆曲线运算，将哈希值、私钥和RFC6979生成的随机数结合在一起，生成签名。

验证过程：使用公钥和签名来验证消息的真实性。验证过程与普通的SM2签名验证相同，不受RFC6979的影响。

效果：



用时0.087s

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

方式：

本次实验中共验证的内容有

1.泄露随机数k从而导致d泄露。

2.对不同的消息重用随机数k从而导致d泄露。

3.两个不同的用户使用同一个k，则其中一个人可以推出另一个人的私钥d。

4.可以(r,s) and (r,-s)均为有效签名，这可能会导致区块网络分裂。

5.使用相同的d和随机数k签发SM2和ECDSA或Schnorr和ECDSA，会导致d泄露。

效果展示（sm2）



\*Project13: Implement the above ECMH scheme

方式：

首先需要选择一个合适的椭圆曲线。该曲线的参数（方程、基点、域参数等）直接影响到 ECMH 的安全性和性能。

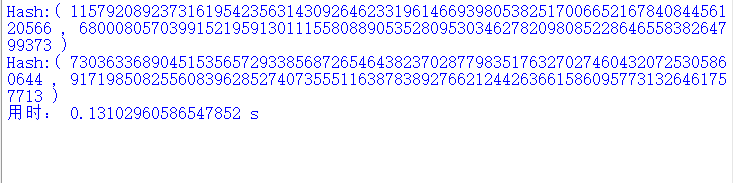
将要进行哈希的多集表示为多项式的形式。在多项式中，每个元素对应一个系数，而重复的元素则对应相同系数的多项式项。

对多项式中的每个系数进行椭圆曲线点运算。具体运算包括点加法、点倍乘等操作。

将得到的所有椭圆曲线点的坐标（X, Y 坐标值）组合起来，形成最终的 ECMH 哈希值。通常，可以将 X 和 Y 坐标值按照一定的规则连接在一起，生成最终的哈希值。

实验中需要求解二次剩余，因此我们使用勒让德符号来进行求解。还需要写出一些点的计算。

效果：



用时：0.13s

\*Project14: Implement a PGP scheme with SM2

方式：

选择SM2的椭圆曲线参数：首先需要选择合适的SM2椭圆曲线参数，包括椭圆曲线的方程、基点、椭圆曲线的域参数等。这些参数的选择直接影响到SM2的安全性和性能。

密钥生成：用户生成一对SM2密钥，包括公钥和私钥。公钥用于加密和验证签名，可以公开分享给其他人。私钥用于解密和生成数字签名，必须保密保存。

加密过程：

* + PGP方案中，消息的加密通常使用混合加密技术，先用对称加密（SM4）加密消息，再用接收方的公钥（SM2）加密对称加密的密钥。
  + 在SM2中，使用接收方的公钥对对称加密的密钥进行加密。这样加密后的对称加密密钥只能被接收方使用其私钥进行解密，保证了对称加密密钥的安全传输。

解密过程：

* + PGP方案中，接收方使用自己的私钥（SM2）解密接收到的对称加密的密钥，得到原始的对称加密密钥。
  + 使用得到的对称加密密钥解密消息。

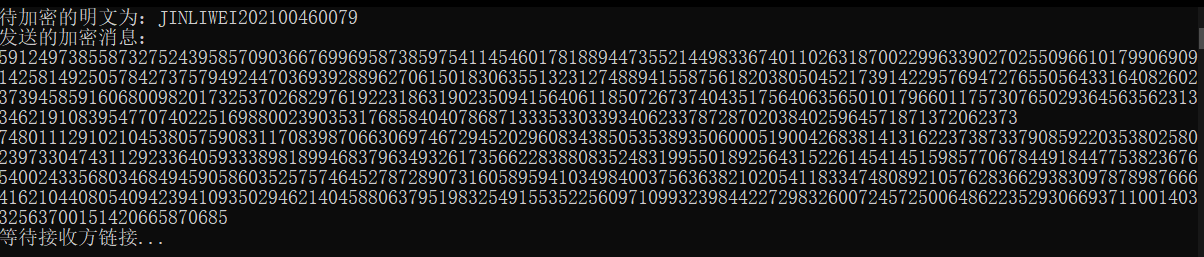
数字签名过程：

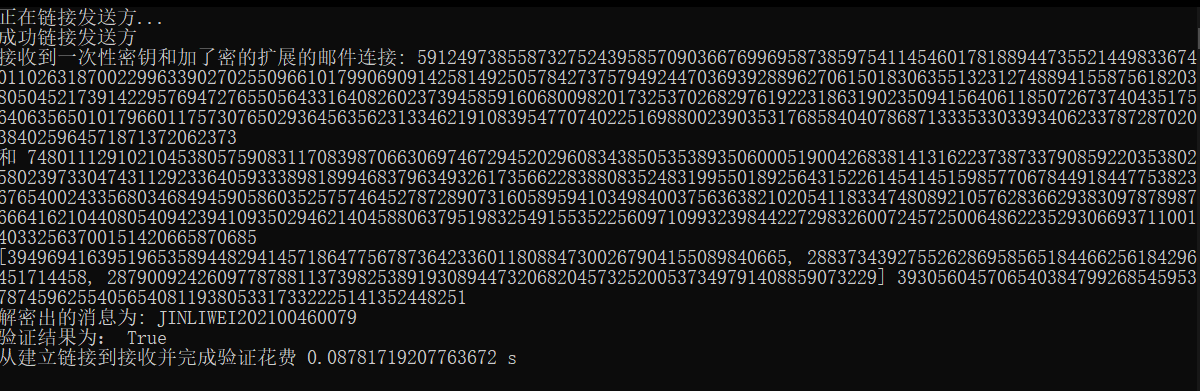
* + PGP方案中，发送方使用自己的私钥（SM2）对消息的哈希值进行数字签名。
  + 将数字签名与原始消息一起发送给接收方。

验证签名：

* + 接收方使用发送方的公钥（SM2）对收到的数字签名进行验证，确认消息的真实性和完整性。

效果：





接收方接受并验证的时间为：0.0878s

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

方式：

密钥生成：

Alice 生成自己的密钥对：SM2 公钥和私钥（PK\_A, SK\_A）。

Bob 生成自己的密钥对：SM2 公钥和私钥（PK\_B, SK\_B）。

协商随机数：

Alice 和 Bob 协商生成一个共同的随机数 R，该随机数用于在签名过程中引入随机性，增加签名的安全性。他们可以通过安全通信渠道或密钥协商协议来生成共同的随机数。

签名过程：

Alice 和 Bob 分别对待签名的消息 M 计算哈希值 H(M)。

Alice 计算自己的部分签名 S\_A = (r, s\_A)：

选择一个随机数 k\_A。

计算椭圆曲线点 k\_A \* G，其中 G 是椭圆曲线的基点。

将 X 坐标值 r 作为签名的一部分。

计算 s\_A = (H(M) + r \* SK\_A) \* k\_A^(-1) mod n，其中 n 是椭圆曲线群的阶。

Bob 计算自己的部分签名 S\_B = (r, s\_B)：

选择一个随机数 k\_B。

计算椭圆曲线点 k\_B \* G。

将 X 坐标值 r 作为签名的一部分。

计算 s\_B = (H(M) + r \* SK\_B) \* k\_B^(-1) mod n。

Alice 和 Bob 将各自的部分签名 S\_A 和 S\_B 交换。

组合签名：

Alice 和 Bob 汇总各自的部分签名 S\_A 和 S\_B，得到最终的两方协同签名 S = (r, s)：

r 为共同的随机数 R。

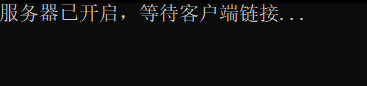
s = (s\_A + s\_B) mod n。

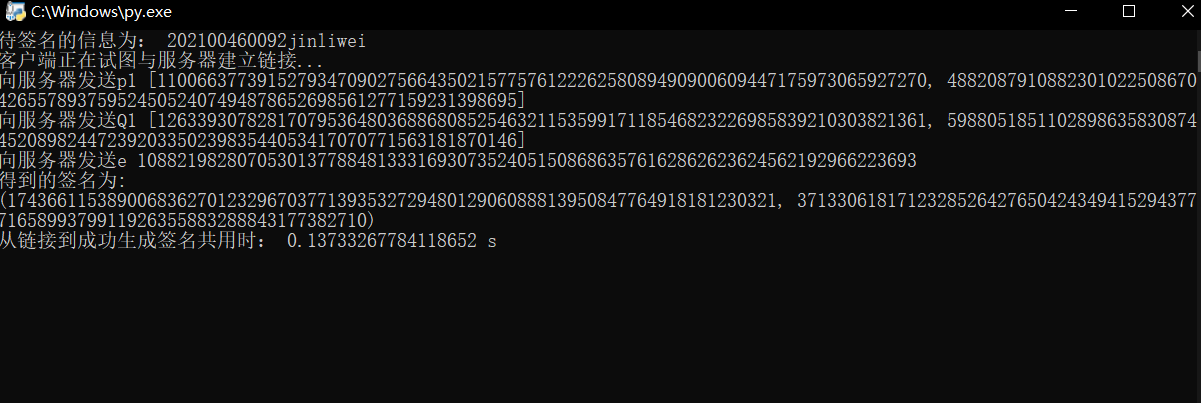
验证签名：

任何人可以使用公钥 PK\_A 和 PK\_B 来验证签名的有效性。

验证过程类似于普通的 SM2 签名验证，即验证 s \* G 是否等于 r + H(M) \* PK。

效果：





连接并签名用时：0.137s

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

方式：

密钥生成：

Alice 生成自己的密钥对：SM2 公钥和私钥（PK\_A, SK\_A）。

Bob 生成自己的密钥对：SM2 公钥和私钥（PK\_B, SK\_B）。

加密过程：

信息发送者使用 Alice 和 Bob 的公钥（PK\_A 和 PK\_B）对消息 M 进行加密，生成密文 C。

协商随机数：

Alice 和 Bob 协商生成一个共同的随机数 R，该随机数用于在解密过程中引入随机性，增加解密的安全性。他们可以通过安全通信渠道或密钥协商协议来生成共同的随机数。

解密过程：

Alice 和 Bob 分别对密文 C 进行解密。

Alice 计算自己的部分解密结果 D\_A：

使用她的私钥 SK\_A 对 C 进行解密，得到中间结果 D\_A'。

使用共同的随机数 R 对 D\_A' 进行异或运算，得到 D\_A = D\_A' ⊕ R。

Bob 计算自己的部分解密结果 D\_B：

使用他的私钥 SK\_B 对 C 进行解密，得到中间结果 D\_B'。

使用共同的随机数 R 对 D\_B' 进行异或运算，得到 D\_B = D\_B' ⊕ R。

合并解密结果：

Alice 和 Bob 将各自的部分解密结果 D\_A 和 D\_B 交换。

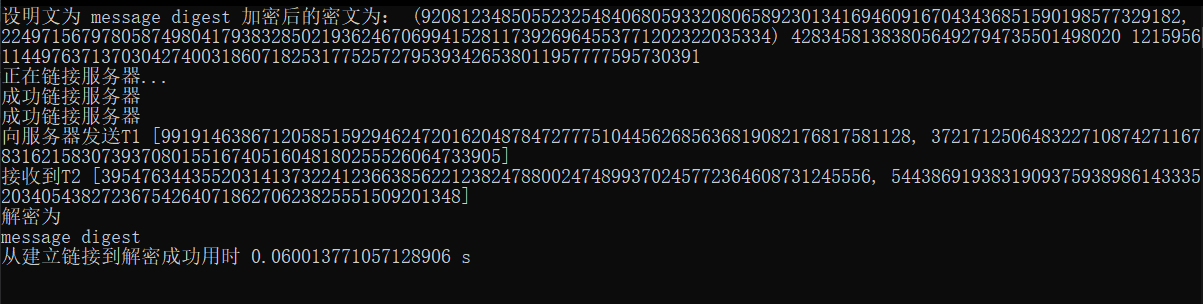
两方合并各自的解密结果，得到最终的明文 M = D\_A ⊕ D\_B。

验证解密：

验证解密的正确性，确保得到的明文 M 是正确的原始消息。

效果：





建立连接并解密用时0.06s

\*Project17：比较Firefox和谷歌的记住密码插件的实现区别

在声明权限时，Firefox使用"permissions"字段，而谷歌浏览器使用"permissions"字段或"optional\_permissions"字段来声明不同类型的权限。

firefox:

"manifest\_version": 2,

"name": "Your Password Manager",

"version": "1.0",

"description": "A password manager extension for Firefox",

"permissions": [

"tabs",

"storage"

],

"content\_scripts": [

{

"matches": ["<all\_urls>"],

"js": ["content.js"]

}

],

"browser\_action": {

"default\_icon": "icon.png",

"default\_popup": "popup.html"

}

google:

"manifest\_version": 2,

"name": "Your Password Manager",

"version": "1.0",

"description": "A password manager extension for Chrome",

"permissions": [

"tabs",

"storage"

],

"content\_scripts": [

{

"matches": ["<all\_urls>"],

"js": ["content.js"]

}

],

"browser\_action": {

"default\_icon": "icon.png",

"default\_popup": "popup.html"

}

在获取当前选项卡信息时，Firefox使用'browser.tabs' API，而谷歌浏览器使用'chrome.tabs' API。

firefox:

browser.tabs.query({ currentWindow: true, active: true }, function(tabs) {

const currentTab = tabs[0];

});

google:

chrome.tabs.query({ currentWindow: true, active: true }, function(tabs) {

const currentTab = tabs[0];

});

对登录表单提交事件的监听和内容脚本的注入的不同

firefox:

document.addEventListener("submit", function(event) {

});

google:

document.addEventListener("submit", function(event) {

});

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

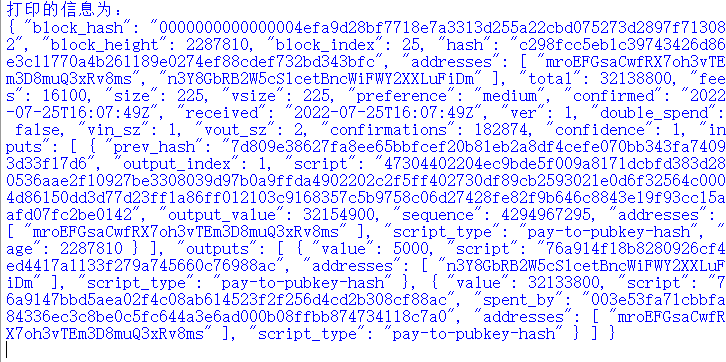
方式：

Step1：首先对比特币测试网交易信息中的一个交易进行分析，通过python脚本可以得到该交易的区块头、难度、时间戳、Merkle根输入、"preference"、"confirmed"、"received"、 等参数信息

Step2：将tx信息发送到指定网站，通过python脚本打印回复信息。

运行期间不能使用VPN

效果：





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

方式：

如果只需要签名消息的Hash值，那么任何人都可以伪造签名。

Ecdsa验证旨在验证：

(eG+rP) = (x',y')=R',r'=x' mod n == r

为了伪造，我们可以选择u,v,计算R'=(x',y')=uG+vP，之后选择r'=x' mod n，计算：

e'=r'uv' mod n

s'=r'v' mod n

伪造即可完成

此代码因随机数选取有几率报错，原因是椭圆曲线部分随机选取了错误的点

利用椭圆曲线若两点同时不为0，则可计算斜率.

若两点相等，斜率为：λ = (3Xp² + a)/2Yp

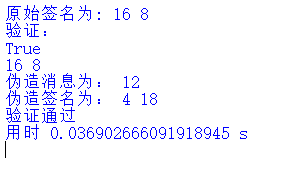
若两点不相等，则斜率为：λ = (Yq - Yp)/(Xq - Xp)

计算结果点R的坐标计算公式为：

Xr = (λ² - Xp - Xq)

Yr = (λ(Xp - Xr) - Yp)

椭圆曲线点数乘法：循环使用加法即可

效果：  


用时：0.0396

\*Project21: Schnorr Bacth

方式：

假设有一个椭圆曲线上的点乘法操作为 "\*"，基点为 G，并且我们有一个私钥 k 和对应的公钥 Q = k \* G。

密钥生成：

私钥：选择一个随机整数 k，并保持为私钥。

公钥：计算公钥 Q = k \* G。

签名过程： 假设消息的哈希值为 h，辅助随机数为 r。

计算辅助点 R = r \* G。

计算挑战值 e = H(h || R || Q)，其中 H 是哈希函数。

计算签名的 s 值：s = r + e \* k mod n，其中 n 是椭圆曲线群的阶。

验证过程： 假设收到的签名为 (s, e)，公钥为 Q，消息的哈希值为 h。

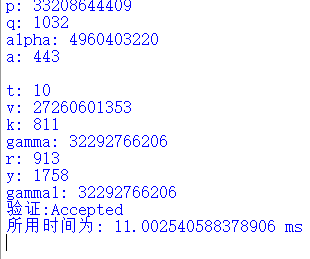
验证 s 和 e 的取值是否满足 0 < s < n 和 0 < e < n。

计算挑战值 e' = H(h || s \* G - e \* Q)。

如果 e' 等于 e，则签名有效；否则，签名无效。

按照Schnorr签名算法，因需要很多计算，所以在caculate.py文件中写了离散对数的求解，素数和素因子的生成以及计算生成元。这样就可以在Schnorr.py文件中直接使用这些计算。

效果：



用时：11ms

\*Project22: research report on MPT

方式：

MPT 的结构如下：

每个节点包含一个或多个 Key-Value 对。

Key 是数据的哈希值或一系列字节组成的前缀，用于唯一标识一个 Value。

Value 可以是实际数据，也可以是另一个 MPT 节点的哈希值，从而形成递归结构。

prefix就是HP编码！对终止符的状态和奇偶性进行编码。最低位表示奇偶性，第二低位编码终止符状态。

总共有2个扩展节点，2个分支节点，4个叶子节点。 右边是叶子节点的情况，左边代表的是节点的prefix（HP编码）

这就是一个状态树的存储形式，其实他应当长的样子，我们可以细细想一下，他的key被编码成一种特殊的16进制的表示，value是一些rlp后的数据，而且比上图要大的多的多。

总的来说相对于普通的前缀树，MPT树能有效减少Trie树的深度，增加Trie树的平衡性。而且通过节点的hash值进行树的节点的链接，有助于提高树的安全性和可验证性，是Trie和Merkle树混合加上平衡操作后的产物。

update \_update\_and\_delete\_storage(self, node, key, value)：

i. 如果node是空节点，直接返回[pack\_nibbles(with\_terminator(key)), value]，即对key加上终止符，然后进行HP编码。

ii. 如果node是分支节点，如果key为空，则说明更新的是分支节点的value，直接将node[-1]设置成value就行了。如果key不为空，则递归更新以key[0]位置为根的子树，即沿着key往下找，即调\_update\_and\_delete\_storage(self.\_decode\_to\_node(node[key[0]]), key[1:], value)。

iii. 如果node是kv节点（叶子节点或者扩展节点），调用\_update\_kv\_node(self, node, key, value)，见步骤iv

iv. curr\_key是node的key，找到curr\_key和key的最长公共前缀，长度为prefix\_length。Key剩余的部分为remain\_key，curr\_key剩余的部分为remain\_curr\_key。

a) 如果remain\_key==[]== remain\_curr\_key，即key和curr\_key相等，那么如果node是叶子节点，直接返回[node[0], value]。如果node是扩展节点，那么递归更新node所链接的子节点，即调用\_update\_and\_delete\_storage(self.\_decode\_to\_node(node[1]), remain\_key, value)

b) 如果remain\_curr\_key == []，即curr\_key是key的一部分。如果node是扩展节点，递归更新node所链接的子节点，即调用\_update\_and\_delete\_storage(self.\_decode\_to\_node(node[1]), remain\_key, value)；如果node是叶子节点，那么创建一个分支节点，分支节点的value是当前node的value，分支节点的remain\_key[0]位置指向一个叶子节点，这个叶子节点是[pack\_nibbles(with\_terminator(remain\_key[1:])), value]

c) 否则，创建一个分支节点。如果curr\_key只剩下了一个字符，并且node是扩展节点，那么这个分支节点的remain\_curr\_key[0]的分支是node[1]，即存储node的value。否则，这个分支节点的remain\_curr\_key[0]的分支指向一个新的节点，这个新的节点的key是remain\_curr\_key[1:]的HP编码，value是node[1]。如果remain\_key为空，那么新的分支节点的value是要参数中的value，否则，新的分支节点的remain\_key[0]的分支指向一个新的节点，这个新的节点是[pack\_nibbles(with\_terminator(remain\_key[1:])), value]

d) 如果key和curr\_key有公共部分，为公共部分创建一个扩展节点，此扩展节点的value链接到上面步骤创建的新节点，返回这个扩展节点；否则直接返回上面步骤创建的新节点

v. 删除老的node，返回新的node

delete \_delete\_and\_delete\_storage(self, key)：

i. 如果node为空节点，直接返回空节点

ii. 如果node为分支节点。如果key为空，表示删除分支节点的值，直接另node[-1]=‘’, 返回node的正规化的结果。如果key不为空，递归查找node的子节点，然后删除对应的value，即调用self.\_delete\_and\_delete\_storage(self.\_decode\_to\_node(node[key[0]]), key[1:])。返回新节点

iii. 如果node为kv节点，curr\_key是当前node的key。

a) 如果key不是以curr\_key开头，说明key不在node为根的子树内，直接返回node。

b) 否则，如果node是叶节点，返回BLANK\_NODE if key == curr\_key else node。

c)如果node是扩展节点，递归删除node的子节点，即调用\_delete\_and\_delete\_storage(self.\_decode\_to\_node(node[1]), key[len(curr\_key):])。如果新的子节点和node[-1]相等直接返回node。否则，如果新的子节点是kv节点，将curr\_key与新子节点的可以串联当做key，新子节点的value当做vlaue，返回。如果新子节点是branch节点，node的value指向这个新子节点，返回。

get \_get(self, node, key)：

i. 如果node是空节点，返回空节点

ii. 如果node是分支节点，如果key为空，返回分支节点的value；否则递归查找node的子节点，即调用\_get(self.\_decode\_to\_node(node[key[0]]), key[1:])

iii. 如果node是叶子节点，返回node[1] if key == curr\_key else ‘’

iv. 如果node是扩展节点，如果key以curr\_key开头，递归查找node的子节点，即调用\_get(self.\_decode\_to\_node(node[1]), key[len(curr\_key):])；否则，说明key不在以node为根的子树里，返回空

insert：

插入操作基于查找完成，首先找到与新插入节点拥有最长相同路径前缀的节点，之后基于此节点将剩下的节点进行封装插入。

效果：

