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

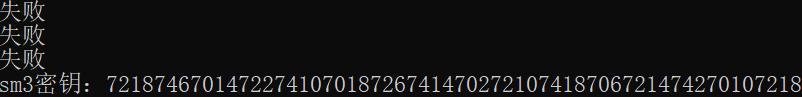
**实验原理**

生日攻击的目的是寻求一个基于sm3哈希值的弱碰撞，原理是一定长度和hash值结果2^32长度，在2^16密文空间中可以以50%以上的概率找到一个hash碰撞。

这里我们一边存表一边查表 ，以便可以在较短时间内找到一个前16bit的hash弱碰撞。

如果寻找更长bit的碰撞，寻找时间也会相应变长。

**实现结果：**



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

通过使用位运算 来代替乘法和除法，可以提高运算速度。

使用SIMD指令集以及多线程技术来利用多核处理器的计算能力，加快计算速度。

**实现结果：**



# Project8

# AES impl with ARM instruction

使用ARM指令集实现AES是利用ARM处理器提供的密码扩展功能。这些扩展包括加速AES加密和解密操作的指令。

**实现方式：**

密钥扩展：实现密钥扩展算法，使用AES密钥表生成一组轮密钥。

**SubBytes：**使用ARM指令执行SubBytes操作，将状态的每个字节替换为AES S-盒查找表中对应的字节。

**ShiftRows**：利用ARM指令执行ShiftRows操作，循环移动状态的最后三行。

**MixColumns：**使用ARM指令实现MixColumns操作，对状态的列应用线性变换。

**AddRoundKey：**将状态的每个字节与轮密钥的相应字节进行异或运算。

根据AES密钥长度，多轮重复上述四个步骤

**\*Project9 AES/SM4softwareimplementation**

AES算法具体的步骤如下**：**

初始化：选择合适的密钥长度，并根据密钥生成扩展密钥和轮密钥。

加密轮处理：AES算法采用多轮迭代处理来加密数据。每一轮包括以下四个步骤：

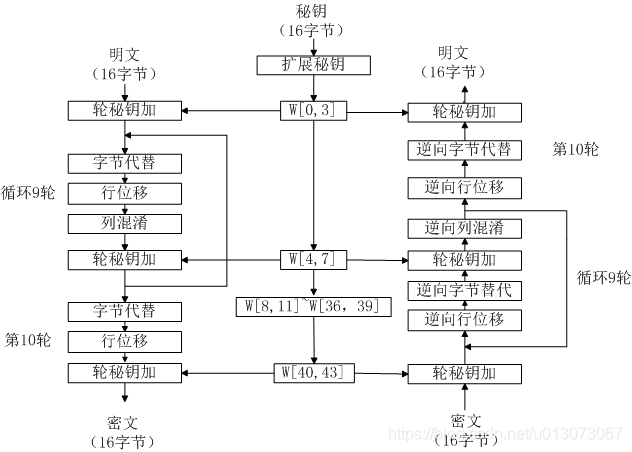
a. 字节替代：将输入数据块中的每个字节替换为S盒中对应的值。

b. 行移位：对输入数据块的行进行循环左移操作，不同行的偏移量依赖于数据块的行号。

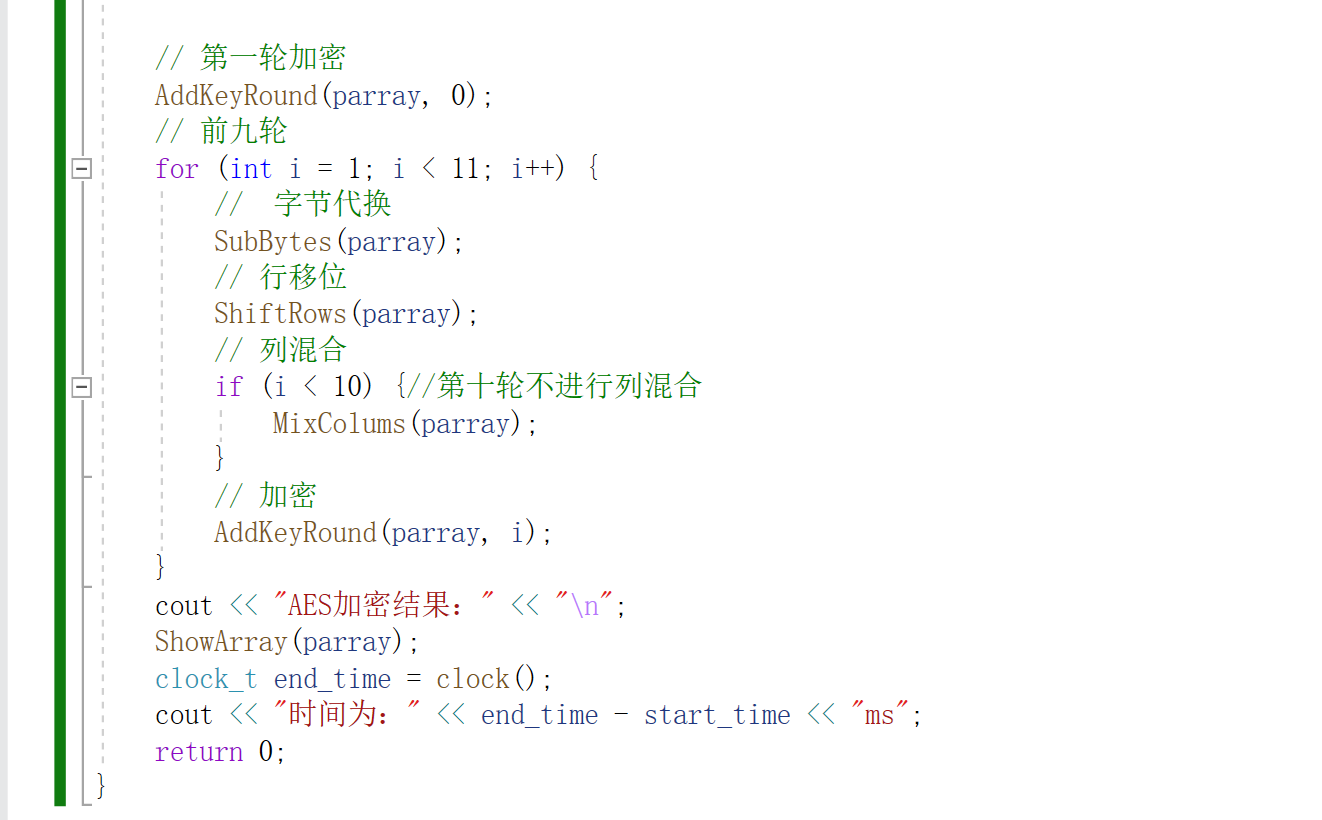
c. 列混淆：将矩阵乘法应用于输入数据块的每一列。这个步骤使用固定的混淆矩阵来改变数据块的列。

d. 轮密钥加：将当前轮的轮密钥与数据块进行按位异或运算。

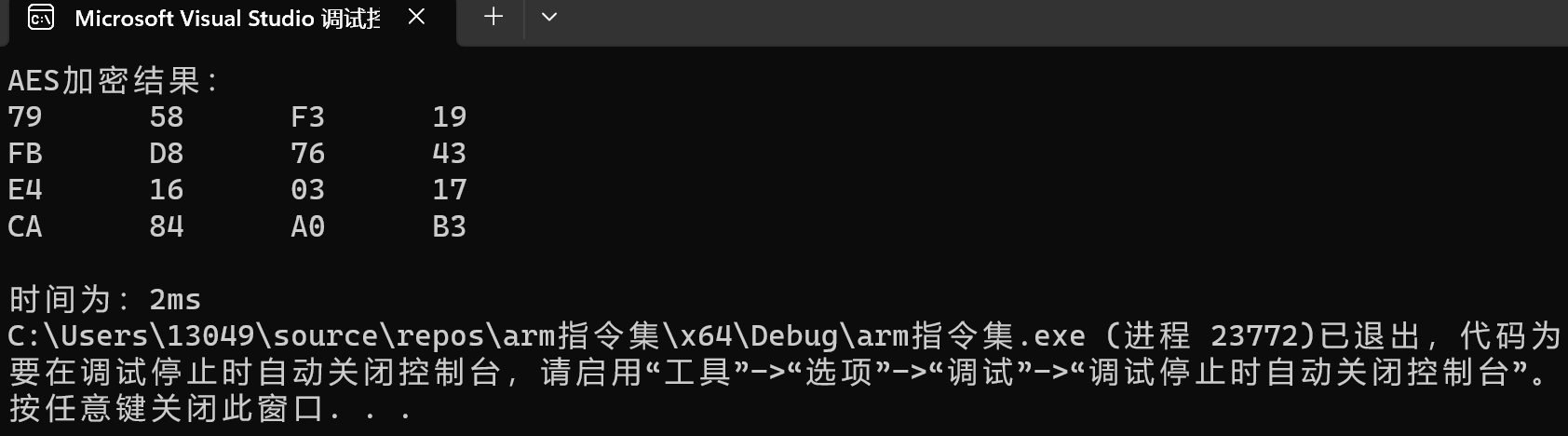
密文输出：经过多轮迭代后，最后一轮的输出即为密文。



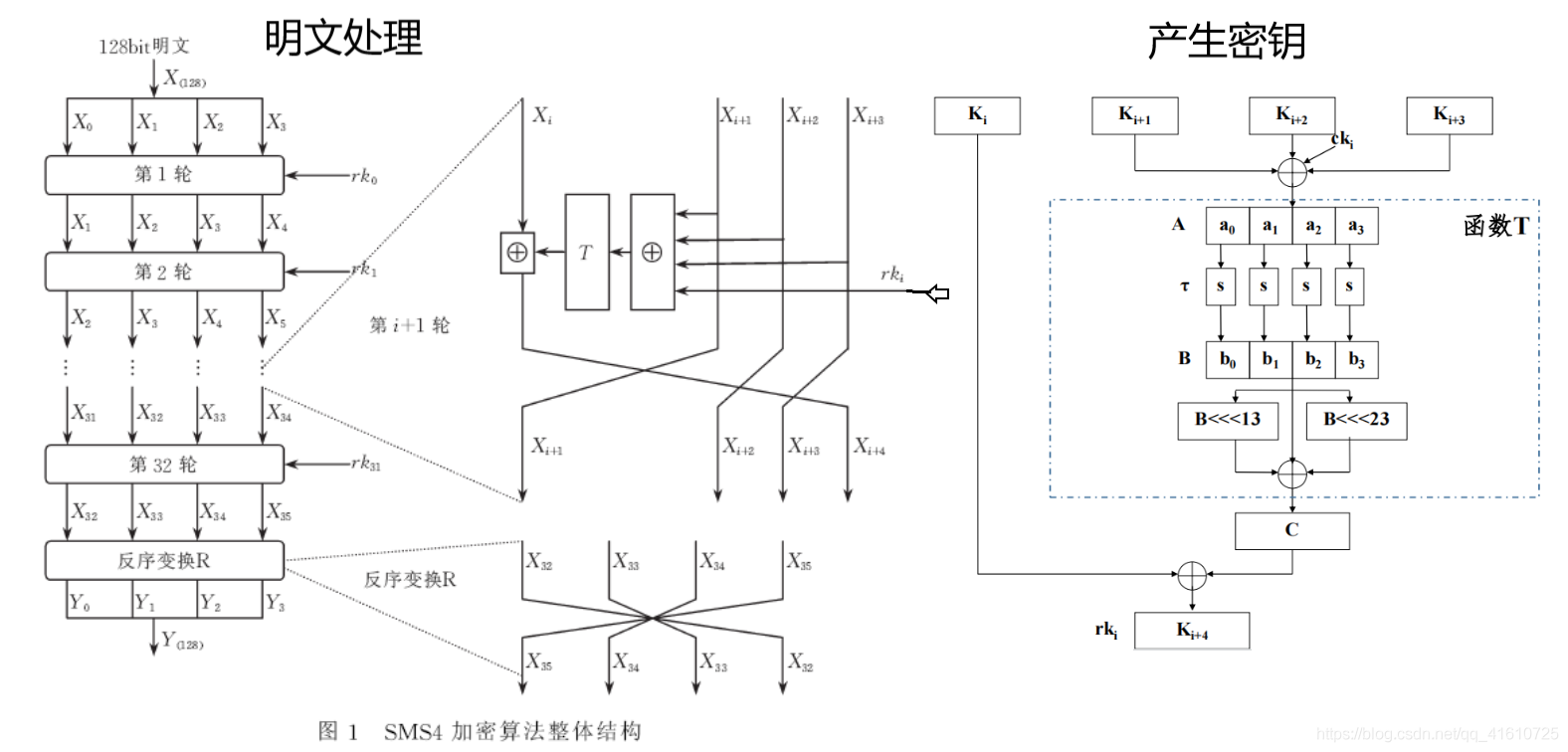




**实现结果**



# SM4算法



基本运算：SM4密码算法使用模2加和循环移位作为基本运算。

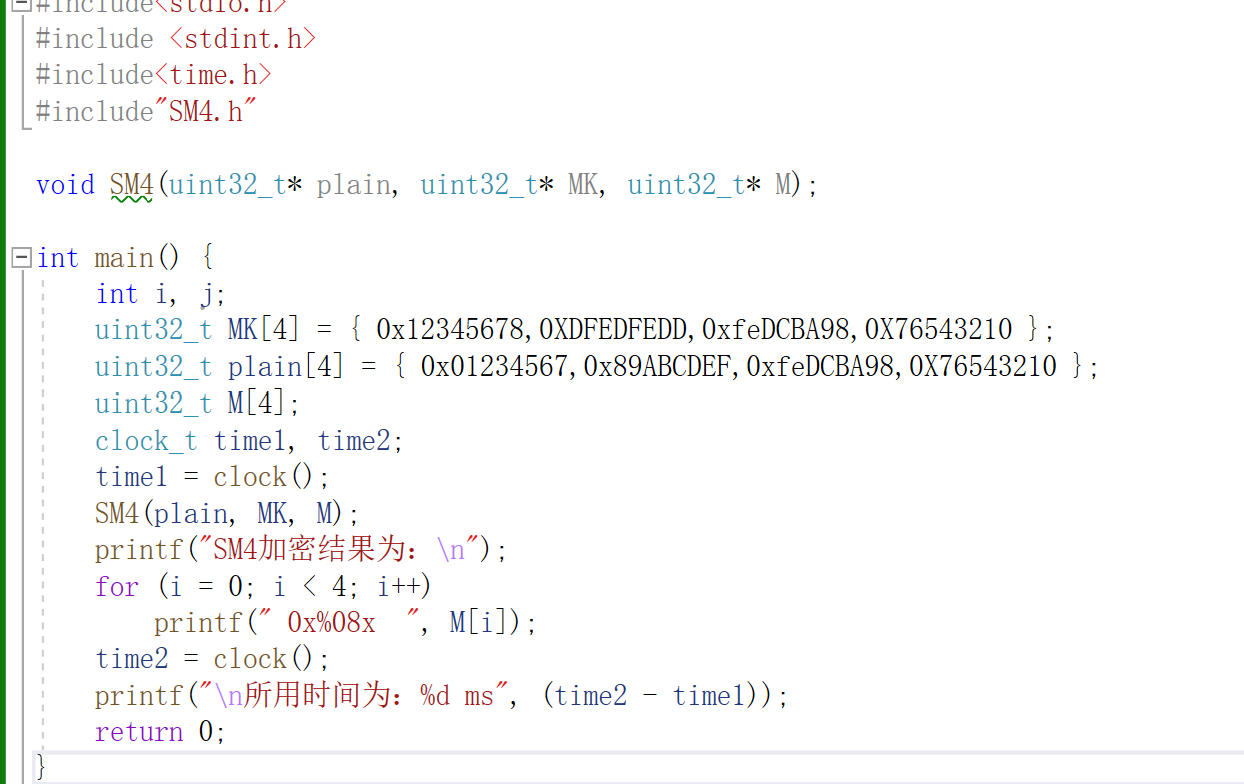
　　基本密码部件：SM4密码算法使用了S盒、非线性变换τ、线性变换部件L、合成变换T基本密码部件。

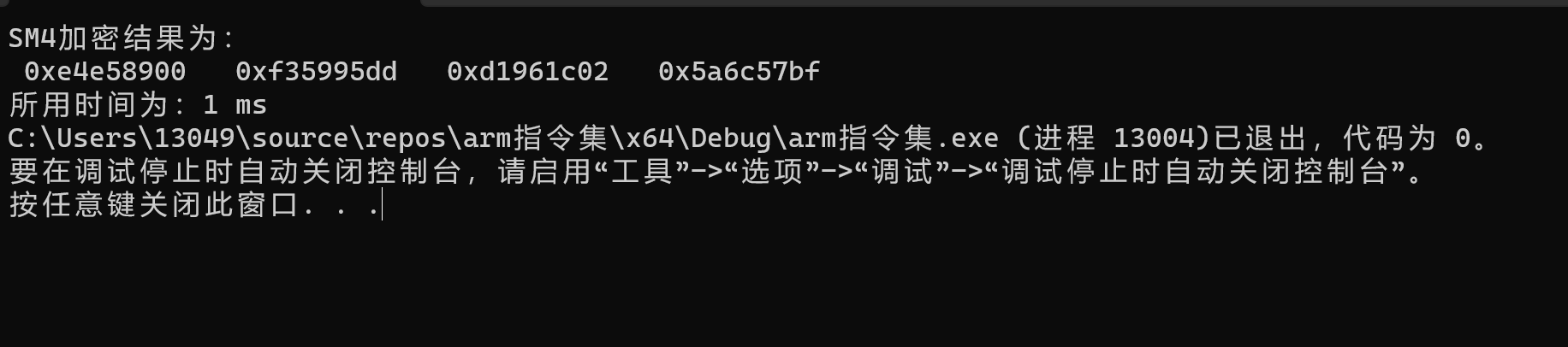
　　轮函数：SM4密码算法采用对基本轮函数进行迭代的结构。利用上述基本密码部件，便可构成轮函数。SM4密码算法的轮函数是一种以字为处理单位的密码函数。

　　加密算法：SM4密码算法是一个分组算法。数据分组长度为128比特，密钥长度为128比特。加密算法采用32轮迭代结构，每轮使用一个轮密钥。

　　解密算法：SM4密码算法是对合运算，因此解密算法与加密算法的结构相同，只是轮密铝的使用顺序相反，解密轮密钥是加密轮密钥的逆序。

密钥扩展算法：SM4密码算法使用128位的加密密钥，并采用32轮法代加密结构，每一轮加密使用一个32位的轮密钥，共使用32个轮密钥。因此需要使用密钥扩展算法，从加密密钥产生出32个轮密钥。





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

**ECDSA原理**

ECDSA是一种基于椭圆曲线的数字签名算法，用于在公开通信网络中验证数据的完整性和身份的真实性。下面是ECDSA的基本原理：

椭圆曲线的选择：首先选择一个适当的椭圆曲线作为公钥密码系统的基础。这个椭圆曲线由以下方程定义：y^2 = x^3 + ax + b，其中a和b是曲线的参数。

**密钥生成**：选择一个私钥，并使用椭圆曲线上的点乘法生成对应的公钥。私钥保密，而公钥可以公开共享。

**签名生成**：要对一段数据进行签名，需要使用私钥对数据的哈希值进行操作生成数字签名。具体过程如下：

a. 计算数据的哈希值。

b. 使用私钥对哈希值进行加密操作，生成签名。

**签名验证**：接收到签名的一方可以使用发送方的公钥验证签名的真实性和数据的完整性。具体过程如下：

a. 使用公钥解密签名，得到哈希值的伪随机数。

b. 计算接收到的数据的哈希值。

c. 对比两个哈希值是否相等，如果相等，则签名有效。

SM4算法是一种对称加密算法，而ECDSA(椭圆曲线数字签名算法)是一种用于以太坊数字签名的非对称加密算法。因此，SM4算法在以太坊与ECDSA的应用可以理解为使用SM4对ECDSA签名和验证过程中涉及的敏感数据进行加密和解密。

涉及以下步骤：

签名验证：使用deduce技术需要使用与生成数字签名所用私钥相关联的公钥。应用数学计算来检查数字签名是否与给定的消息和公钥匹配。

哈希处理：在应用deduce技术之前，通常会使用密码哈希函数（如SHA-256）对要签名的消息进行哈希处理。这样可以确保消息具有固定的长度，并增强安全性。



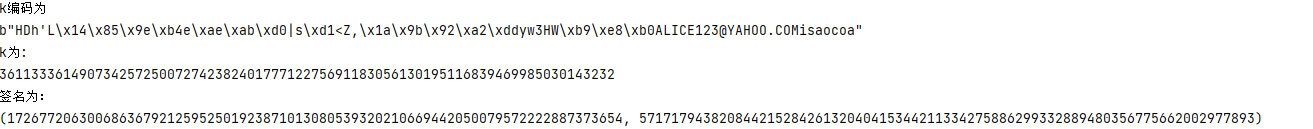
# \*Project11: impl sm2 with RFC6979

我们需要保证k随机，于是采用不可以的哈希算法，保证k不会出现相同的情况。

最后我们令k=（d||ID||"isaocoa"）其中d为私钥。我们在最后接了一个用于混淆的字符串。我们的哈希算法采用的是sm3。

结合sm2算法，我们可以定义和实现各种椭圆曲线的运算函数，最后完成该实验完成该实验

**最后可得结果**：

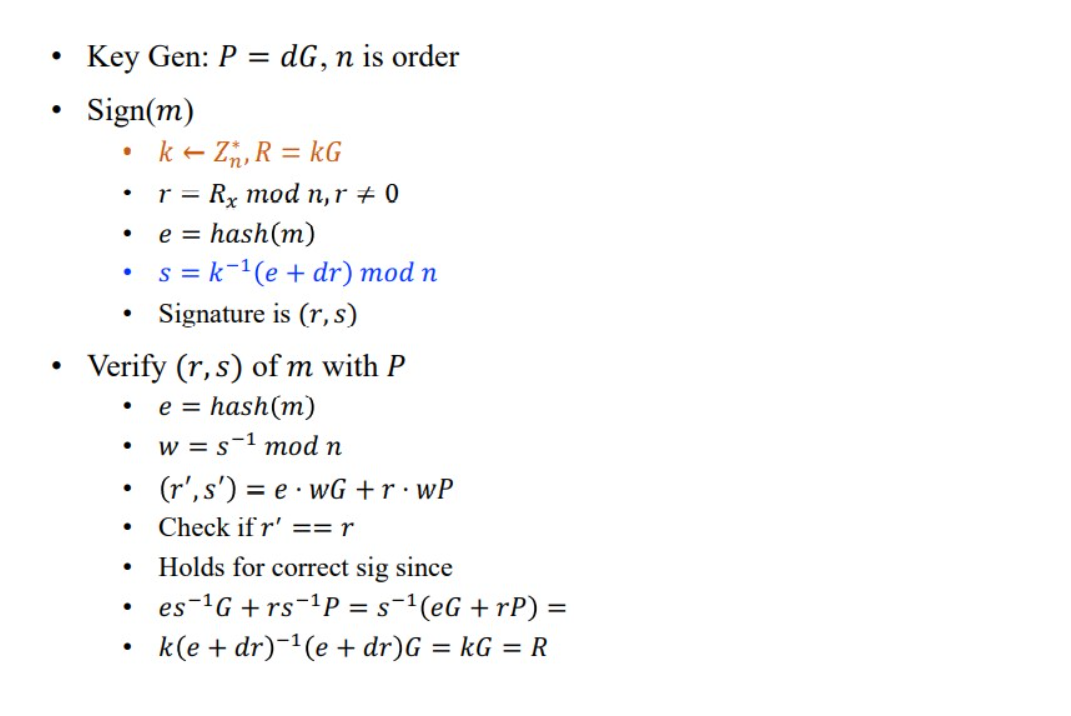


**最后可得耗时**



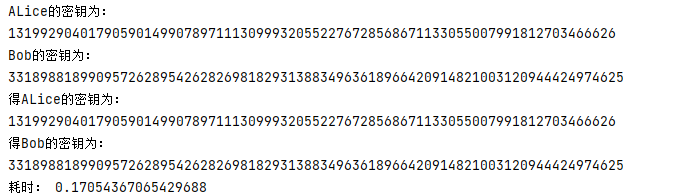
# \*Project12 verify the above pitfalls with proof-of-concept code

**实验原理**



在实现了椭圆曲线各种操作函数的基础上，不难实现该方案。

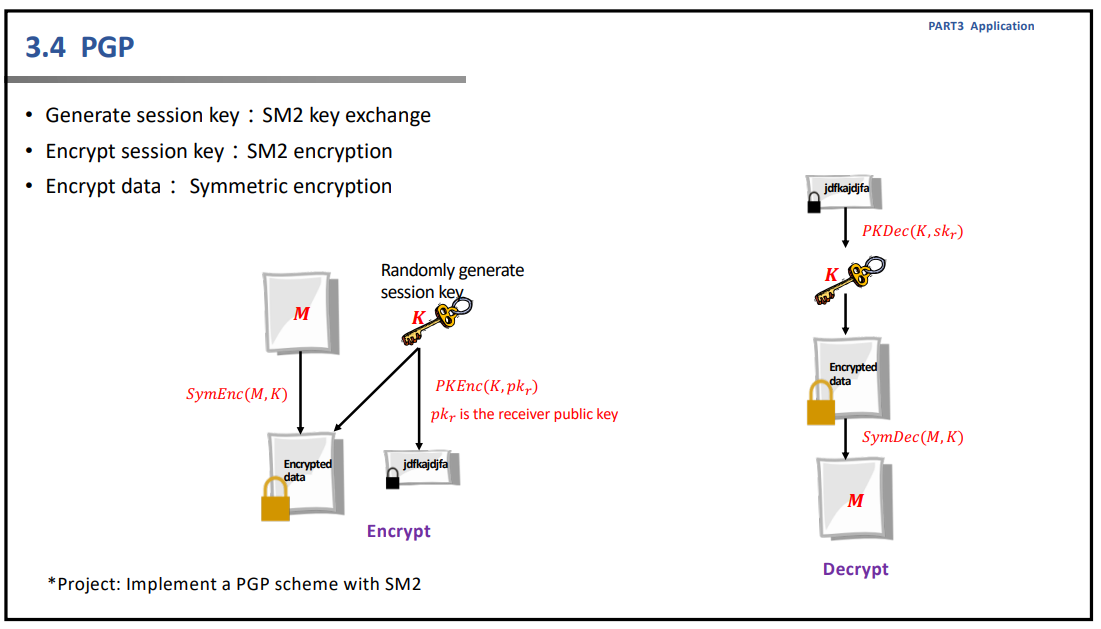
**实验结果：**

：

我们发现，最后还原出的密钥是一样的。说明还原生效。

# \*Project14: Implement a PGP scheme with SM2

根据课件知识进行实验：



按照此结构，实现PGP方案

于是在代码中可以体现出:

在加密函数encrypt中

随机产生密钥k

对data进行hash等操作后进行sm4对称加密

用sm2对k进行加密

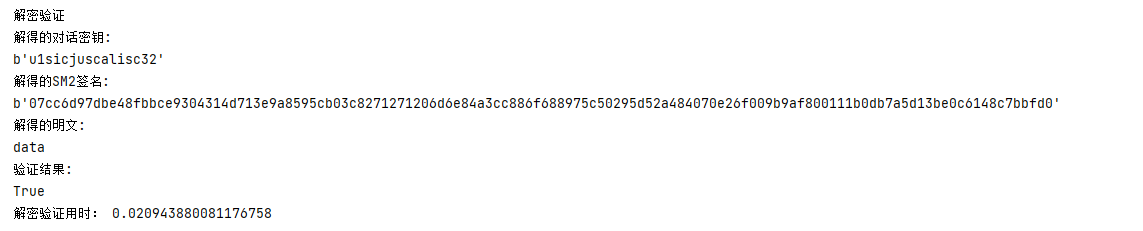
拼接后完成发送

在解密函数decrypt中：

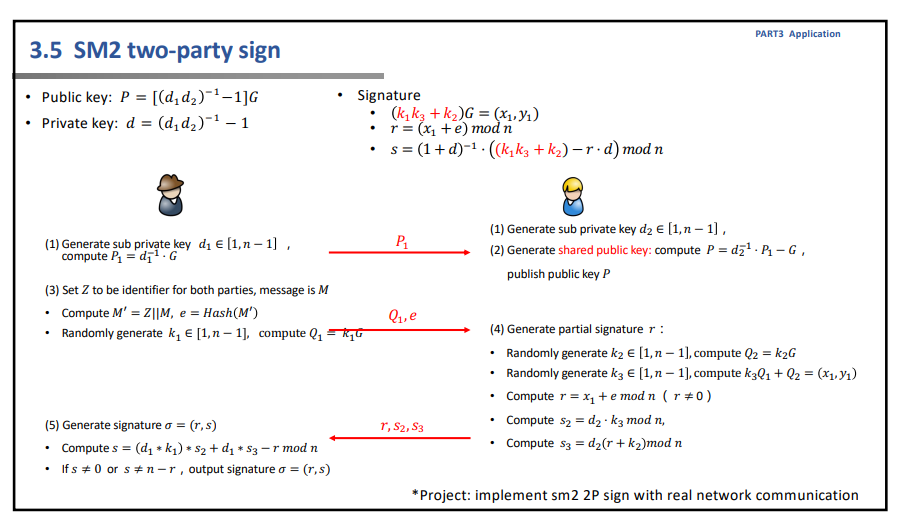
按照顺序分开解密，验证公钥即可

最后结果为：





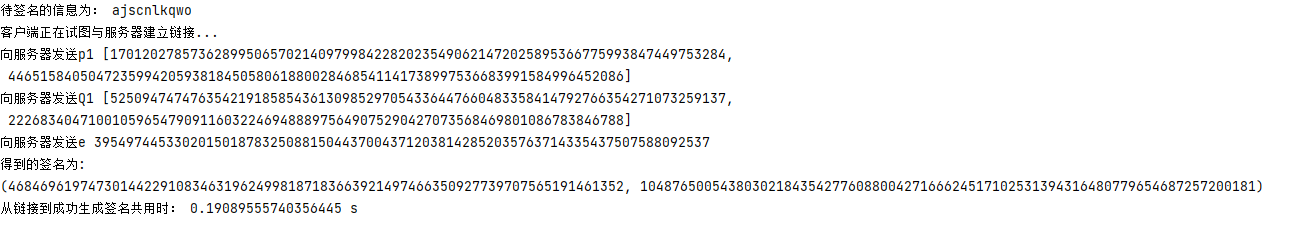
# \*Project15: implement sm2 2P sign with real network communication



这种签名方案在已经实现了椭圆曲线算法和操作的基础上是很好实现的

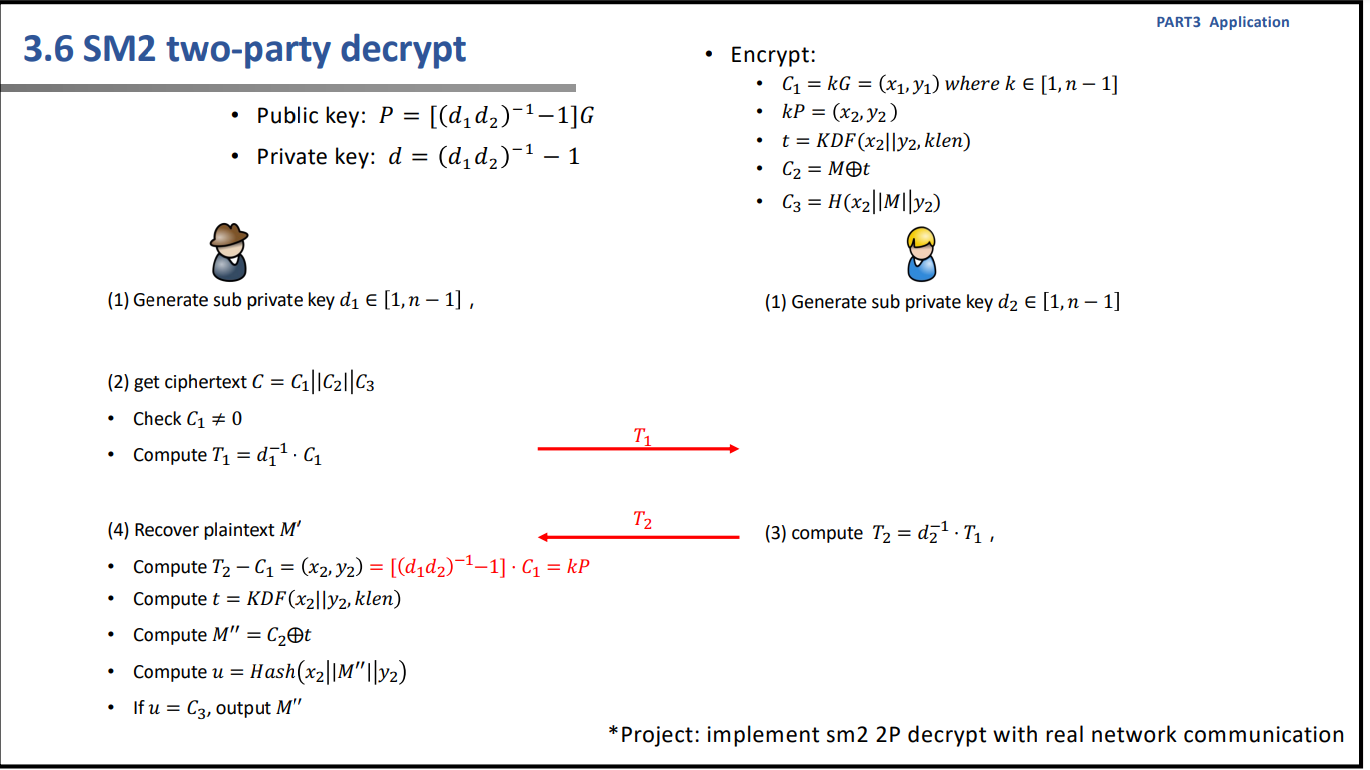
最终通过调用socket库，在本机网络上实现了真实网络通信。打开两个py文件，轮流启动两个程序便可实现交互。

最后结果为：



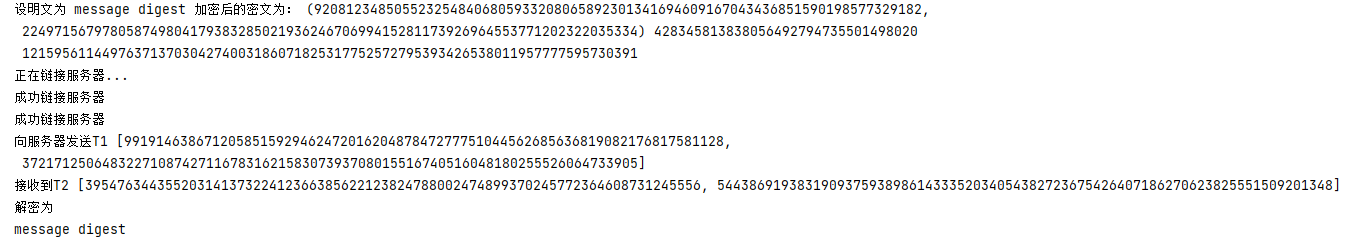
可以发现验证正确

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



按照这种方案，依据椭圆曲线运算和网络通信的原理在真实网络实现sm2解密方案

实验结果为：



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

### Firefox插件实现：

> ### 1,创建插件清单文件（manifest.json）：

>

> \* 在插件项目目录中创建一个名为 `manifest.json` 的文件。该清单文件描述了插件的名称、版本、权限等信息。

>

> \* 下面是一个示例的 `manifest.json` 文件内容：

>

> \* ```python

> - {

> -   "manifest\_version": 2,

> -   "name": "My Password Manager",

> -   "version": "1.0",

> -   "description": "A password manager extension for Firefox.",

> -   "permissions": ["storage", "<all\_urls>"],

> -   "browser\_action": {

> -     "default\_popup": "popup.html"

> -   },

> -   "icons": {

> -     "48": "icon.png"

> -   }

> - }

> ```

> ### 2,创建弹出窗口页面

>

> \* 在插件项目目录中创建一个名为 `popup.html` 的文件。该文件定义了插件弹出窗口的外观和交互。

>

> \*下面是一个示例的 `popup.html` 文件内容：

> <!DOCTYPE html>

<html>

<head>

<title>Password Manager</title>

<script src="popup.js"></script>

</head>

<body>

<h1>Password Manager</h1>

<form id="login-form">

<label for="username">Username:</label>

<input type="text" name="username" required><br>

<label for="password">Password:</label>

<input type="password" name="password" required><br>

<input type="submit" value="Login">

</form>

</body>

</html>

> ### 3,创建脚本文件

>

> \* 在插件项目目录中创建一个名为 `popup.js` 的文件。该文件包含与插件交互的 JavaScript 代码。

>

> \* 下面是一个示例的 `popup.js` 文件内容：

>

> // 当popup.html加载完毕时，获取已保存的密码并填充到表单中

document.addEventListener('DOMContentLoaded', function() {

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

.then(tabs => {

const currentTab = tabs[0];

const origin = new URL(currentTab.url).origin;

browser.passwords.search({ origin: origin })

.then(credentials => {

if (credentials.length > 0) {

const { login, password } = credentials[0];

document.querySelector('input[name="username"]').value = login;

document.querySelector('input[name="password"]').value = password;

}

});

});

});> ```

> ### 4,打包插件

>

> \* 将插件项目目录打包成zip格式，确保清单文件和其他文件位于压缩包的根目录。

> ### 5,在Firefox中加载插件

>

> \* 打开 Firefox 浏览器，访问 `about:debugging` 页面，在左侧导航栏选择 "This Firefox"，然后点击 "Load Temporary Add-on" 按钮，选择刚才打包的压缩包。

### Google chrome记住密码插件

### 1，chrome密码插件实现简化

> ### 1,创建插件文件夹

> // 监听表单的提交事件

chrome.tabs.onUpdated.addListener(function(tabId, changeInfo, tab) {

if (changeInfo.status === 'complete') {

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

const currentTab = tabs[0];

const origin = new URL(currentTab.url).origin;

// 获取已保存的密码并填充到表单中

chrome.storage.sync.get(origin, function(data) {

if (data && data[origin]) {

const { username, password } = data[origin];

chrome.tabs.executeScript(tabId, {

code: `

document.querySelector('input[name="username"]').value = '${username}';

document.querySelector('input[name="password"]').value = '${password}';

`

});

}

});

});

}

});

// 监听表单的提交事件

chrome.runtime.onMessage.addListener(function(request, sender, sendResponse) {

if (request.action === 'store\_password') {

const origin = new URL(sender.tab.url).origin;

const { username, password } = request;

// 存储密码

chrome.storage.sync.set({ [origin]: { username, password } }, function() {

sendResponse({ message: 'Password stored successfully' });

});

return true;

}

});

> \* 在计算机上创建一个新的文件夹，并为插件选择一个唯一的名称。

> ### 2,创建清单文件

>

> \* 在插件文件夹中创建一个名为 `manifest.json` 的文件，并添加以下基本配置信息：

> {

"manifest\_version": 2,

"name": "My Password Manager",

"version": "1.0",

"description": "An example Google Chrome password manager extension",

"permissions": ["tabs", "storage"],

"background": {

"scripts": ["background.js"],

"persistent": false

},

"browser\_action": {

"default\_popup": "popup.html"

}

}> ### 3,创建内容脚本

popup.html文件：

htmlCopy Code

<!DOCTYPE html>

<html>

<head>

<title>Password Manager</title>

<script src="popup.js"></script>

</head>

<body>

<h1>Password Manager</h1>

<form id="login-form">

<label for="username">Username:</label>

<input type="text" name="username" required><br>

<label for="password">Password:</label>

<input type="password" name="password" required><br>

<input type="button" value="Login" id="login-button">

</form>

</body>

</html>

popup.js文件：

javascriptCopy Code

// 监听登录按钮的点击事件

document.getElementById('login-button').addEventListener('click', function() {

// 获取表单字段的值

const username = document.querySelector('input[name="username"]').value;

const password = document.querySelector('input[name="password"]').value;

// 通过消息传递将密码发送给background脚本进行存储

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

const activeTab = tabs[0];

chrome.runtime.sendMessage({

action: 'store\_password',

username: username,

password: password

}, function(response) {

console.log(response.message);

});

});

});

## 主要区别：

存储位置：Firefox使用基于本地浏览器的存储方式来保存密码，而谷歌浏览器使用Google账户来保存密码。

同步功能：Firefox采用本地存储，因此它不提供跨设备的密码同步功能。谷歌浏览器可以通过Google账户自动同步密码，使得用户可以在不同设备上访问已保存的密码。意味着在使用Firefox时，密码仅存储在本地计算机上，而在使用谷歌浏览器时，密码会与Google账户同步到云端。

安全性：Firefox和谷歌浏览器都使用了加密的方式来保存密码，以保证用户数据的安全性。然而，谷歌浏览器的密码同步功能涉及到云端存储和传输，在安全性方面可能存在一定风险。

插件支持：Firefox和谷歌浏览器都支持第三方开发的记住密码插件。这些插件可以增强密码管理功能，并提供额外的安全选项、自定义设置等

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

#### 实验内容

先下载[electrum]([Electrum Bitcoin Wallet](https://electrum.org/" \l "download)) 比特币钱包，然后通过注册收获了30个比特币地址以及私钥。

#### 比特币交易流程：

创建钱包：首先，您需要选择一个合适的数字钱包来存储您的比特币。可以选择在线钱包、硬件钱包或者软件钱包。创建钱包后，会生成一个独特的地址用于接收和发送比特币。

获取比特币：要获取比特币，您可以通过购买、挖掘或接受比特币作为支付。购买比特币可以在交易所进行，挖掘则需要算力和硬件设备，接受比特币可以通过与他人进行交易。

发起交易：当您想要将比特币发送给他人时，需要知道对方的比特币地址。在您的钱包中选择发送选项，并输入收款人的比特币地址和要发送的数量。

确认交易：在发送比特币之前，您需要确认交易细节，例如发送数量和收款地址是否正确。此外，还需要支付相应的交易费用，以确保交易被快速验证并添加到区块链中。

交易广播和确认：完成交易确认后，您的钱包会将交易广播到比特币网络中的节点。然后，矿工会将您的交易包含在新的区块中，并开始进行工作量证明的计算。一旦有足够的矿工验证并确认您的交易，它就会被添加到区块链中。

交易完成：一旦您的交易被添加到区块链中，您和对方就可以在区块链上查看交易记录。此时，比特币所有权已从您转移到了接收人手中，交易即完成。

可以在 <https://live.blockcypher.com/btc-testnet> 中查看详细的交易。

但之后没有使用上面的比特币钱包软件，而是获得了比特币测试地址

https://www.bitaddress.org)//登录后还需要在网址后加上?testnet=true可获取测试地址

<https://live.blockcypher.com/btc-testnet/tx/bdd6a952621cc7653c43ebd6eb8e4d1e6b8b233d820e87aec2db8174675c291f/> 看到取得测试地址后可以在水龙头领取测试的比特币之后便会有一笔交易，打入比特币进这个比特币地址。

在[比特币测试网交易查询]([BlockCypher Testnet Block Explorer |BlockCypher](https://live.blockcypher.com/bcy/)) 网址里面可以查询比特币地址，以及通过txid（交易id，唯一标识某笔交易的交易号）也可以找到某笔交易。

通过查询我的比特币地址：可以看到上面的余额情况：

再通过查询一笔交易的txid可以查到该交易。

交易哈希（Transaction Hash）：每笔比特币交易都有一个唯一的交易哈希，用于标识该交易。通过交易哈希可以在区块链上查找到该交易的详细信息。

输入（Inputs）：交易的输入是之前交易的输出。输入包含了之前某些比特币交易的输出信息，即发送方的地址和发送金额。可以通过解析输入找到发送方的地址和发送的比特币数量。

输出（Outputs）：交易的输出是指交易接收方的地址和接收的金额。可以通过解析输出找到接收方的地址和接收的比特币数量。

手续费（Fee）：交易手续费是发送方支付给矿工的费用，以便将交易添加到区块链中。通过解析交易数据，可以确定交易手续费的大小。

签名（Signatures）：交易数据还包含用于验证交易有效性的数字签名。签名是发送方使用其私钥对交易进行加密的结果，接收方使用发送方的公钥对签名进行解密和验证。

比如我们查询刚才进行的把比特币打入账户的交易，查到这笔交易的

我们的任务是爬取该部分信息并且解析出来，所以可以把网址交给自己写的爬虫程序，它从网址上爬取之后再解析出文本格式。

爬虫程序为parse.py，部分如下：

# 指定目标网址

url'https://live.blockcypher.com/btc-testnet/tx/f96ba3e152b89a7318f88566d192ef37fcd44dd25b5305b5e1dedcd6b2c7861e/'

# 发送GET请求获取网页内容

response = requests.get(url)

html\_content = response.text

# 使用BeautifulSoup解析HTML

soup = BeautifulSoup(html\_content, 'html.parser')

# 获取网页文本内容（去除HTML标签）

text\_content = soup.get\_text()

# 将文本内容保存到.txt文件中

with open('output after parse.txt', 'w', encoding='utf-8') as file:

file.write(text\_content)

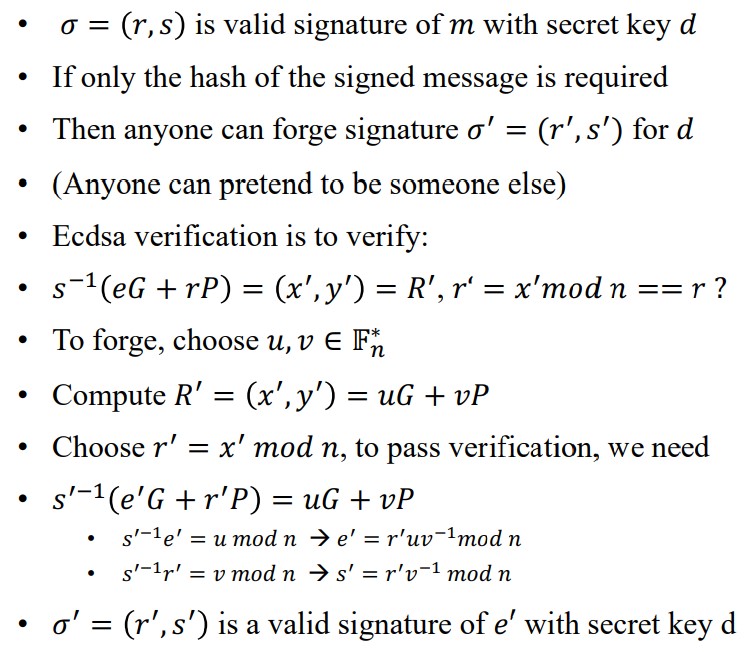
即该笔交易的信息

## Project18-send a tx on Bitcoin testnet, and parse the tx data down to every bit

# Project-19 forge a signature to pretend that you are Satoshi

## 实验内容

通过中本聪的公钥，在没有密钥情况下伪造中本聪签名并通过验证



首先可以查到中本聪的公钥和签名：

P\_Satoshi=(26877259512020005462763638353364532382639391845761963173968516804546337027093,48566944205781153898153509065115980357578581414964392335433501542694784316391)

sig\_Satoshi=(41159732757593917641705955129814776632782548295209210156195240041086117167123, 57859546964026281203981084782644312411948733933855404654835874846733002636486)

```

### 伪造签名的原理如下：

只需要中本聪的公钥便可伪造其签名（在验签过程中不重新计算e，而是直接使用给出的e进行验证）

验签过程如下：

from base import \*

import random

if \_\_name\_\_ == "\_\_main\_\_":

prikey, pubkey = Key\_Generation()

print("生成的 Satoshi 的公钥为:", pubkey)

u = random.randint(1, N - 1)

v = random.randint(1, N - 1)

R = add\_point(mul\_point(u, G), mul\_point(v, pubkey), P)

r = R[0] % N

s = (r \* calc\_inverse(v, N)) % N

sign = (r, s)

print("生成的签名为:", sign[0])

e = (r \* u \* calc\_inverse(v, N)) % N

r\_ = (mul\_point(calc\_inverse(s, N), add\_point(mul\_point(e, G), mul\_point(r, pubkey), P)))[0] % N

print("伪造的签名问:", r\_)

if r\_ == r:

print("签名伪造成功！")

else:

print("伪造失败！")```

额外传入了参数e，直接用e进行验证，而不是传入消息m，用消息m重新计算e再验证。

### 2，运行指导

直接运行代码即可，能够输出伪造的签名，以及该伪造签名是否通过验证。

### 3，运行结果

运行程序结果如下：

675b90ff3ec4cac4e1bf73b9f0eaf44

#### \*Project22 research report on MPT

##### 一．MPT概念

MPT树（Merkle Patricia Tree）是一种基于前缀树和Merkle树的数据结构，是以太坊（Ethereum）区块链中的关键数据结构之一。它被用来存储和验证区块链上的状态（包括账户、合约和存储的数据）。

MPT树的设计灵感来源于Patricia Trie（也称为Radix Trie），它是一种高效的前缀树数据结构，用于快速存储和检索键值对。与传统的哈希表不同，MPT树可以支持按照键的前缀进行范围查询和证明。

MPT树的核心思想是使用键的哈希值作为叶子节点，将键值对保存在叶子节点上。每个内部节点都包含一个分支和一个扩展，分支存储了一系列键的共同前缀，扩展则指向下一个节点。通过递归地将键拆分为前缀和后缀，MPT树可以有效地组织和管理大量的键值对。

MPT树的另一个重要特性是其使用了Merkle树的概念。Merkle树是一种哈希树，可以对整个数据集进行完整性和一致性验证。MPT树使用Merkle树的哈希值作为每个节点的标识，并通过验证哈希值来确保树的完整性。

在以太坊中，MPT树被广泛应用于存储区块链状态（包括账户、合约和存储的数据），并且被用来生成区块的状态根哈希。通过MPT树，以太坊可以高效地存储和验证区块链上的状态，并支持高度可扩展的智能合约和去中心化应用。

##### 二．模型

##### Merkle树

Merkle Tree，通常也被称作Hash Tree，顾名思义，就是存储hash值的一棵树。Merkle树的叶子是数据块(例如，文件或者文件的集合)的hash值。非叶节点是其对应子节点串联字符串的hash。

梅克尔树就是最经典的解决数据校验的一种方式，用每个节点的hash值来建立对应的关系，底层的叶子节点都算一个hash，这是一个二叉树，两两hash之间再算一次hash，不断往上计算得出top hash算作一个根节点存到区块里面，去校验的时候，如果叶子节点发生改动，按照规则两两一hash计算得出的根节点会不一样，就知道数据发生了变动。

![image](https://github.com/lemonade310/homework-group25/assets/139195261/63eee6f6-a70c-421b-aa9b-69462fc09a8d)

梅克尔树可以实现数据校验，防止篡改。以太坊要去做hash的是整个要存储内容的RLP编码，所以以太坊相当于把自己的value先做RLP编码，然后再去求hash，然后把最后得到的hash值作为在数据库中存储的位置，所以在MPT中的节点里面用hash作为key，访问的的时候根据hash在数据库中找到对应的值。

##### MPT（Merkle Patricia Tree）树

MPT（Merkle Patricia Tree）就是Merkle Tree和Patricia Tree这两者混合后的产物。

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。

##### 三．基本结构

**节点（Node）：**MPT树由不同类型的节点组成，包括扩展节点（Extension）、分支节点（Branch）、叶子节点（Leaf）和空节点（Null）。

**扩展节点（Extension Node）**：扩展节点用于存储键的共同前缀，它包含一个扩展值和一个指向下一个节点的引用。

**分支节点（Branch Node）**：分支节点用于存储键的分支，它有16个槽位，每个槽位可以存储一个子节点的引用。分支节点可以通过键的第一个字节的值来定位相应的子节点。

**叶子节点（Leaf Node）：**叶子节点用于存储键值对，它包含一个键的哈希值和对应的值。

**空节点（Null Node）：**空节点表示结束或空白的部分，它在MPT树中用于表示一个键的结束或者一个空的子节点。

MPT树的根节点是一个分支节点，它通过哈希值来标识整个树的完整性。从根节点开始，通过递归地查找键的前缀和后缀，可以找到叶子节点，并获取对应的值。MPT树的节点之间通过引用来连接，形成一个树状的数据结构。

MPT树的设计通过共享相同前缀和使用哈希值来压缩和优化存储空间，避免了数据冗余。同时，MPT树还采用了Merkle树的特性，可以对整个数据集进行验证，并提供了高度可扩展的存储和验证能力。

##### 四，操作

**合并操作**

首先判断当前节点是否只有一个子节点且没有值。

如果满足条件，找到唯一的子节点，并判断该子节点是否也满足合并条件（即只有一个子节点且没有值）。

如果子节点满足条件，将子节点的子节点和值合并到当前节点中，并删除子节点。

如果子节点不满足条件，递归地调用merge\_node方法，将合并操作应用于子节点。

**分裂操作**

创建两个新的子节点：new\_node和existing\_node（旧节点）。

去除共享前缀，得到旧节点和新键的剩余部分。

将旧节点插入到新节点中，并将新节点设置为当前节点的子节点。

更新旧节点的值或递归地将旧节点的剩余部分插入到旧节点中。

将新键插入到新节点中。

**Get操作**

将需要查找Key转成Hex编码，得到搜索路径，从根节点开始搜寻与搜索路径内容一致的路径；

1. 若当前节点为叶子节点，存储的内容是数据项的内容，且搜索路径的内容与叶子节点的key一致，则表示找到该节点；反之则表示该节点在树中不存在。

2. 若当前节点为扩展节点，且存储的内容是哈希索引，则利用哈希索引从数据库中加载该节点，再将搜索路径作为参数，对新解析出来的节点递归地调用查找函数。

2. 若当前节点为扩展节点，存储的内容是另外一个节点的引用，且当前节点的key是搜索路径的前缀，则将搜索路径减去当前节点的key，将剩余的搜索路径作为参数，对其子节点递归地调用查找函数；若当前节点的key不是搜索路径的前缀，表示该节点在树中不存在。

3. 若当前节点为分支节点，若搜索路径为空，则返回分支节点的存储内容；反之利用搜索路径的第一个字节选择分支节点的孩子节点，将剩余的搜索路径作为参数递归地调用查找函数。

**Insert操作**

1. 首先找到与新插入节点拥有最长相同路径前缀的节点，记为Node；

2. 若该Node为分支节点：

1. 剩余的搜索路径不为空，则将新节点作为一个叶子节点插入到对应的孩子列表中；

2. 剩余的搜索路径为空（完全匹配），则将新节点的内容存储在分支节点的第17个孩子节点项中（Value）；

3. 若该节点为叶子／扩展节点：

1. 剩余的搜索路径与当前节点的key一致，则把当前节点Val更新即可；

2. 剩余的搜索路径与当前节点的key不完全一致，则将叶子／扩展节点的孩子节点替换成分支节点，将新节点与当前节点key的共同前缀作为当前节点的key，将新节点与当前节点的孩子节点作为两个孩子插入到分支节点的孩子列表中，同时当前节点转换成了一个扩展节点（若新节点与当前节点没有共同前缀，则直接用生成的分支节点替换当前节点）；

4. 若插入成功，则将被修改节点的dirty标志置为true，hash标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在。

**Delete操作**

1. 找到与需要插入的节点拥有最长相同路径前缀的节点，记为Node；

2. 若Node为叶子／扩展节点：

1. 若剩余的搜索路径与node的Key完全一致，则将整个node删除；

2. 若剩余的搜索路径与node的key不匹配，则表示需要删除的节点不存于树中，删除失败；

3. 若node的key是剩余搜索路径的前缀，则对该节点的Val做递归的删除调用；

3. 若Node为分支节点：

1. 删除孩子列表中相应下标标志的节点；

2. 删除结束，若Node的孩子个数只剩下一个，那么将分支节点替换成一个叶子／扩展节点；

4. 若删除成功，则将被修改节点的dirty标志置为true，hash标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在。

**五 安全性分析：**

完整性保证：MPT使用节点的哈希值来标识和验证树的完整性。每个节点的哈希值都取决于它的内容，包括键、值以及子节点的哈希值。通过根节点的哈希值，可以验证整个树的完整性。即使树中的一个节点被篡改，它的哈希值将与实际内容不匹配，从而破坏树的完整性。

非可逆性：MPT使用哈希函数进行节点的哈希计算，哈希函数具有非可逆性的特点。这意味着无法通过节点的哈希值逆推出原始的键或值。这种非可逆性提供了保护数据隐私和安全性的重要保障。

防篡改性：MPT的设计通过共享前缀和使用哈希值来压缩存储空间，并减少数据冗余。这使得树中的每个节点都依赖于其父节点和相邻节点的哈希值。如果有人试图篡改或删除树中的某个节点，将会破坏整个树的完整性。因此，MPT提供了一种防篡改的机制。

高效性：MPT通过共享前缀和使用哈希值来优化存储和查询性能。相同前缀的键被合并为一个扩展节点，减少了存储空间的需求。同时，使用哈希值作为节点的标识，可以快速定位和验证节点的内容。这使得MPT具有高效的存储、验证和查询能力。