

## 

## **网络空间安全创新创业实践**



****

创新创业实践

***Project1-22汇总报告***

**姓名： 赵玉玺**

**学号： 202100460099**

**学院： 网络空间安全学院**

**班级： 21级本科网安二班**

**2023年07月22日**

2022

**访**

**报告说明：**

**\*\*\*个人实验，未参与小组，实验均在个人电脑上实现，为AMD Ryzen 7 5800H\*\*\***

**\*\*\*project中可能有多个python文件，其中project X.py为主文件，其余文件是供project引用或导入的文件\*\*\***

**\*\*\*已完成project1、2、3、4、5、6、8、10、11、12、13、14、15、16、17、19、22\*\*\***

**\*\*\*未完成project7、9、18、20、21\*\*\***

**项目报告：**

**·Project1: implement the naive birthday attack of reduced SM3**

**①代码解释：**

在github提交的project1.py中，代码实现了简化的md5生日攻击。它主要通过生成随机的消息，并计算其哈希值，然后将哈希值和消息存储在一个哈希表中。如果发现两个不同的消息具有相同的哈希值（碰撞），则返回这两个消息作为结果。

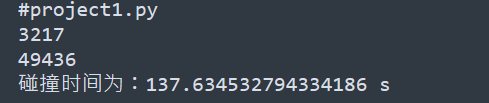
函数**generate\_message()**用于生成随机的消息，其长度为32字节。每个字符都是从0到255的随机整数转换而来。

函数**birthday\_attack()**实施生日攻击。它使用一个**哈希表hash\_table**来存储已经计算过的哈希值和消息。在每次生成新的消息后，它会计算哈希值，并检查这个哈希值是否已经在哈希表中。如果是，则找到了碰撞，返回当前的消息和与之碰撞的消息；否则，将当前消息和该哈希值存储在哈希表中。

**measure\_time()**函数用于测量代码运行时间。它接受一个函数作为参数，并在函数执行前后记录时间，并输出运行时间。

最后，在主程序中调用**measure\_time()**函数来执行生日攻击，并输出结果。

**②实验截图：**

****

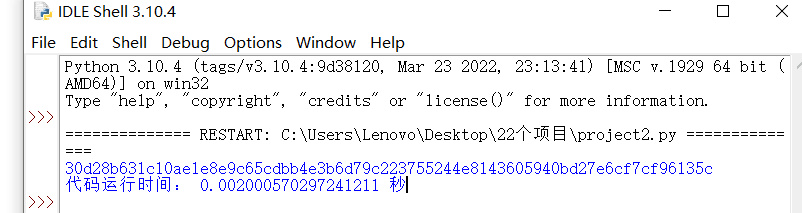
**·Project2: implement the Rho method of reduced SM3**

**①代码解释：**

**定义常量**，包括SM3算法中使用的IV（初始向量）和Tj。**实现辅助函数**，如循环左移、FF函数、GG函数等。**实现P0函数和P1函数**，用于进行密钥扩展时的异或运算。**实现CF函数**，用于进行压缩函数的迭代计算。其中，根据SM3算法规定，每个消息块分成16个字，扩展为68个字，然后计算W1。再根据W和W1计算A-H的值。**实现sm3函数**，接受消息作为输入，对消息进行填充，并使用CF函数进行迭代压缩生成哈希结果。**实现hash\_message函数**，将字节转换为十六进制字符串表示哈希结果。**实现get\_execution\_time函数**，用于计算代码运行时间。**进行测试，计算 "Hello, World!" 的SM3哈希值，并输出代码运行时间。**

代码定义了SM3哈希算法中涉及的各种操作，并通过逐步实现这些操作来计算消息的哈希值。并使用**get\_execution\_time函数来计算代码的运行时间**。

**②实验截图：**

****

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

**①代码解释：**

**此处选择了SHA-256进行拓展长度攻击。**在上传的代码中，**sha256\_length\_extension\_attack函数**接受原始消息和原始哈希值作为输入，并执行长度扩展攻击。它通过构造填充数据和长度信息来实现消息的扩展。然后，使用**hashlib.sha256()创建一个新的SHA256对象**，**并手动设置其内部状态为原始哈希值**。最后，通过调用update方法来更新哈希对象的状态，并输出攻击后的哈希值。

此外，使用**time.time()函数**来获取代码开始和结束的时间戳，并计算两者之间的差值，以得到代码执行所需的时间。

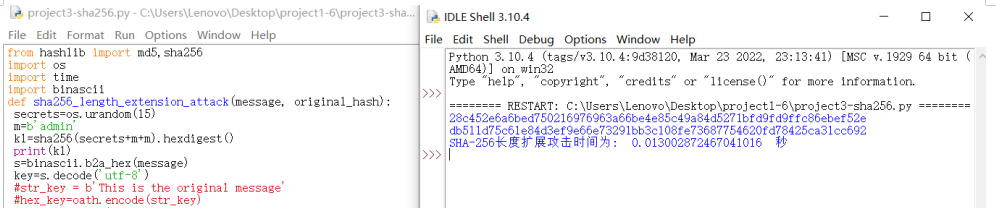
下面具体展示代码的功能和执行流程：

**导入 hashlib 和 time 模块。定义了一个函数 sha256\_length\_extension\_attack，接受两个参数：message（原始消息的字节表示）和 original\_hash（原始消息的哈希值的十六进制表示）**。

在函数内部，首先**获取原始消息的长度 original\_length**。接下来，**构造填充数据**。在 SHA256 哈希算法中，消息需要进行填充以满足特定的要求。**填充数据以 \x80 开头**，后面跟着零字节直到满足长度**对64取模的结果为56**。然后，将**原始消息的长度以大端字节序表示并添加到填充数据的末尾，得到 forged\_message**。创建一个新的 SHA256 对象，并将**原始哈希值的字节数组作为初始哈希值设置给该对象**。调用 update 方法**更新哈希对象的内部状态，传入 forged\_message作为参数**。最后，**调用 hexdigest 方法获取攻击后的哈希值，并打印出来。示例用法**部分定义了一个原始消息和其对应的哈希值，并调用了 sha256\_length\_extension\_attack 函数进行攻击。最后，**计算并打印出代码执行的时间。**

**\*\*\*对于SM3算法的代码解释：此处上传的SM3套用了SHA-256的模板，但是显然hashlib库中并不能像导入SHA-256那样直接导入SM3以及对其的各种应用（因为库中没有），所以我提交的SM3套用代码实际上是所谓的“伪代码”，但是它离运行只差一位好心人按照hsahlib库的格式将SM3算法加入进去，虽然SM3代码实现在网上并不难找（实际上project4就是我实现SM3的py代码），但是创建一个库还是比较麻烦的，所以此处对于SM3的算法我就此搁置，后面另行研究。后上网搜集资料后，实现了SM3的拓展长度攻击，并提交了代码\*\*\***

**②实验截图：**

****

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

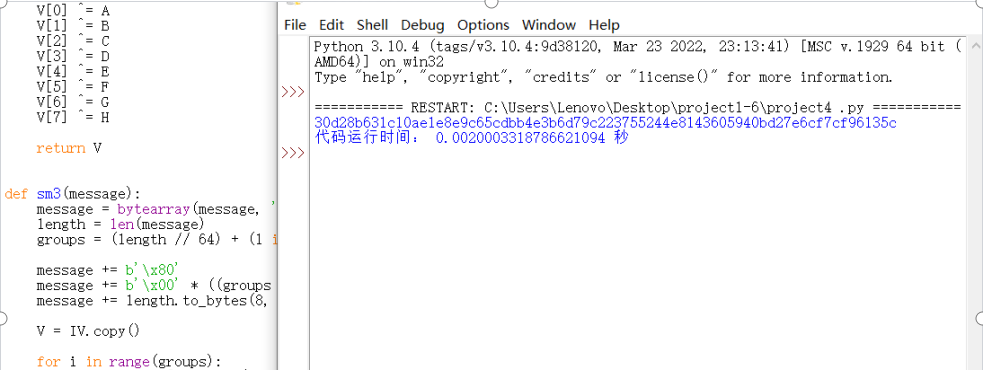
**①代码解释：**

当调用SM3()函数时，需要传入一个消息作为输入。首先，函数会**初始化初始值（IV）数组**，然后**计算消息的长度**。接下来，消息会被**填充**以满足SM3算法的要求。填充的过程是将一个“1”添加到消息末尾，然后用零字节填充，直到消息的位数满足对512取模等于448的条件。**最后，使用8个字节的消息长度将消息进行扩展。**然后，消息被**分割成一系列64字节的块**。每个块都会通过**加密函数CF()**进行处理。CF()函数实现了SM3算法的加密过程。它接受一个初始向量V和一个消息块B作为输入。首先，消息块B被分解为68个32位的字W[0:63]。其中，前16个字是从消息块中提取的。接下来，使用W数组计算W'数组，其中W'[j] = W[j] ^ W[j+4]（0 <= j < 64）。然后，使用64次循环来更新A、B、C、D、E、F、G、H的值。这些变量是过程中的中间状态。循环中的计算过程采用了**特定的置换、逻辑运算和移位操作**。循环结束后，计算得到的A、B、C、D、E、F、G、H的值将与初始向量V进行异或运算并返回。在主函数中，我们将消息分组处理后，不断更新初始向量V，直到处理完所有消息块。最后，**将得到的初始向量V转换为16进制字符串作为哈希值返回**。

**calculate\_execution\_time()**函数用于测量代码的执行时间。它记录下调用目标函数前和调用之后的时间，并计算二者之差。这样我们就可以获得目标函数的执行时间。

**\*\*\*在github中提交了project3.py与SM3.py、原SM3.py，在后续project中导入的SM3均为SM3.py（除特殊说明外）\*\*\***

**②实验截图：**

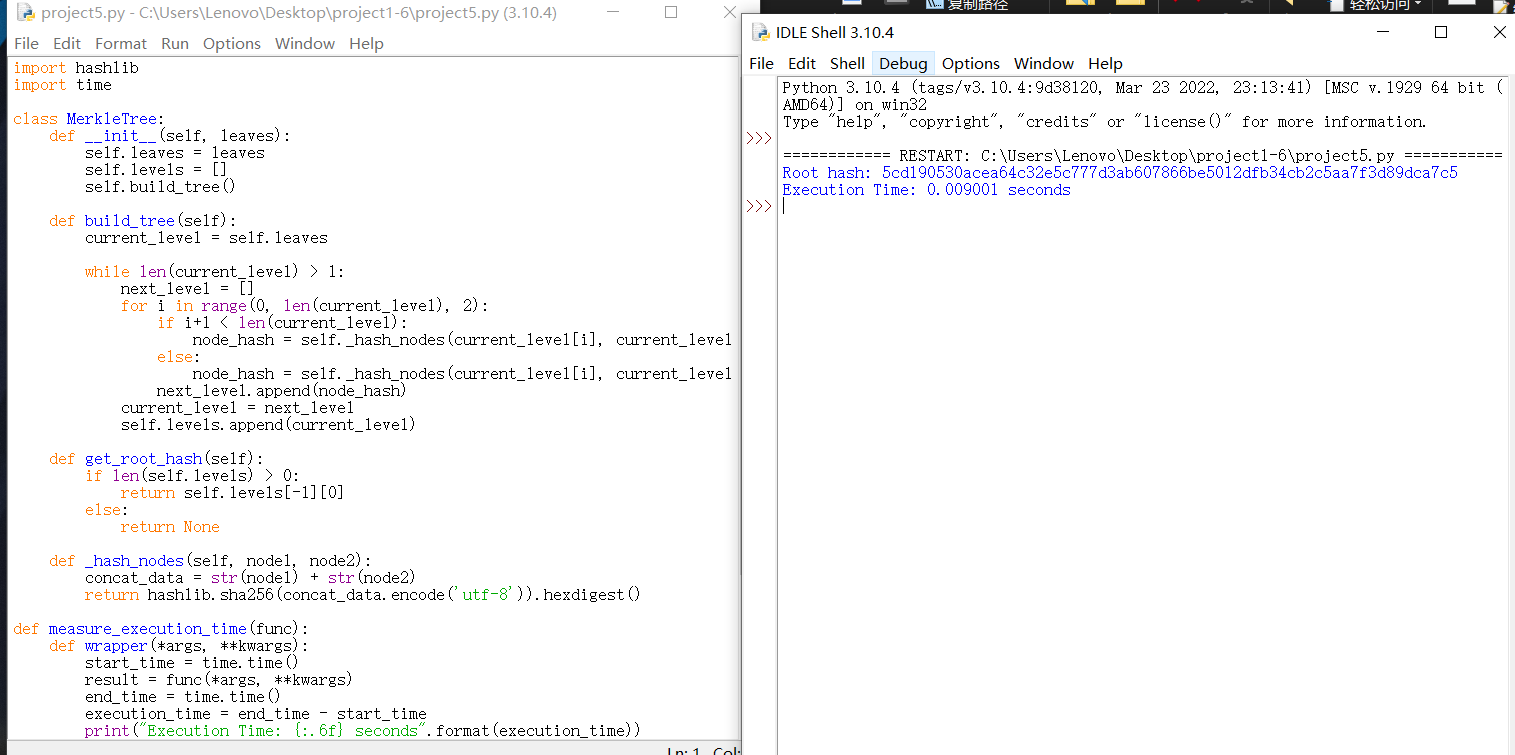
****

**·Project5: Impl Merkle Tree following RFC6962**

**①代码解释：**

在代码中，首先定义了一个 MerkleTree 类。在初始化方法中，**我们接收一个由叶子节点组成的列表作为输入，并创建一个空的 levels 列表来存储每个层级的节点。**然后，我们**调用 build\_tree** 方法来构建 Merkle 树。在 build\_tree 方法中，我们使用**一个循环来迭代计算每一层的节点**。对于每个节点对，我们使用 **\_hash\_nodes 方法来计算它们的哈希值**。如果在**当前层级中节点数量是奇数，我们复制最后一个节点来保持节点数为偶数，并继续计算哈希值。将计算得到的哈希值添加到下一层级的列表中，并将当前层级更新为下一层级，继续迭代直到只剩下一个节点作为根节点。**get\_root\_hash 方法用于获取 Merkle 树的**根哈希值**。如果**树为空，则返回 None。否则，返回 levels 列表中最后一个元素的第一个元素作为根哈希**。measure\_execution\_time函数用于**测量代码的执行时间**。最后，定义了一个名为 **test\_merkle\_tree 的测试函数**，其中包含了使用 MerkleTree 类构建 Merkle 树的代码，并通过调用 get\_root\_hash 方法获得根哈希值。

**②实验截图：**

****

**·Project6: impl this protocol with actual network communication**

**①代码解释：**

**没看懂啥意思，于是自己写了一个服务器？**

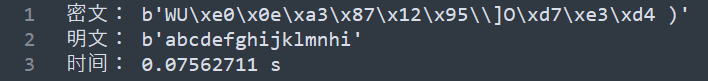
在代码中，我们首先定义了一个服务器地址和端口号（这里使用本地地址和端口8000作为示例）。然后，我们创建了一个套接字对象，并连接到服务器。send\_receive\_message 函数用于发送消息给服务器并接收响应。它将消息字符串编码为字节流后发送给服务器，然后接收服务器的响应并解码为字符串后返回。calculate\_run\_time 是一个装饰器函数，可以计算被装饰函数的运行时间。它使用 time 模块来获取开始和结束时间，并计算两者之差来获得代码运行时间。在 main 函数中使用装饰器将其应用于实际的代码逻辑。最后，在 main 函数中，我们定义了一个示例消息，并调用了 send\_receive\_message 函数来发送消息并接收服务器的响应。然后，打印出服务器的响应。

**·Project8: AES impl with ARM instruction**

**①代码解释：**

**此处直接导入了Crypto.Cipher模块（想要运行该代码请自行使用pip安装导入相应的库并为其配置环境），可以直接调用AES，并选择了ECB模式。**

**②实验截图：**



**·Project10: report on the application of this deduce technique in**

**Ethereum with ECDSA**

（参考：①https://blog.csdn.net/weixin\_43867940/article/details/130258535 ）

（参考：② https://learnblockchain.cn/article/5012 ）

**报告如下：**

**以太坊中的 ECDSA**

我们已经了解了 ECDSA 是怎么回事，接下来就看看以太坊中如何使用 ECDSA 对交易进行签名。

Secp256k1是指比特币中使用的ECDSA(椭圆曲线数字签名算法)曲线的参数，并且在高效密码学标准（Certicom Research，http://www.secg.org/sec2-v2.pdf）中进行了定义。以太坊也使用了 Secp256k1。

对以太坊一笔交易进行签名的大致步骤如下（简化后）：

1.对交易数据进行 RLP 编码

2.对第一步得到的编码进行哈希

3.将哈希与标识以太坊的特定字符串拼接在一起，再次哈希。这一步是为了保证该签名仅在以太坊上可用

4.用上一节介绍的ECDSA算法对第三步得到的哈希进行签名，得到 (r, s, v)，v 是一个字节，可以看这里对它的解释

https://bitcoin.stackexchange.com/questions/38351/ecdsa-v-r-s-what-is-v

5.将第四步得到的签名与交易数据拼接，再次进行RLP编码，得到最终的签名消息。

验证签名应该也很简单，这里不再赘述。不过这里有个点比较好奇，就是接收到交易的节点在验证签名时，如何获得签名者的公钥？可以参考这篇文章https://www.jianshu.com/p/6cb992091886

**因为可以从签名中恢复公钥，再从公钥中得到签名者地址，所以签名消息中的交易内容不需要包含发送者的地址。**

**·Project11: impl sm2 with RFC6979**

**①代码/原理解释：**

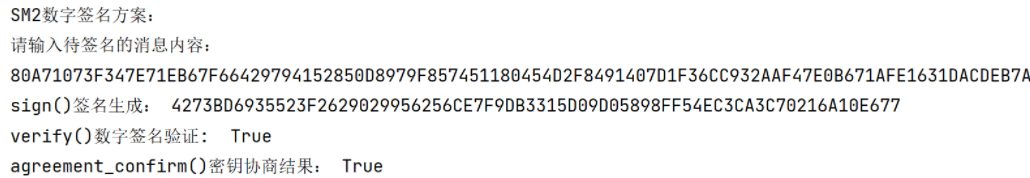
椭圆曲线数字签名算法（ECDSA）是使用椭圆曲线密码（ECC）对数字签名算法（DSA）的模拟。与普通的离散对数问题（DLP）和大数分解问题（IFP）不同，椭圆曲线离散对数问题没有亚指数时间的解决方法。因此椭圆曲线密码的单位比特强度要高于其他公钥体制。 数字签名算法（DSA）在联邦信息处理标准FIPS中有详细论述，称为数字签名标准。它的安全性基于素域上的离散对数问题。可以看作是椭圆曲线对先前离散对数问题（DLP）的密码系统的模拟，只是群元素由素域中的元素数换为有限域上的椭圆曲线上的点。椭圆曲线离散对数问题远难于离散对数问题，单位比特强度要远高于传统的离散对数系统。因此在使用较短的密钥的情况下，ECC可以达到于DL系统相同的安全级别。这带来的好处就是计算参数更小，密钥更短，运算速度更快，签名也更加短小。

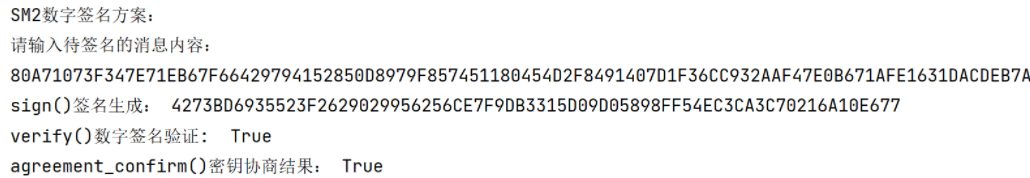
在依据RFC6979规定的标准，对于SM2方案的实现采用Python语言编写的国密工具包主要是gmssl-python库和snowland-smx-python（pysmx）库

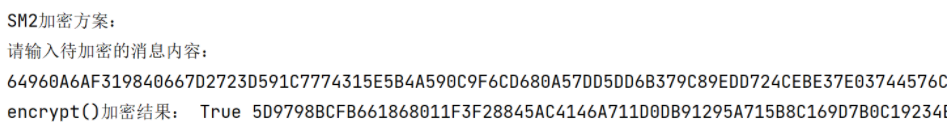
为分功能实现SM2相关的部件，所以进行的了分块的提交。首先对于SM体系下的椭圆曲线加解密体系，实现了一个ECC—class,椭圆曲线密码类（实现一般椭圆曲线的运算，不局限于SM2）。对于SM2-class，则是调用了ECC—class作为底层运算部件，根据RFC6979协议标准实现。该方案进行了基础性的ECDH正确性测试，SM2密钥协商测试，SM2数字签名与验证测试，测试过程均在key-Enc-test文件中可运行。最后main\_part运行结果，可复现RFC6979，SM2文档中的示例结果。

若要完整运行测试代码还需安装gmssl（pip install gmssl）和pysmx（pip install snowland-smx）

**②实验截图：**





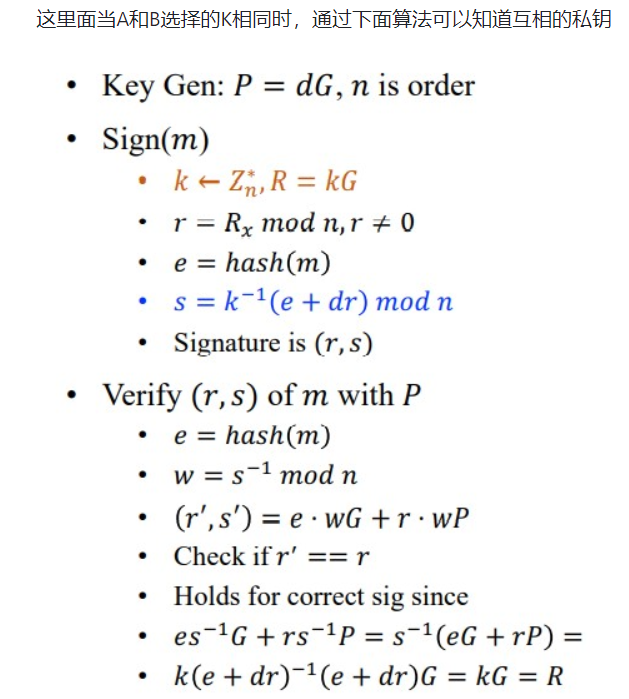




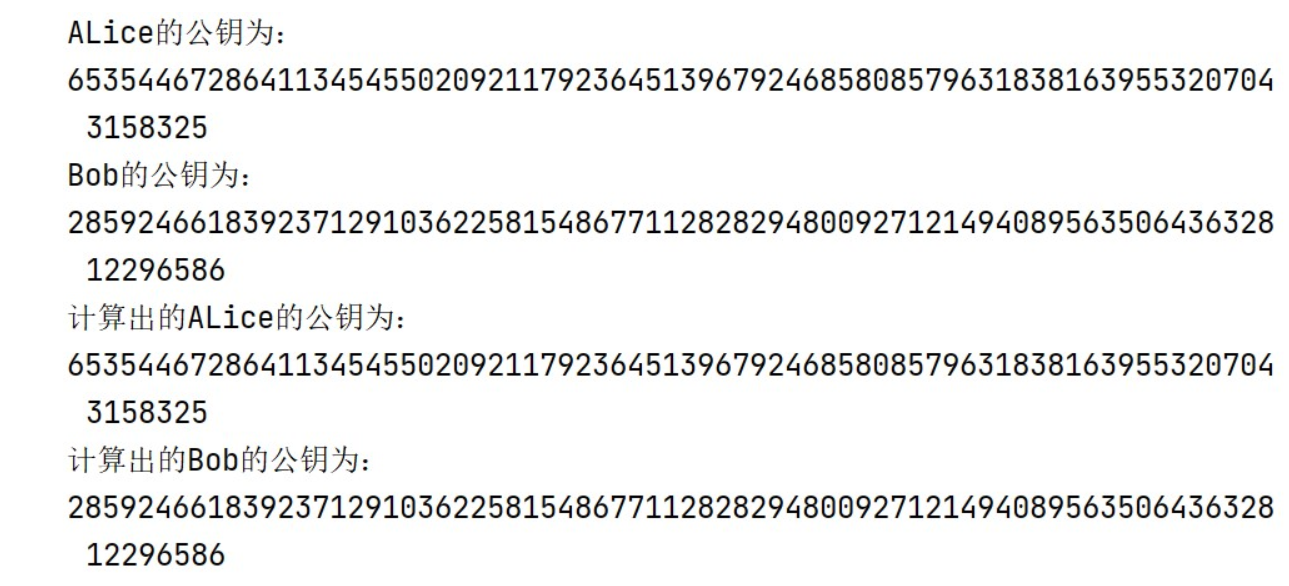
**·Project12: verify the above pitfalls with proof-of-concept code**

**①原理解释：**

**（该实验同11，导入了gmssl库，运行的话需要安装该库）**

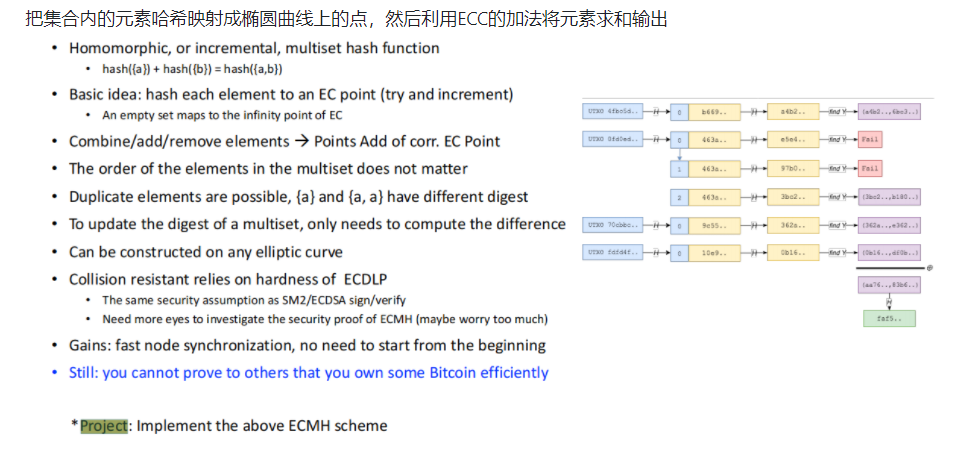


**②实验截图：**



**·Project13: Implement the above ECMH scheme**

**①原理解释：**

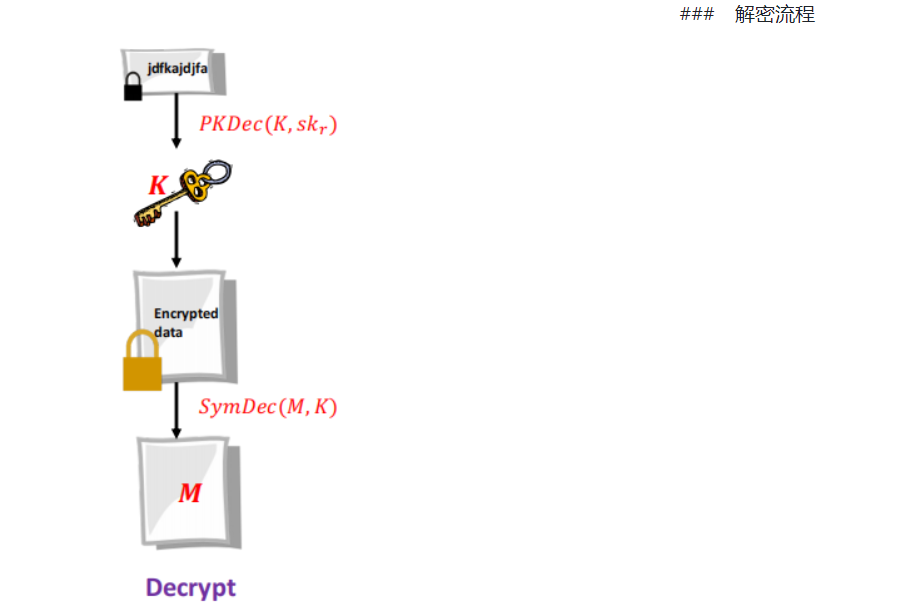
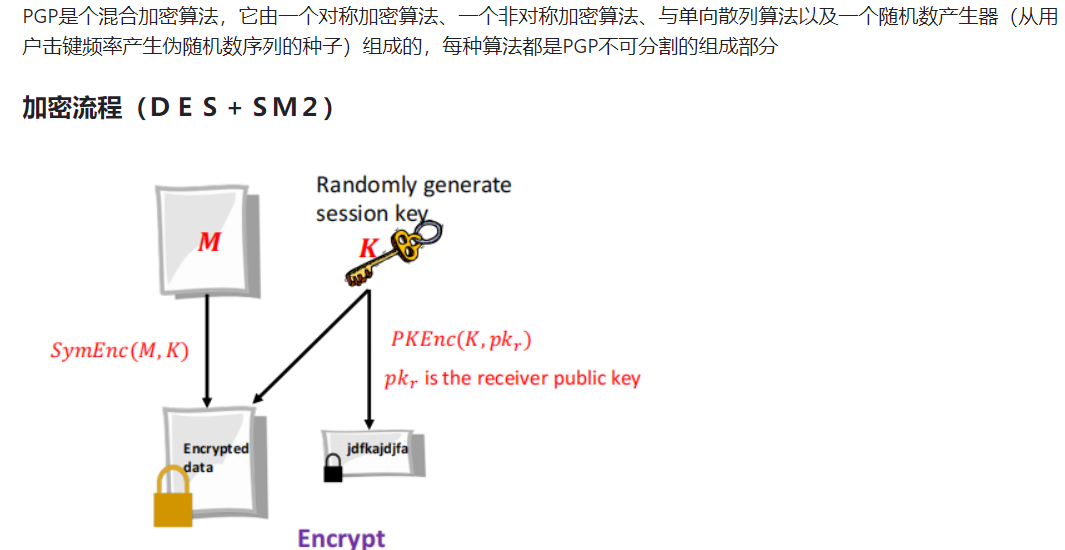
****

**②实验截图：**

****

**·Project14: Implement a PGP scheme with SM2**

**①原理与代码解释：**

****

**②实验截图：**

****

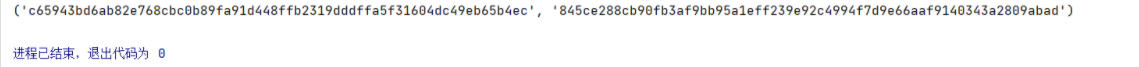
**·Project15: implement sm2 2P sign with real network**

**Communication**

**①代码解释：**

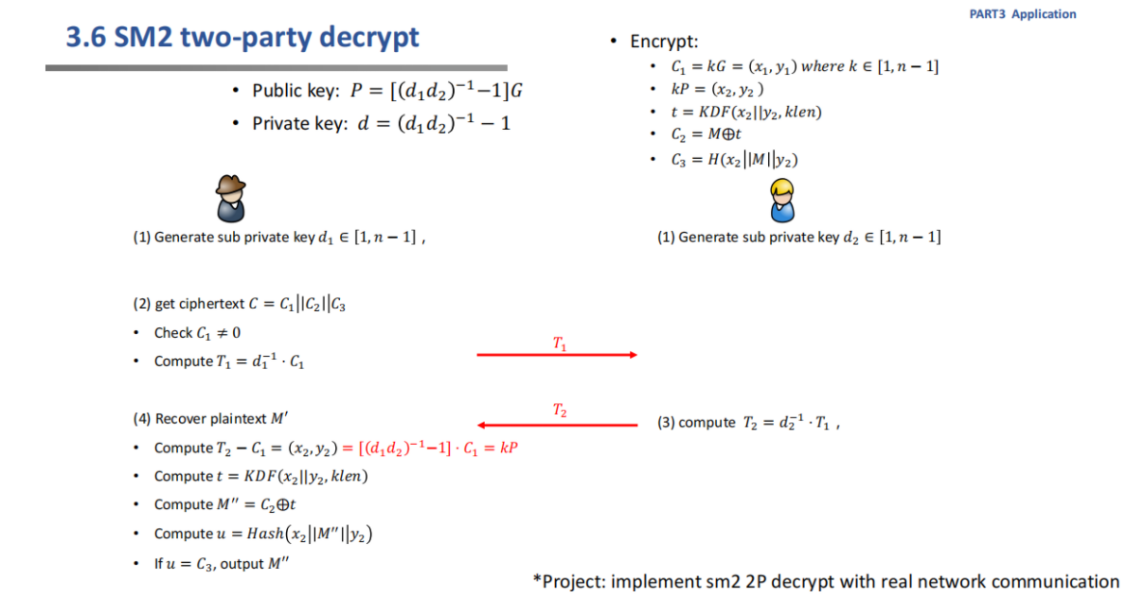
**此处编写了SM2的签名算法，导入的SM2.py即为github库中上传的SM2.py，其中涉及到的椭圆相关代码均在前面展示。**

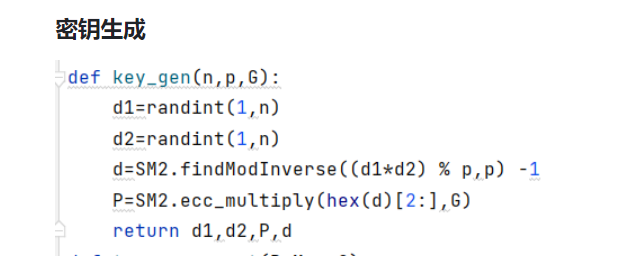
**②实验截图：**

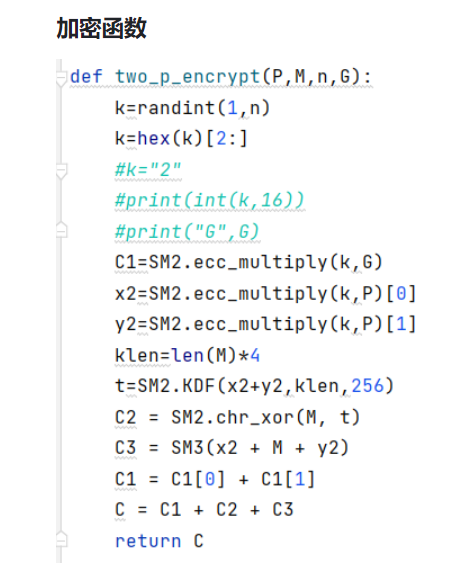
****

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

**①原理/代码解释：**









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

经过查阅资料后，Firefox和谷歌的记住密码插件在实现上有一些区别。

1.**存储密码的方式**：Firefox使用一个名为“Login Manager”的组件来存储密码。这个组件将密码以加密的形式保存在一个名为logins.json的文件中。谷歌浏览器使用一个名为“Password Manager”的组件来存储密码。这个组件将密码以加密的形式保存在一个名为Login Data的文件中。

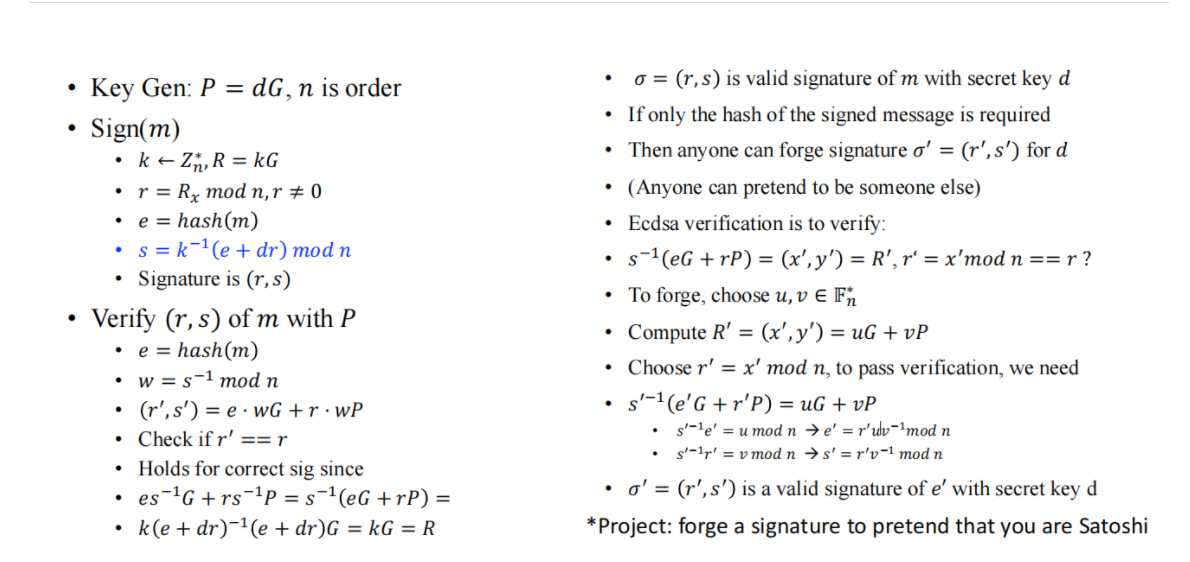
2.**加密算法**：Firefox使用一种称为“Login Manager Crypto”（LMC）的算法来加密密码。这个算法使用主密码（Master Password）作为密钥，并使用3DES算法来加密密码。谷歌浏览器则使用一种称为“Chrome Crypter”（CC）的算法来加密密码。这个算法使用操作系统登录密码作为密钥，并使用AES算法来加密密码。

3.**自动填充机制**：Firefox使用一个名为“Form Autofill”（FAF）的组件来实现自动填充功能。这个组件会根据已存储的密码信息，自动填充表单中的用户名和密码字段。谷歌浏览器则使用一个名为“Autofill”（AF）的组件来实现自动填充功能。这个组件除了填充用户名和密码字段，还可以填充其他表单字段，如姓名、地址等。

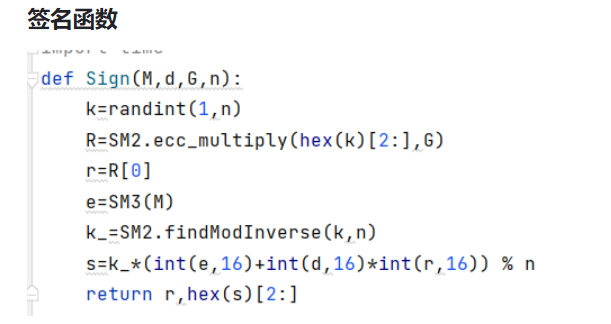
4.**用户界面**：Firefox的记住密码插件在用户界面上较为简洁，只提供了一个开关来控制是否记住密码。用户可以在浏览器设置中查看和管理已保存的密码。谷歌浏览器的记住密码插件在用户界面上更为丰富，用户可以在浏览器设置中查看和管理已保存的密码，并可以为不同的网站设置是否记住密码。

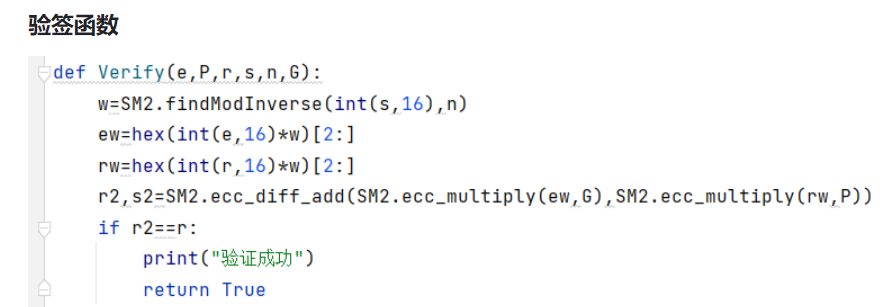
**·Project19: forge a signature to pretend that you are Satoshi**

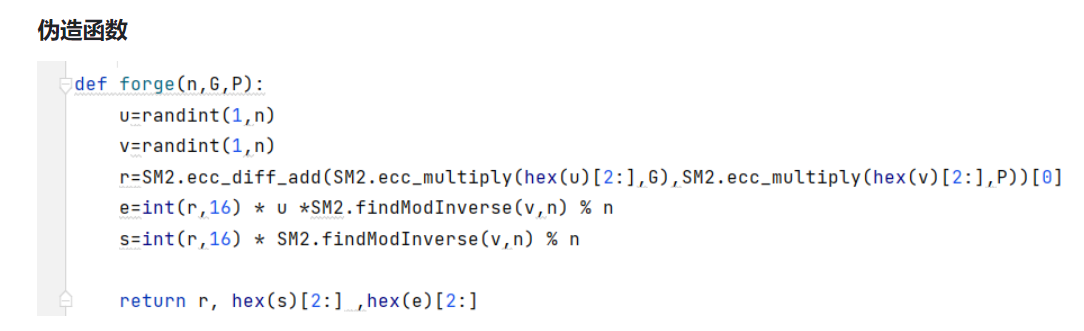
**①原理解释：**



**②代码分析：**







**③结果展示：**



**\*Project20: ECMH PoC**

**①代码/原理解释：**

**②实验截图：**

**\*Project21: Schnorr Bacth**

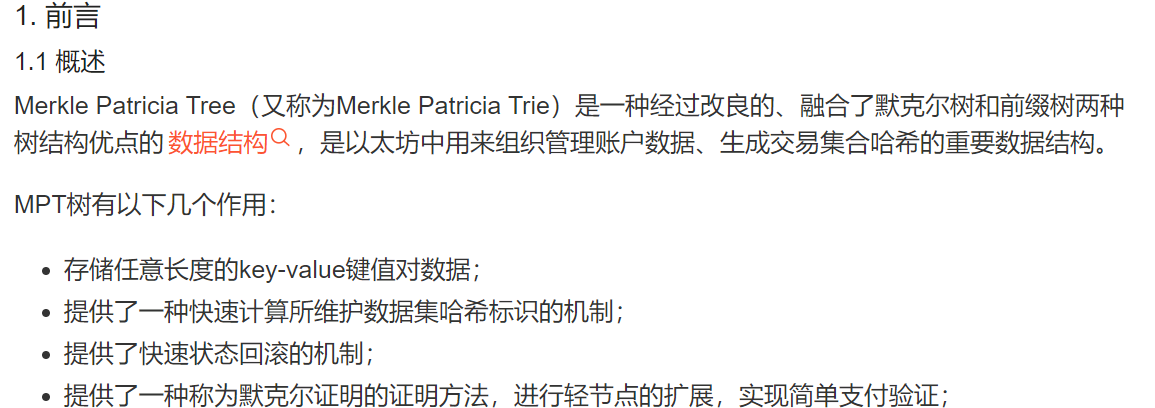
**①代码解释：**

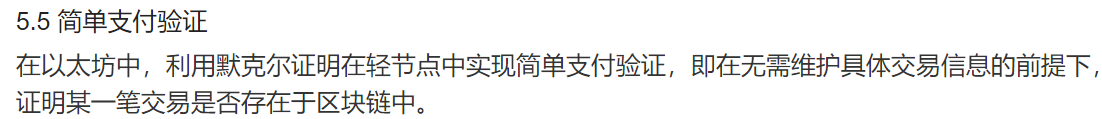
**②实验截图：**

**\*Project22: research report on MPT**

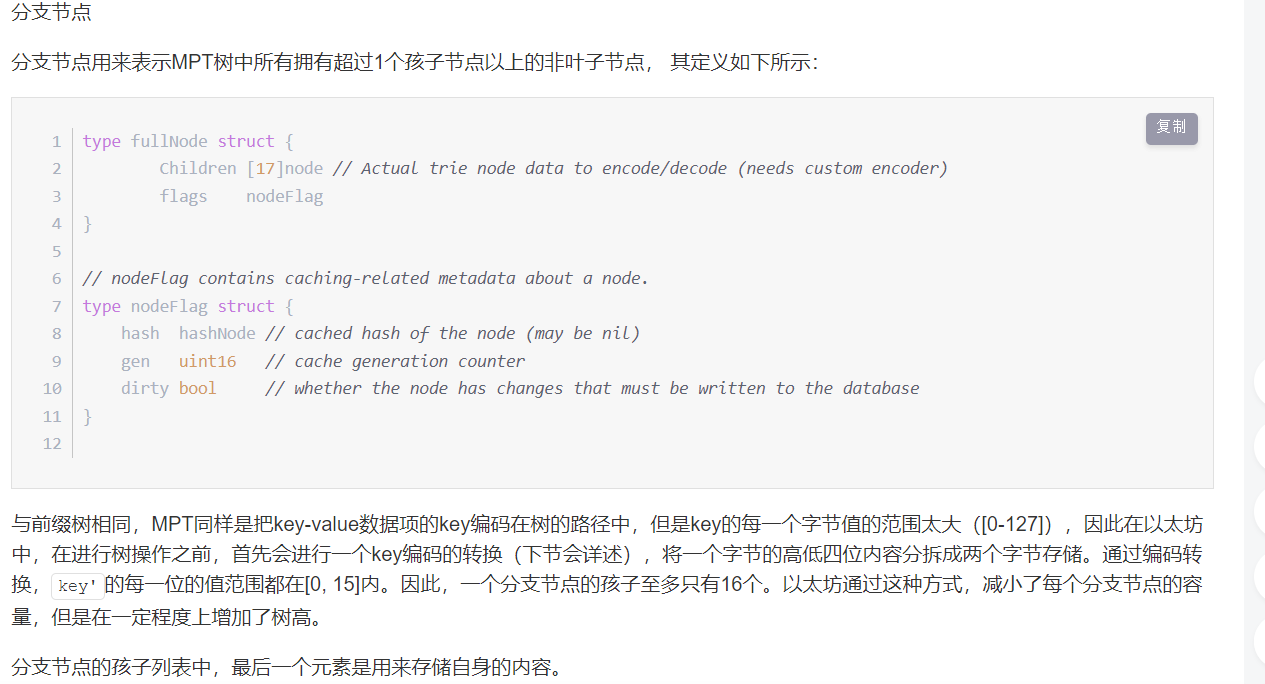
**①原理/代码解释：**

**（参考：https://blog.csdn.net/tianlongtc/article/details/80418923）**





**（相关代码也同样参考此篇文章）**

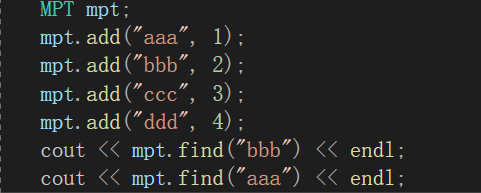


**②实验截图：**

**向树中插入了4个键值对**

**然后程序就会计算他们的hash值并以他为路径来记录1，2，3，4**

**之后查找bbb与aaa，会输出2、1。**



**↓**

