

**汇总报告**

山东大学网络空间安全学院

创新创业实践课程报告

刘莹 202100460164

|  |  |
| --- | --- |
| 硬件环境： | 处理器：AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz  内存: 16.0 GB (13.9 GB 可用) |
| 软件环境： | 操作系统：win 10  μC/OS-II:专门为嵌入式应用设计的实时操作系统。  ARM Cortex-M3 编程模型: ARM Cortex-M3 的嵌入式微处理器作为硬件环境的处理器。  IDLE (Python 3.9 64-bit) ：用于编写python代码。  Visual Studio 2019:用于编写C及C++代码。 |

实验环境：

**完成项目列表**：

|  |  |
| --- | --- |
| 项目序号 | 项目名称 |
| **project2** | implement the Rho method of reduced SM3 |
| **project 3** | implement length extension attack for SHA256 |
| **project 5** | implement Merkle Tree following RFC6962 |
| **project 10** | report on the application of this deduce technique in Ethereum with ECDSA |
| **project 14** | Implement a PGP scheme with SM2 |
| **project 15** | implement sm2 2P sign |
| **project 16** | implement sm2 2P decrypt |
| **project 17** | 比较Firefox和谷歌的记住密码插件的实现区别 |
| **project 22** | research report on MPT |

**Project\_2: 实现简化SM3的Rho方法**

**实现思路：**该实验设计f函数为f:H(x)f:H(x),即Wi=H(Wi−1)Wi=H(Wi−1)(除第一次输入信息mm外，f函数输入输出均为256bit)

Polladr rho method to fing collision：利用了生日悖论，使碰撞的复杂度降到O(n−−√)O(n)级别，同时能有效避免内存过大。

其思想是：利用f函数随机游走，构造出随机序列，可以发现，该序列发展到一定程度，会得到与之前相同的元素，形成环。因此可以应用到哈希函数中。

如何找环？如何找到进入环的元素（即找到哈希碰撞）？可以通过Floyd判圈法，规定两个指针，其中一个按照序列一步一步走，另外一个两步两步走。当两者相遇时，即找到了一个环，此时找到进入环的元素的两个前驱，即可找到一对碰撞。

如何找到刚进入环的元素？定义两个指针，一个指向初始信息mm，另外一个指向上一步两个指针相遇的点hh，两个指针都是一步一步走，最后相遇的点即为第一次进入环的点。

值得注意的是，要避免信息mm在环上，因此初始值要大于256bit。

**硬件环境：**

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

内存: 16.0 GB (13.9 GB 可用)

**软件环境：**

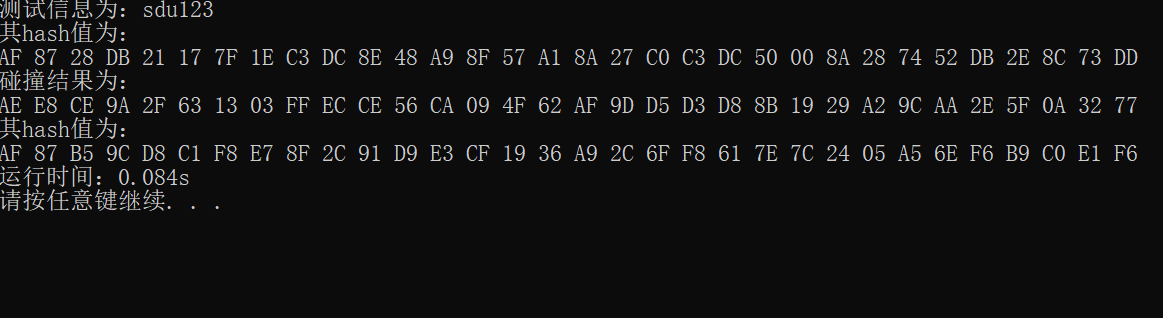
操作系统：win 10

Visual Studio 2019:用于编写C代码。

**实现方式：**#c

**运行时间：**测试十次平均运行时间为0.078s

**测试结果：**



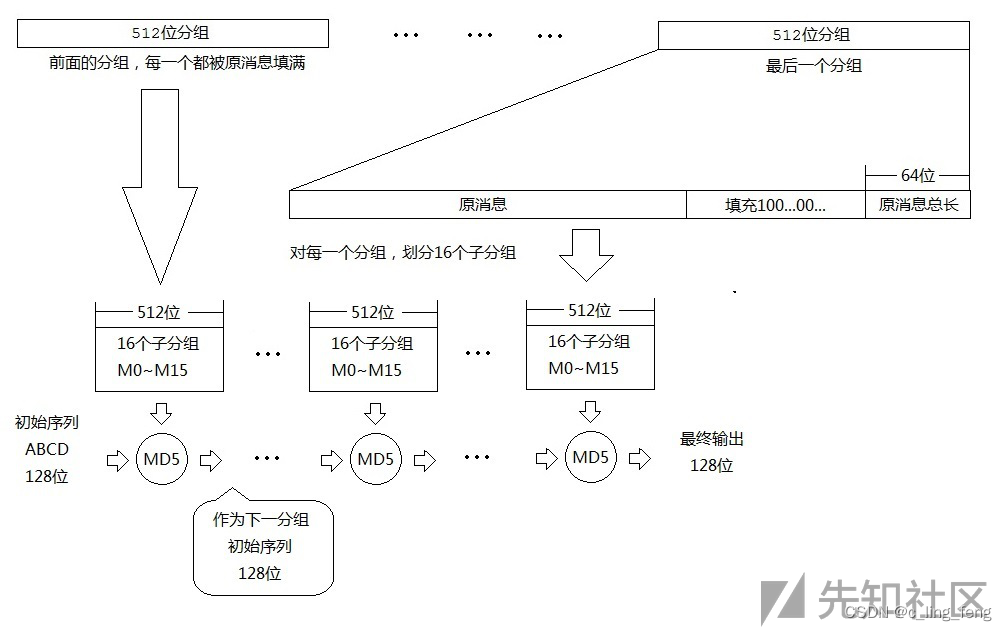
**Project\_3: 对SHA256进行长度扩展攻击**

**大致描述：**

系统会自动生成一串任意的字节串secret，以及一串明文c1然后进行哈希加密，m1=hash(secret+c1);我们可以得到加密后的密文m1，然后我们需要提供一个密文m2，以及一串明文c2使得m2=hash(secret+c2);一般来说，c1 in c2,c2实际上为c1的拓展部分，我们需要把明文的长度拓展一下，进行攻击，然后即可得到flag。



攻击原理：





分块：哈希计算进行加密时，通常是将明文信息以类似块密码的形式进行分组，对各个组块依次加密，每一块一般为512bit，也就是64bytes，每组的明文部分为56bytes，剩下的8bytes表示这块明文消息未填充前的长度。

填充：在明文消息的最后一块一般是不满足56ytes时就对这个最后一块进行padding填充，填充至56bytes的位置，，填充方式为，在16进制下，我们需要在消息后添加一个80，然后加0，直至56bytes时。

变量计算：哈希加密的在分块之后，每块在进行加密运算之前都会有一个链变量(key)，有每一个链变量与该块进行运算，除了第一块，每一块相对应的链变量都是前一块进行哈希计算后的字符串生成的，也就是说，每一块都对下一块有影响，（有点类似于CBC了），而第一块会有一个初始的链变量，为（无需考虑计算过程的细节）

而最后一块生成的链变量需要进行高低位互换（如：aabbccdd -> ddccbbaa），再拼接在一起就是我们计算出来的哈希值。

**硬件环境：**处理器：AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz

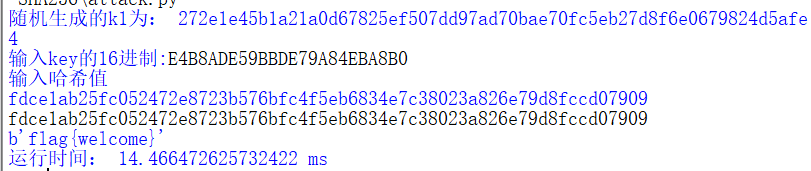
内存: 16.0 GB (13.9 GB 可用)

**软件环境：**操作系统：win 10；IDLE (Python 3.9 64-bit) ：用于编写python代码。

**实现方式：**python

**运行时间：**测试十次平均运行时间为18.2354263ms

**测试结果：**



**Project\_5: 按照RFC6962实现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。

Hash是一个把任意长度的数据映射成固定长度数据的函数。例如，对于数据完整性校验，最简单的方法是对整个数据做Hash运算得到固定长度的Hash值，然后把得到的Hash值公布在网上，这样用户下载到数据之后，对数据再次进行Hash运算，比较运算结果和网上公布的Hash值进行比较，如果两个Hash值相等，说明下载的数据没有损坏。可以这样做是因为输入数据的稍微改变就会引起Hash运算结果的面目全非，而且根据Hash值反推原始输入数据的特征是困难的。

生成一棵完整的Merkle树需要递归地对Hash节点对进行Hash，并将新生成的hash节点插入到Merkle树中，直到只剩一个Hash节点，该节点就是Merkle 树的根。在比特币的 Merkle树中两次使用到了SHA256算法，因此其加密哈希算法也被称为 double-SHA256。

(本次Merkle Tree构建过程使用的hash函数为SHA-256)

实现方式：c++

**Project\_10: ECDSA中的应用报告**

# ECDSA算法介绍报告

202100460164 刘莹

## 一、ECDSA概述

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。

## 二、ECDSA原理

ECDSA是ECC与DSA的结合，整个签名过程与DSA类似，所不一样的是签名中采取的算法为ECC，最后签名出来的值也是分为r,s。

**签名过程如下：**

1、选择一条椭圆曲线Ep(a,b)，和基点G；

2、选择私有密钥k（k<n，n为G的阶），利用基点G计算公开密钥K=kG；

3、产生一个随机整数r（r<n），计算点R=rG；

4、将原数据和点R的坐标值x,y作为参数，计算SHA1做为hash，即Hash=SHA1(原数据,x,y)；

5、计算s≡r - Hash \* k (mod n) 6、r和s做为签名值，如果r和s其中一个为0，重新从第3步开始执行

**验证过程如下：**

1. 接受方在收到消息(m)和签名值(r,s)后，进行运算 。

2、计算：sG+H(m)P=(x1,y1), r1≡ x1 mod p。

3、验证等式：r1 ≡ r mod p。 4、如果等式成立，接受签名，否则签名无效。

ECDSA处理过程：

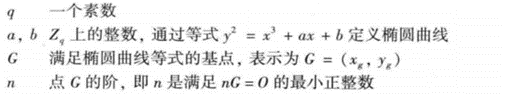
1.参与数字签名的所有通信方都使用相同的全局参数，用于定义椭圆曲线以及曲线上的基点

2.签名者首先生成一对公、私钥。对于私钥，选择一个随机数或者伪随机数作为私钥，利用随机数和基点算出另一点，作为公钥。

3.对消息计算Hash值，用私钥、全局参数和Hash值生成签名

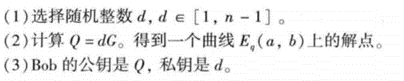
4.验证者用签名者的公钥、全局参数等验证。

全局参数：



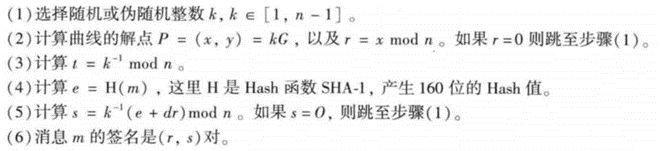
密钥生成：

每个签名者都要生成一对公、私钥，假设是Bob。



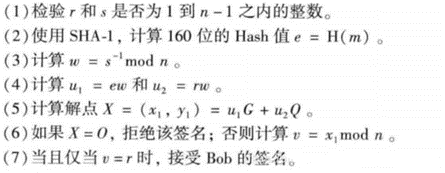
这里是定义在在这里插入图片描述

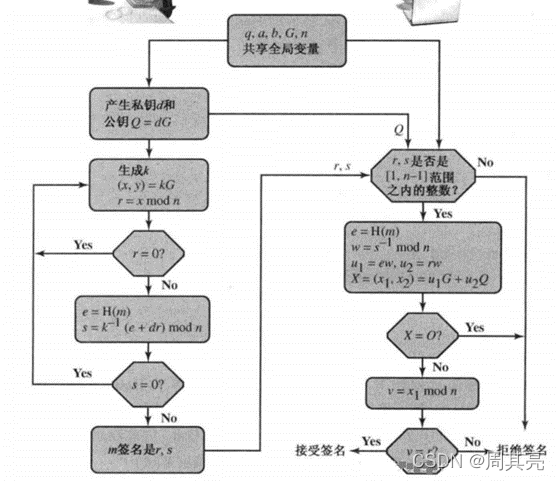
上的椭圆曲线，椭圆曲线上的乘法运算就是多个点的累加运算，最后的结果还是椭圆曲线上的点，有了公钥之后，Bob对消息m生成320字节的数字签名：



第2步确保最后算出的公钥（椭圆曲线上的点）是落在曲线上。第5步，如果为O点也是不符合的，所以也要重新生成。

Alice在获得Bob的公钥和全局参数后，即可校验签名。





该过程有效性证明如下，如果Alice收到的消息确实是Bob签署的，那么

s = (e+dr)k^-1 mod n

于是 k= (e+dr)s^-1 mod n

= (e s^-1 + dr s^-1) mod n

= (we +wdr) mod n

=(u1 + u2d) mod n

现在考虑u1G + u2Q = u1G + u2dG = (u1 + u2d)G = KG

在验证过程的步骤6中，有v = x1 mod n , 这里解点 X =(x1,y1) = u1G + u2Q。因为

R = x mod n 且x 是解点kG的x 坐标，又因为 我们已知 u1G + u2Q = KG，所以可得到 v = r。

## 三、ECDSA的实践

实施ECDSA时出现的一些问题在曲线和密钥生成或签名生成和验证过程中可能会出现一些漏洞。我们只调查与椭圆曲线的选择有关的问题。在实施过程中出现的一般问题，例如不检查一个点是否是无穷大的点，在这里不涉及。

第一个漏洞可能是操纵∶建议的安全曲线可能有一个后门不安全因素。

比特币和以太坊使用一个固定的曲线--secp256k1--并且只生成私钥和公钥。根据Safecurves,椭圆曲线secp256k1可以被认为有些“僵硬"，这意味着几乎所有的参数对公众是透明的，因此可以假设不是为了弱点而生成的。

### 3.1.梯子

椭圆曲线E上的一个点P的标量乘法在ECDSA中经常使用--例如用于公钥的生成。所谓的Mont- gomery梯子是一种快速而简单的算法，可以在恒定时间内完成这一计算。为了实现这个阶梯，椭圆曲线必须是一个特定的形状。secp256k1曲线不允许使用蒙哥马利阶梯。作为一个序列，除了简单和高效之外，secp256k1可能会因为某些计算的时间不恒定而泄露信息，从而导致侧信道攻击。这已经导致了成功的密钥提取，并反映在libsecp256ki实现套件中（见[GPP+16])。笔者不知道这个实现是否快速和简单。Brier-Joye梯子也可以应用，但会使运算速度降低很多。最后，Safecurves推荐蒙哥马利的单坐标梯子，因为它更容易实现对我们接下来讨论的攻击的保护。

### 3.2.扭曲的安全性

正如[BHH+14]中指出的，无效曲线攻击可能导致secp256k1的严重漏洞。因此，攻击者使用一个类似的椭圆曲线--原始曲线的扭曲--而只是假装使用原始曲线。如果这个扭曲在第2.2节的意义上是不安全的，而且实现者没有检查攻击者建议的点是否位于原始曲线上，那么他就有很大的机会在一些查询之后提取私钥。现在，secp256k1的标准二次扭曲也是一条安全的曲线(群的cardinality有220位素数﹔但更大的自动变形群又导致了四个扭曲。它们的最大素数除数是133、i88、135和16l，但也有较小的素数因子，使得攻击可能更加可行。一条不是扭曲安全的曲线需要在签名和验证过程中检查这些点是否真的在曲线上。这一点是可以做到的，但会使实现的效率和安全性降低。关于这种攻击的更多细节，我们可以参考[FLRVo8]，在那里我们可以找到一个关于secp256k1的计算实例。

### 3.3.完整性和不可分性

椭圆曲线上的加法和标量乘法的公式对于曲线上的某些特定点可能会略有变化。一个没有照顾到这些例外情况的实现会产生错误。完整性是指曲线没有例外情况。曲线secp256k1上的标量乘法是不完整的。 椭圆曲线上的点的表示通常可以从随机产生的字符串中区分出来。为了掩盖椭圆曲线上的点的外观，有一些可用的策略（参见[BL13])。这些构造不适用于secp256k1。

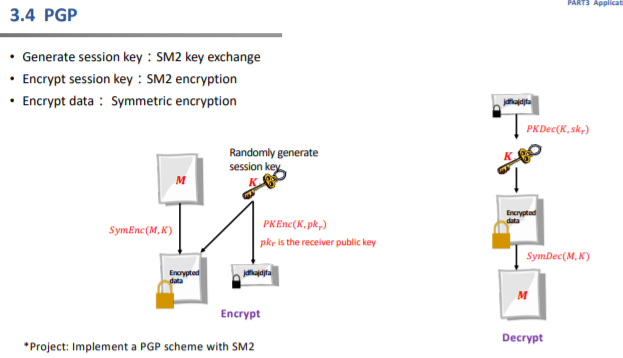
### 3.4.多个ECDLP

在比特币中，任何用户的公钥都是由secp256k1，他们的私钥和基点P产生的。情况看起来如下。假设我们有L个用户，他们的公钥Qi = ai PE(Fp), 1 iL。如果有人想找到他们的私钥，她必须解决以下离散对数问题。l Qi = ai P(1 ≤i≤ L)。 我们必须要解决L倍的离散对数问题吗?或者可以利用ECDLPs发生在同一基点P的椭圆曲线上的事实?到目前为止，还没有已知的算法能够更快地找到一个实例的解决方案。但使用Pollard's rho方法的扩展版本，一旦找到一个，就能逐步加速找到其他的解。例如，如果L<r八，r是我们组的大小，Kuhn和Struik表明，我们平均需要2rL组操作。在[HMVo4]第164页提出的Pollard's rho的扩展算法，在找到第一个实例后，第二个ECDLP的运行时间降低了50%，第三个降低了37%，以此类推。所以我们必须确保找到一个实例是不可行的，而secp256k1做到了。

**参考文献：** 1.https://weibo.com/ttarticle/p/show?id=2309404720225936605782#\_loginLayer\_1659241375037

2.Hartwig Mayer.ECDSA Security in Bitcoin and Ethereum: a Research Survey

**Project\_14:用SM2实现PGP方案**

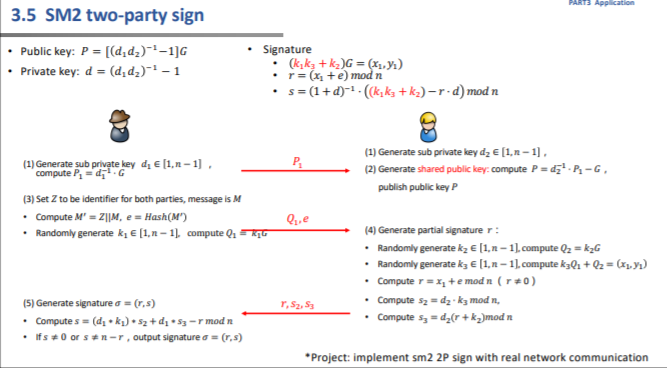


PGP技术是一个基于非对称加密算法RSA公钥体系的邮件加密技术，也是一种操作简单、使用方便、普及程度较高的加密软件。PGP技术不但可以对电子邮件加密，防止非授权者阅读信件；还能对电子邮件附加数字签名，使收信人能明确了解发信人的真实身份；也可以在不需要通过任何保密渠道传递密钥的情况下，使人们安全地进行保密通信。  
PGP技术创造性地把RSA非对称加密算法的方便性和传统加密体系结合起来，在数字签名和密钥认证管理机制方面采用了无缝结合的巧妙设计，使其几乎成为最为流行的公钥加密软件包。  
　　PGP使用两个密钥来管理数据：一个用以加密，称为公钥（Public Key）；另一个用以解密，称为私钥(Private Key)。公钥和私钥是紧密联系在一起的，公钥只能用来加密需要安全传输的数据，却不能解密加密后的数据；相反，私钥只能用来解密，却不能加密数据。

实现方式：python

运行时间：测试十次平均运行时间为0.1035746573s

**Project\_15: 实现sm2 2P签名**



**签名流程：**M’ = ZA || Msg，e = Hash(M’)，并转为大数；生成随机数k，范围0<k<n；计算kG = (x1, y1)，r = (e + x1) mod n, 若r0或(r+kn)则重新生成k；s = (k-rd) / (1+d) mod n，若s==0则重新生成k；返回签名(r, s)

ZA：关于用户A的可辨别标识、部分椭圆曲线系统参数和用户A公钥的杂凑值。

**验签流程：**检查r, s范围，M’ = ZA || Msg，e = Hash(M’), 并转为大数；t = (r + s) mod n，若t==0，则验证不通过；(x1, y1) = sG + tQ，计算R = (e + x1) mod n == r是否成立，成立则验签通过。

硬件环境：

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

内存: 16.0 GB (13.9 GB 可用)

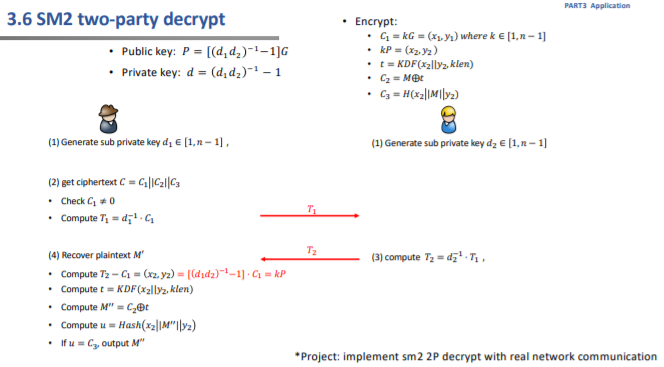
软件环境：

操作系统：win 10

Visual Studio 2019:用于编写C代码。

实现方式：#c

**Project\_16:实现sm2 2P解密**



算法步骤：

1. 加密算法

设需要发送的消息为比特串M，klen为M的比特长度。

为了对明文M进行加密，作为加密者的用户A应实现以下运算步骤：

A1：用随机数发生器产生随机数k∈[1,n-1]；

A2：计算椭圆曲线点C1=[k]G=(x1,y1)，将C1的数据类型转换为比特串；

A3：计算椭圆曲线点S=[h]PB，若S是无穷远点，则报错并退出；

A4：计算椭圆曲线点[k]PB=(x2,y2)，将坐标x2、y2 的数据类型转换为比特串；

A5：计算t=KDF(x2 || y2, klen)，若t为全0比特串，则返回A1；

A6：计算C2 = M ⊕ t；

A7：计算C3 = Hash(x2 || M || y2)；

A8：输出密文C = C1 || C2 || C3。

2. 解密算法

设klen为密文中C2的比特长度。

为了对密文C=C1 || C2 || C3 进行解密，作为解密者的用户B应实现以下运算步骤：

B1：从C中取出比特串C1，将C1的数据类型转换为椭圆曲线上的点，验证C1是否满足椭圆曲线方程，若不满足则报错并退出；

B2：计算椭圆曲线点S=[h]C1，若S是无穷远点，则报错并退出；

B3：计算[dB]C1=(x2,y2)，将坐标x2、y2的数据类型转换为比特串；

B4：计算t=KDF(x2 || y2, klen)，若t为全0比特串，则报错并退出；

B5：从C中取出比特串C2，计算M′ = C2 ⊕ t；

B6：计算u = Hash(x2 || M′ || y2)，从C中取出比特串C3，若u ≠ C3，则报错并退出；

B7：输出明文M′。

函数简要介绍：

1. 数据类型转换

本部分所定义的函数实现了域元素、字节串、比特串、整数等数据类型的转换，方便计算与阅读。

def int\_to\_bytes(x, k)：实现整数到字节串的转换。接收非负整数x和字节串的目标长度k，k满足2^8k > x。返回值是长为k的字节串。注意字节串长度k是给定的参数！

def bytes\_to\_int(M)：字节串到整数的转换。接受长度为k的字节串。返回值是整数x。

def bits\_to\_bytes(s)：比特串到字节串的转换。接收长度为m的比特串s。返回长度为k的字节串M。其中k = [m/8] 向上取整。先判断字符串整体是否能正好转换为字节串，即长度是否为8的倍数。若不是则左填充至长度为8的倍数。

def bytes\_to\_bits(M)：字节串到比特串的转换。接收长度为k的字节串M，返回长度为m的比特串s，其中m = 8k。字节串逐位处理即可。

def fielde\_to\_bytes(e)：域元素到字节串的转换。域元素是整数，转换成字节串要明确长度。文档规定域元素转换为字节串的长度l是ceil(ceil(log(q, 2)/8))。接收的参数是域元素a，返回l长字节串M 。

def bytes\_to\_fielde(M)：字节串到域元素的转换。直接调用bytes\_to\_int( )。接收的参数是字节串M，返回域元素a。

def fielde\_to\_int(a)：域元素到整数的转换。域元素就是整数，直接返回即可。

def point\_to\_bytes§：点到字节串的转换。接收的参数是椭圆曲线上的点p，元组表示。输出字节串S。选用未压缩表示形式，即字节串s = PC + x + y，共1 + 2l个字节。

def bytes\_to\_point(s)：字节串到点的转换。接收的参数是字节串s，返回椭圆曲线上的点p，点P的坐标用元组表示。

附加数据类型转换。是上述几种数据类型转换的复合转换，或者是数制之间的转换。注意字符串的填充即可。共定义了def fielde\_to\_bits(a)、def point\_to\_bits§、def int\_to\_bits(x) 、def bytes\_to\_hex(m)、def bits\_to\_hex(s)、def hex\_to\_bits(h)、def hex\_to\_bytes(h)、def fielde\_to\_hex(e)几种函数。

2.辅助函数

本部分定义的函数是用于辅助实现SM2加解密算法中某些步骤的模块，这些模块若直接在加解密算法中会使代码变长造成阅读困难，因此将其定义为单独的函数以使结构清晰。

def add\_point(P, Q, p)：椭圆曲线上的点加运算。接收的参数是元组P和Q，表示相加的两个点，p为模数。返回二者的点加和。

def double\_point(P, p, a): 二倍点算法。不能直接用点加算法，否则会发生除零错误。接收的参数是点P，素数p，椭圆曲线参数a。返回P的二倍点。

def mult\_point(P, k, p, a)：多倍点算法。通过二进制展开法实现。接收的参数[k]p是要求的多倍点，m是模数，a是椭圆曲线参数。

def frac\_to\_int(up, down, p)：将分式模运算转换为整数。输入 up/down mod m, 返回该分式在模m意义下的整数。点加和二倍点运算时求λ用。

def calc\_inverse(M, m)：模逆算法。返回M模m的逆。在将分式模运算转换为整数时用，分子分母同时乘上分母的模逆。

def on\_curve(args, P)：验证某个点是否在椭圆曲线上。接收的参数是椭圆曲线系统参数args和要验证的点P(x, y)。

def KDF(Z, klen)：密钥派生函数KDF。接收的参数是比特串Z和要获得的密钥数据的长度klen。返回klen长度的密钥数据比特串K。

3. 加解密算法

本部分是加解密的两个函数。

def encry\_sm2(args, PB, M)：加密算法。接收的参数是椭圆曲线系统参数args(p, a, b, h, G, n)。其中n是基点G的阶。PB是B的公钥，M是明文消息。

def decry\_sm2(args, dB, C)：解密算法。接收的参数为椭圆曲线系统参数args(p, a, b, h, G, n)。dB是B的私钥，C是密文消息。

4. 参数获取

本部分的函数用于获取算法使用的相关数据。

def get\_args( )：椭圆曲线系统参数args(p, a, b, h, G, n)的获取。

def get\_key( )：密钥获取。本程序中主要是消息接收方B的公私钥的获取。

硬件环境：

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

内存: 16.0 GB (13.9 GB 可用)

软件环境：

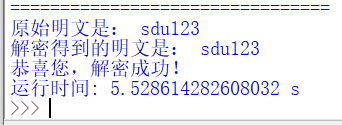
操作系统：win 10

IDLE (Python 3.9 64-bit) ：用于编写python代码。

实现方式：python

效果：运行十次平均时间为5.428614s

测试结果：



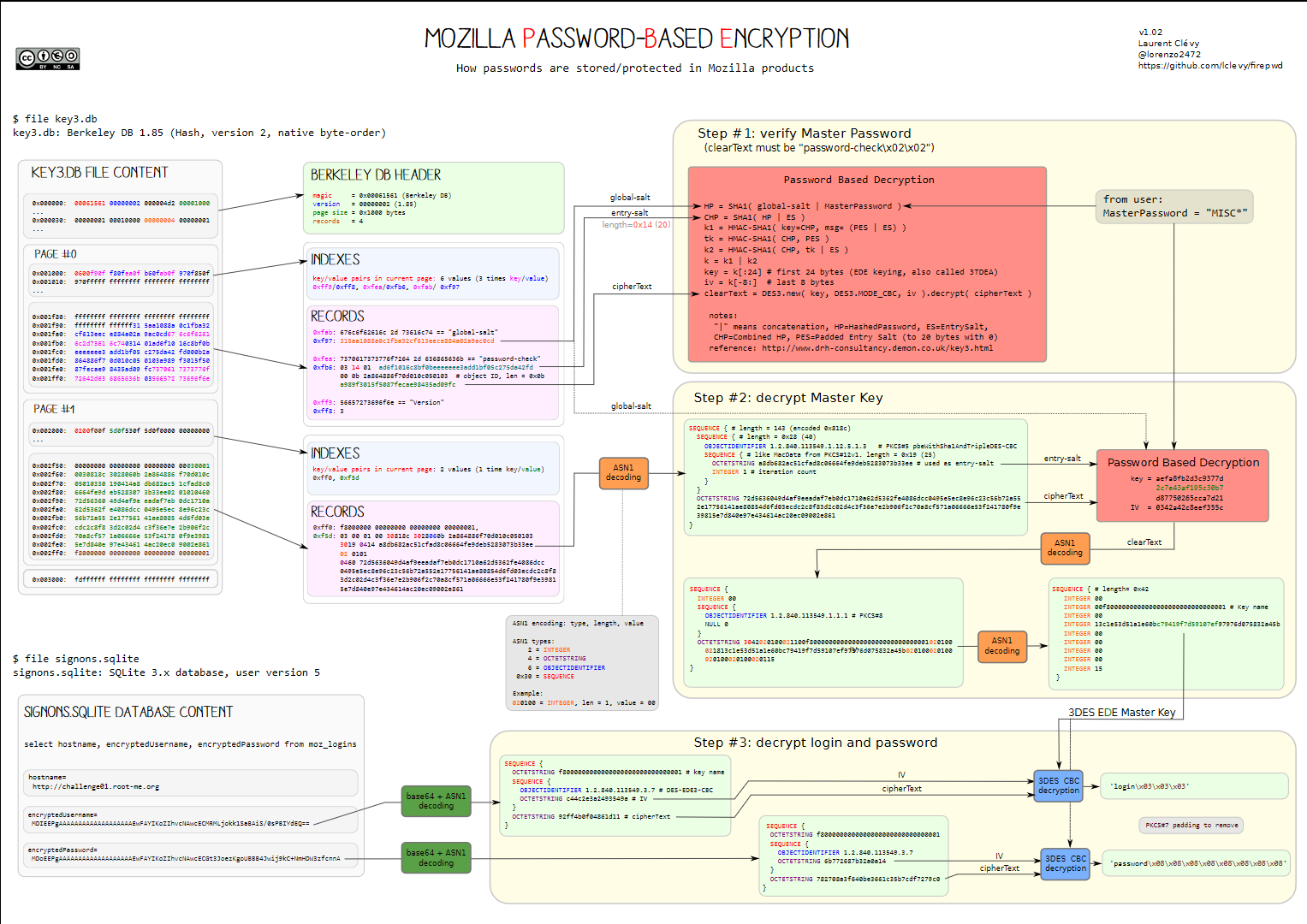
**Project\_17: 比较Firefox和谷歌的记住密码插件的实现区别**

# 比较Firefox和谷歌的记住密码插件的实现区别

202100460164 刘莹

## （一）Firefox记住密码插件

### 一、Firefox加密流程：



**应用了SHA-1哈希加密算法,将用户的真实密码加随机盐组成的字符串加密,形成密钥。**

### 二、登录信息存储过程

以Firefox 版本 >=58.0.2为例，logins.json将用户所有登录信息（包括URL，用户名，密码和其他元数据）存储为JSON。值得注意的是，**这些文件中的用户名和密码均经过3DES加密，然后经过ASN.1编码，最后写入base64编码的文件中**。

key4.db是一个sqlite数据库，里面存储用于3DES解密logins.json的密钥，以及被加密的用于验证主密钥解密的password-check值，里面有两个表metaData和nssPrivate。

metaData中id为password的item1列为包含加密期间使用的全局盐值(globalSalt)；item2列为ASN.1编码后的加密password-check数据，里面包含被加密的password-check字符串和用于加密的入口盐值(entrySalt)。

**在加密password-check数据和主密钥使用了hmacWithSHA256的哈希算法和AES256 cbc的加密算法。**

### 三、Firefox也可以下载一名为Lastpass的记住密码插件

Lastpass是一个在线密码管理器和页面过滤器，采用了强大的加密算法，自动登录/云同步/跨平台/支持多款浏览器。

1、Lastpass是一个在线密码管理器和页面过滤器，它可以网页浏览更加的轻松和更安全。

2、Lastpass采用了强大的密码加密算法（使用了256位的AES密匙），保证了在本机上不获取得到用户的信息，所以用户可以在任何时候和地点取回用户的信息。用户在本机的密码将被加密存储，用户的密码可以存在用户的PC，MAC和移动设备上。用户大可以放心，只有用户的Lastpass密码才能解锁它们。如果用户换了计算机，或者计算机丢失了，用户也不要惊慌，因为用户的加密数据将被备份在用户在官方主页的账户中，只要用户登陆官方主页和安装Lastpass，即可无缝的恢复用户的密码。

3、LastPass提供了一个额外定制，包括iOS设备，黑莓，安卓(Android)，Windows Mobile和Symbian应用程序，加强支持，多因素认证。

**LastPass采用256位AES加密算法对本地和网站上的密码数据库进行加密，并在数据传输时使用SSL加密连接等措施确保数据安全**。

## （二）谷歌的记住密码插件

### 一、Chrome浏览器记住密码插件介绍

1. chrome浏览器密码保存和同步功能

chrome浏览器提供了一项非常方便的功能，即自动保存和同步密码。当用户第一次登录某个网站时，chrome会提示用户是否保存该网站的用户名和密码。如果用户选择保存，则下一次访问该网站时，chrome会自动填充用户名和密码，并且在不同设备间同步这些信息。这项功能为用户省去了记住各种复杂密码的烦恼，但也为黑客窃取密码提供了便利。

2. chrome浏览器密码加密

chrome浏览器并没有直接将用户的密码明文保存在本地或者云端服务器上，而是采用了加密技术来保护用户隐私。**chrome浏览器使用AES算法对用户的密码进行加密，并采用PBKDF2算法生成一个密钥，用于加密和解密用户密码**。

3. chrome浏览器密码抓取原理

尽管chrome浏览器使用了加密技术来保护用户的密码，但黑客仍然有多种方法来窃取这些密码。其中最常用的方法是通过恶意软件或者浏览器插件来实现。恶意软件可以通过截获chrome浏览器的输入事件来获取用户的密码明文。例如，当用户在chrome浏览器中输入密码时，恶意软件可以拦截输入事件，并将用户的密码明文发送给黑客服务器。浏览器插件也可以获取用户的密码信息。一些恶意插件会伪装成正常插件，当用户安装这些插件后，它们就可以访问chrome浏览器保存的所有密码信息，并将这些信息上传到黑客服务器上。

Chrome浏览器对显示的密码进行了一道验证，需要输入正确的电脑账户密码才能查看。为了执行加密（在Windows操作系统上），Chrome使用了Windows提供的API，该API只允许用于加密密码的Windows用户账户去解密已加密的数据。所以基本上来说，你的主密码就是你的Windows账户密码。所以，只要你登录了用自己的账号Windows，Chrome就可以解密加密数据。

### 二、Chrome浏览器密码存储机制

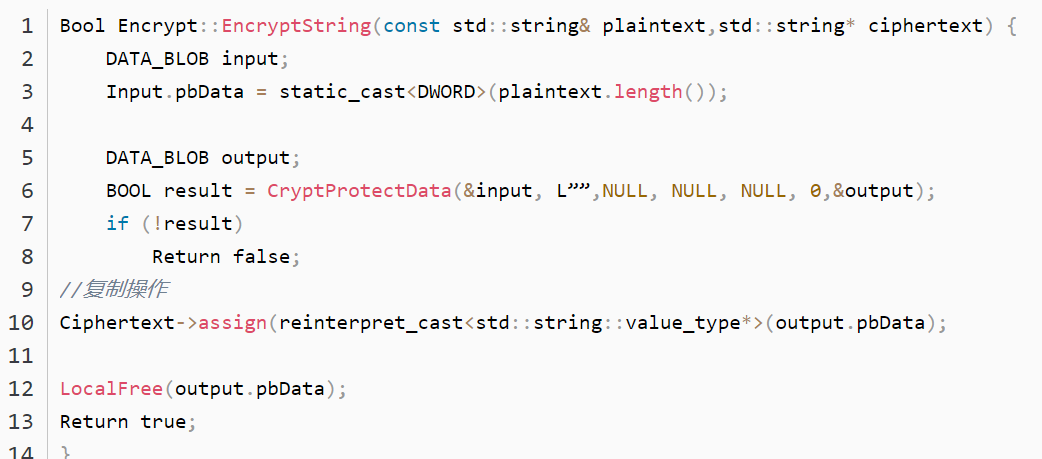
谷歌浏览器加密后的密钥存储于%APPDATA%…\Local\Google\Chrome\User Data\Default\Login Data”下的一个SQLite数据库中。

首先，我们作为用户登录一个网站时，会在表单提交Username以及Password相应的值，Chrome会首先判断此次登录是否是一次成功的登录，部分代码如下：



当我们登录成功时，并且使用的是一套新的证书(也就是说是\*\*\*次登录该网站)，Chrome就会询问我们是否需要记住密码。

那么登录成功后，密码是如何被Chrome存储的呢?答案在EncryptedString函数，通过调用EncryptString16函数，代码如下：



代码利用了Widows API函数CryptProtectData(前面提到过)来加密。当我们拥有证书时，密码就会被回复给我们使用。在我们得到服务器权限后，证书的问题已经不用考虑了，所以接下来就可以获得这些密码。下面通过Python代码实现从环境变量中读取Login Data文件的数据，再获取用户名和密码，并将接收的结果通过win32crypt.CryptUnprotectData解密密码。



## （三）总结二者区别

和Chrome浏览器不同，Mozilla拥有自己的加密库，被称为网络安全服务（NSS），特别之处是NSS使用了ASN.1进行数据序列化。ASN.1 – Abstract Syntax Notation dot one，数字1被ISO加在ASN的后边，是为了保持ASN的开放性，可以让以后功能更加强大的ASN被命名为ASN.2等，但至今也没有出现。ASN.1是一种对数据进行表示、编码、传输和解码的数据格式。它提供了一整套正规的格式用于描述对象的结构，而不管语言上如何执行及这些数据的具体指代，也不用去管到底是什么样的应用程序。

Chrome和Firefox之间的有一个很大的区别，那就是Firefox允许用户提供一个主密码来加密所有存储的登录名和密码。如果用户设置了主密码，需要解密者提供主密码才能解密登录信息。

**Project\_22: research report on MPT**

# Research report on MPT

一、概念

默克尔树（Merkle Patricia Tree）在以太坊中是一种通用的，用来存储键值对的数据结构，可以简称为“MPT”，是字典树Redix tree的变种，也是以太坊的核心算法之一。

MPT对于树中节点的插入、查找、删除操作，这种结构可以提供对数级别的复杂度O(log(N))，所以它是一种相对高效的存储结构。

二、如何根据键值对构造默克尔树

（一）、节点类型

1. **Branch**

（1）由17个元素组成的元组，格式为：(v0,……,v15,vt)。

（2）其中，v0～v15的元素是以其索引值（0x0~0xf）为路径的子节点数据的keccak256哈希值，如果没有子节点数据则元素为空。

（3）vt为根节点到当前节点的父节点所经过的路径对应的value值，也就是根节点到父节点所经过的路径组成了一个键key，这个key对应的value存在vt里面，如果这个key没有对应的value，那么vt为空。

1. **Leaf**

（1）两个元素组成的元组，格式为：（encodePath,value）；

（2）encodedPath为当前节点路径的十六进制前缀编码；

（3）value是从根节点到当前节点路径组成的键对应的值。

1. **Extension**

（1）两个元素组成的元组，格式为：（encodePath,key）；

（2）encodedPath为当前节点路径的十六进制前缀编码；

（3）key为当前节点子节点数据的keccak256哈希值。

（二）十六进制前缀编码

branch和extension元组的第一个元素encodePath就是当前节点路径的十六进制前缀编码（Hex-Pretix Encoding，HP编码）。使用HP编码能够区分节点是扩展结点还是叶子节点。

而HP编码，和当前节点类型还有当前路径半字节长度的奇偶有关。

共有四种前缀：



所以extension节点有两种前缀：0x00、0x1；leaf有两种前缀：0x20、0x3。

可以看到最终前缀在偶数个半字节0x0、0x2后补了一个0，变成了0x00，0x20，目的是为了凑成整字节，避免出现半字节导致长度不便于合并。

HP前缀需要放在原始路径前面去组成HP编码，实例：



三、构造一颗默克尔树

上面的概念不容易理解，现在我们以下面的例子，一步步来进行树的构造，帮助我们更好的理解：  
我们假设有一组（4个）键值对数据需要用树来存储：

<64 6f> : 'verb'

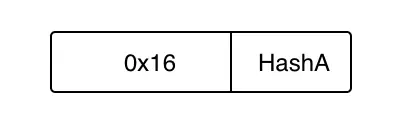
<64 6f 67> : 'puppy'

<64 6f 67 65> : 'coin'

<68 6f 72 73 65> : 'stallion'

为方便解释说明以及阅读，我们把键值对数据的“键”表示为十六进制字符串，“值”则保留为原始字符串。在实际使用时，它们都需要经过特定的编码变换。

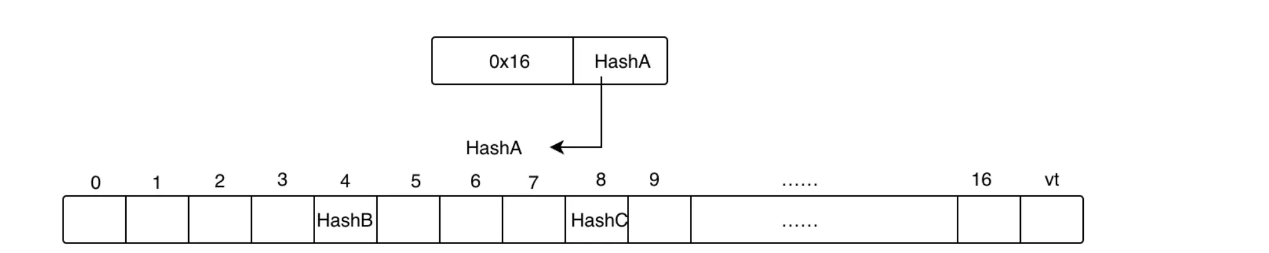
1、每棵树都有根节点，默克尔树的根节点会保存当前路径和子节点哈希，所以很明显，根节点会是一个extension节点。  
上面节点类型介绍了extension格式为：（encodePath,key），encodePath是十六进制的HP编码。分析给出的4个键我们可以得出都是以6开头，后面分为4、8两条路。所以根节点存储的共同路径值为0x6。  
由于0x6只有一位，所以路径长度是奇数，节点又是extension类型，所以HP前缀是0x1，组合出来的HP编码：0x16。  
所以当前默克尔树如下图：



HashA代表着子节点的哈希值。

2、根节点已经找到，但在根节点后出现了两条路，这个时候需要使用branch来处理这种多条路径的情况。

上文说到，branch由17个元素组成的元组，格式为：(v0,……,v15,vt)。其中，v0～v15是以其索引值（0x0~0xf）为路径的子节点数据的keccak256哈希值，如果没有子节点数据则为空。  
这里4和8就是索引值，4、8对应元素是其字节点的哈希值。  
所以当前默克尔树如下图：

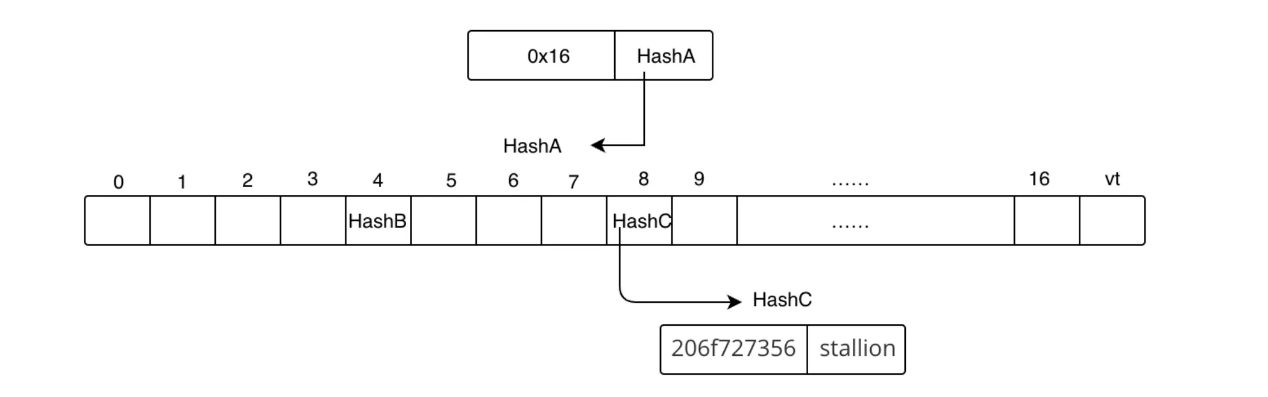


3、我们可以观察到，在0x68后只有唯一路径了，即0x6f727356，而value为“stallion”，所以不再分叉的情况下，就不是branch或者extension了，而应该是一个叶节点。

上文提到，leaf节点是两个元素组成的元组，格式为：（encodePath,value），encodedPath为当前节点路径的十六进制前缀编码，value是从根节点到当前节点路径组成的键，所对应的值。

当前节点的路径是0x6f727356，长度是偶数，节点类型是leaf，所以可以得出HP前缀是0x20，HP编码是0x206f727356。所以可得该leaf节点：（0x206f727356,"stallion"）。

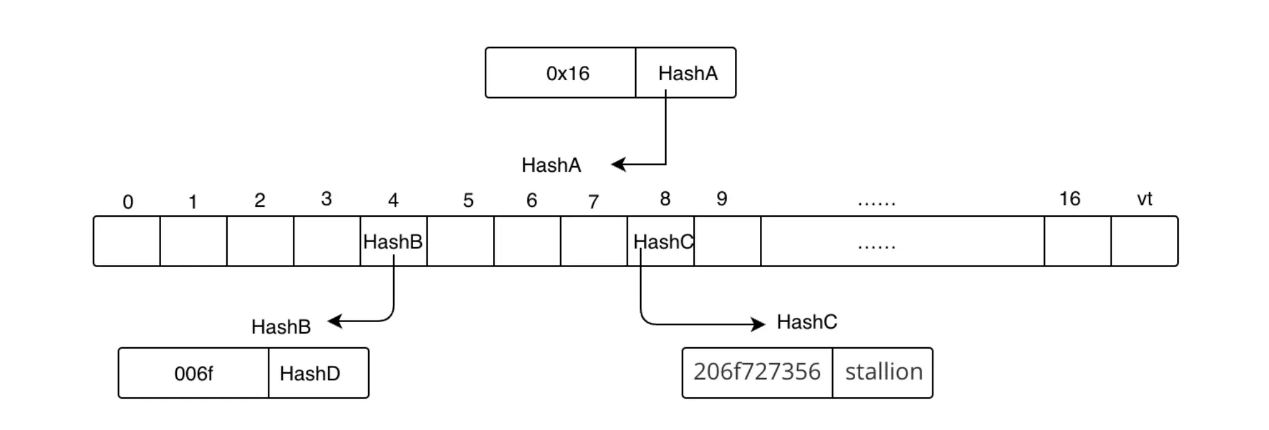
所以当前默克尔树如下图：



4、说完了8，我们再说说4这部分，路径4后面有共同路径6f，6f后才产生null和6两条分叉。

共同路径6f是一个extension节点，extension节点格式不再介绍，开始计算HP编码，6f长度是偶数，又是extension类型，所以HP前缀为0x00，HP编码为0x006f。

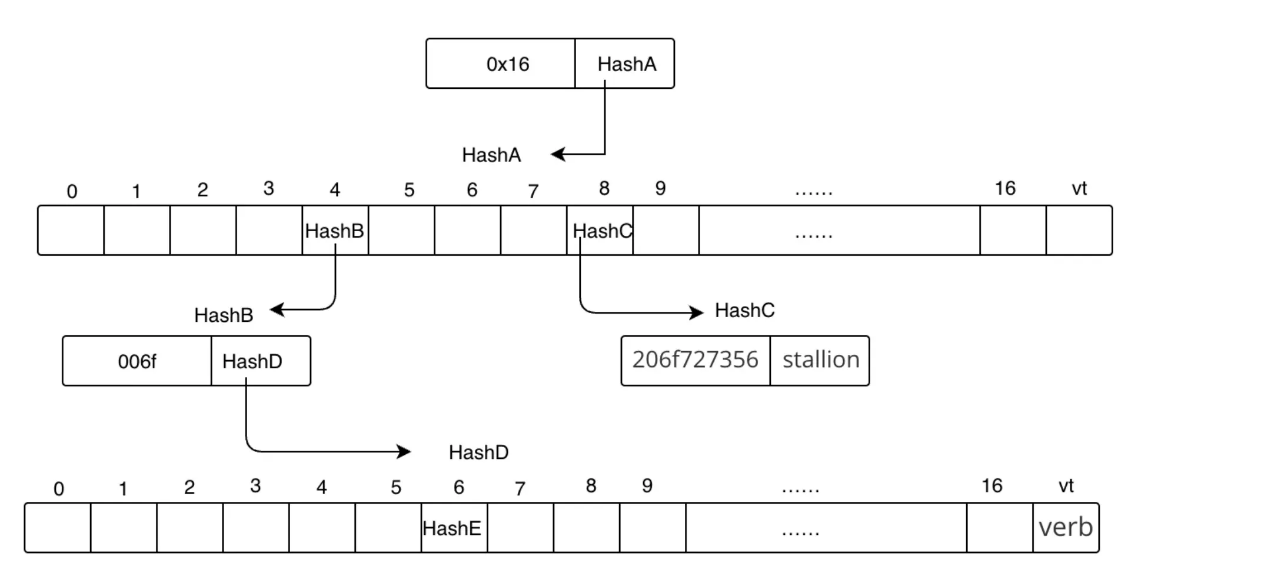
所以当前默克尔树如下图：



5、6f后分出了null和6，是多条路径，所以HashD的节点是一个branch节点，6是索引值，索引为6的元素存储着子节点hash；而null是没有的，上文提到：vt为根节点到当前节点的父节点所经过的路径组成的键对应的value。

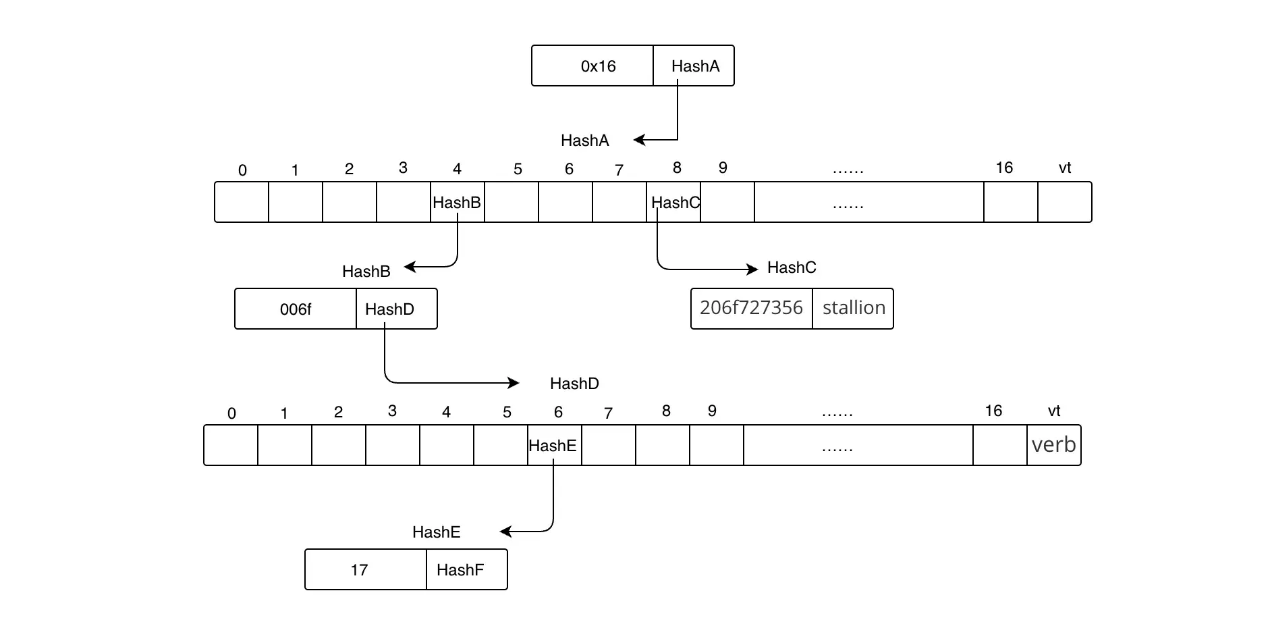
则代表当前HashD节点该存储从根节点到父节点0x646f组成的键对应的值：'verb'。那么该由HashD的vt保存'verb'。

所以当前默克尔树如下图：

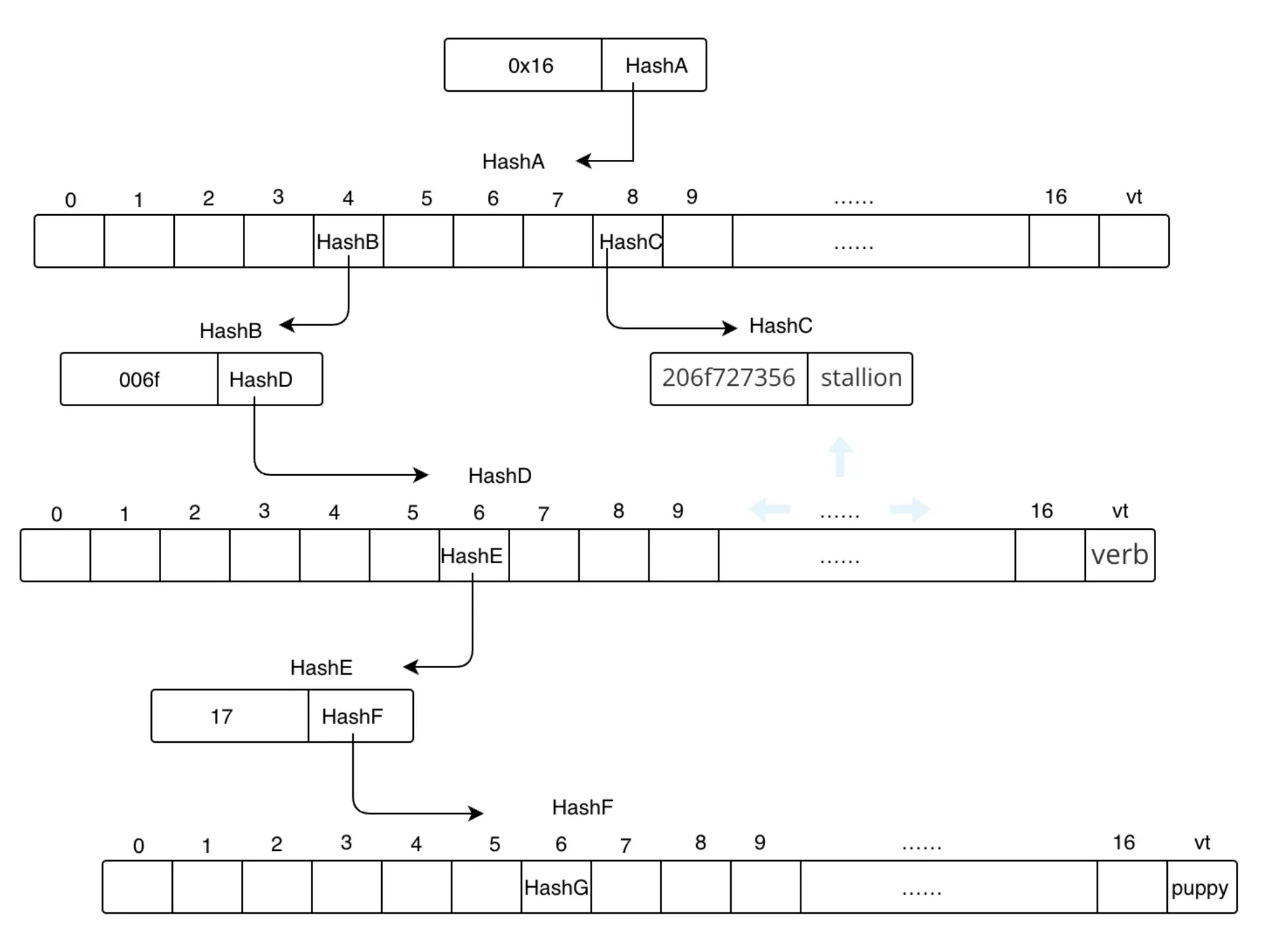


6、接下来是共同路径7，一个extension节点，开始计算HP编码，7长度是奇数，又是extension类型，所以HP前缀为0x1，HP编码为0x17。

所以当前默克尔树如下图：

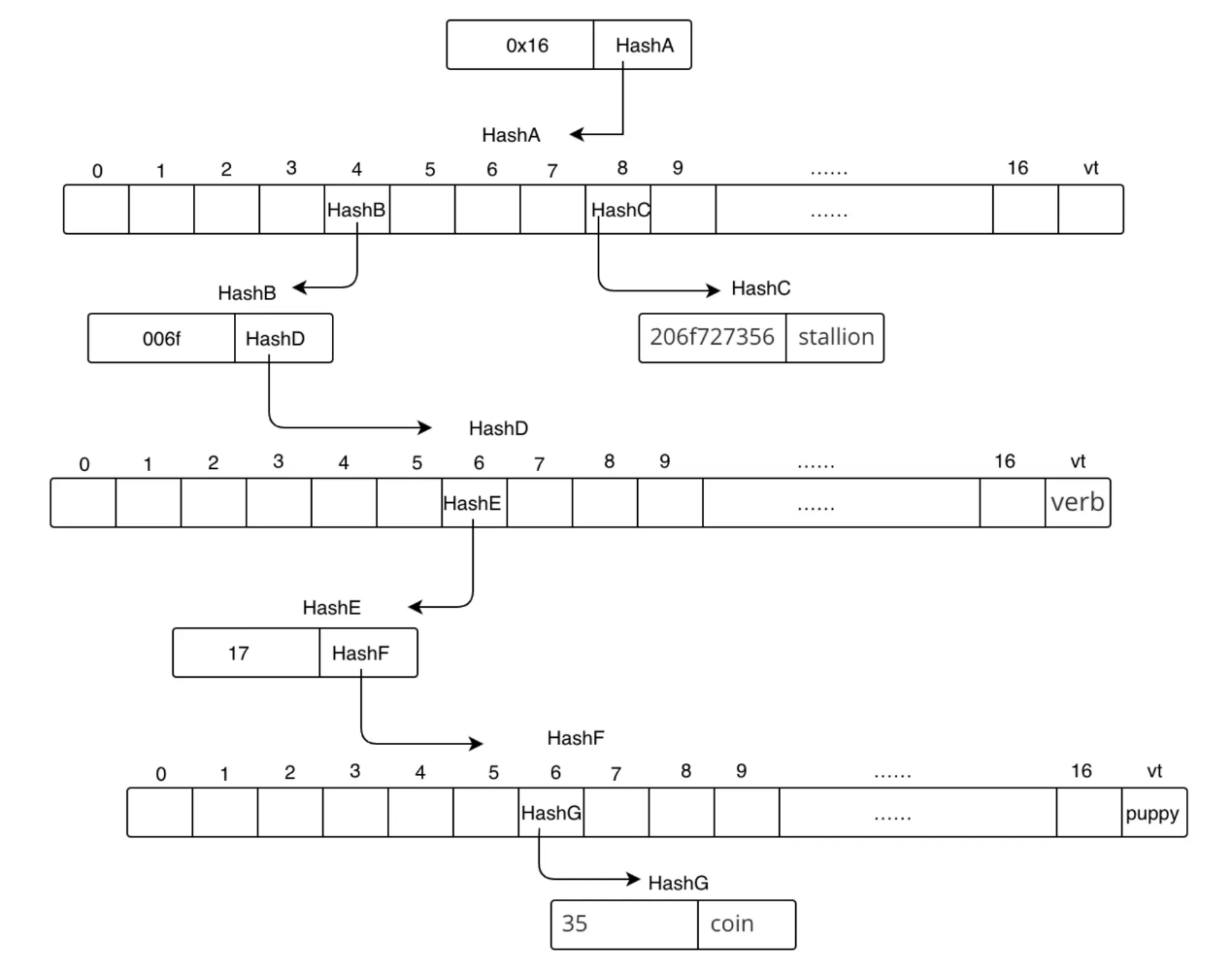


7、7后分出了null和6，是多条路径，与第五步相同，HashF是一个branch节点，索引为6的元素存储子节点哈希，vt存储'puppy'的值。所以当前默克尔树如下图：



8、好了，现在只剩下一条路径了，表示这最后一个是一个leaf叶子节点，路径为5，路径长度为奇数，索引HP前缀为0x3，HP编码为0x35。

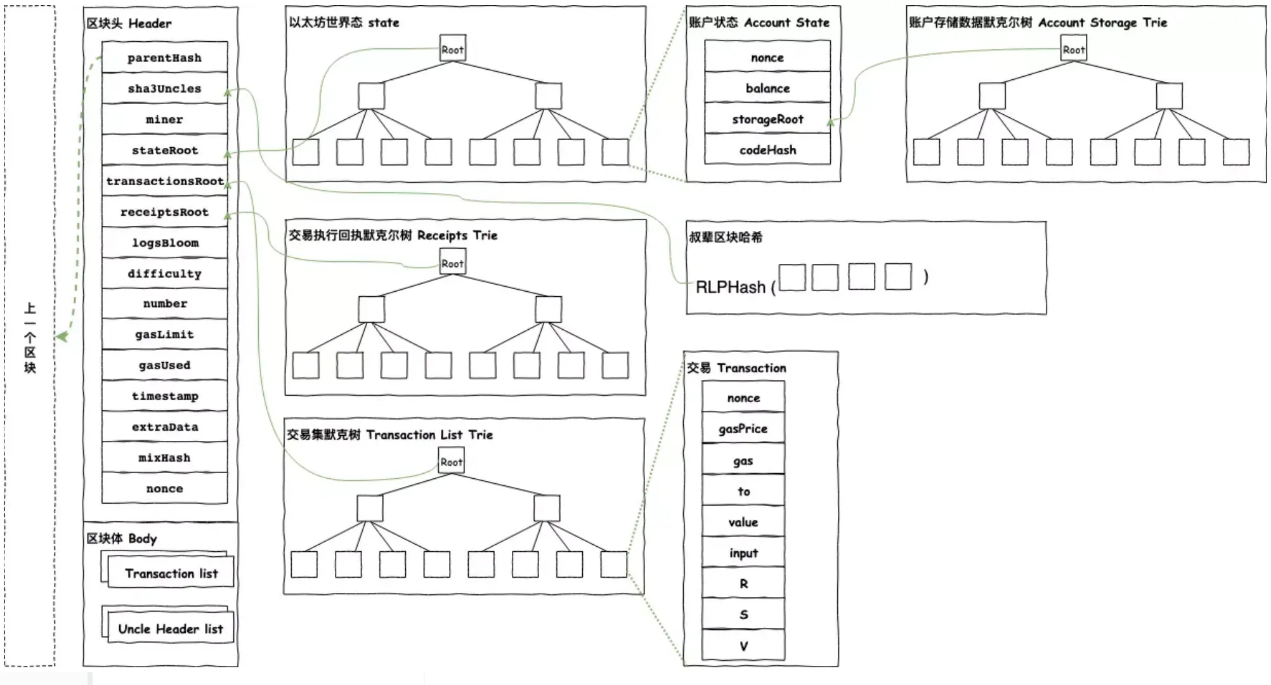
所以当前也是最终的默克尔树如下图：



三、总结

从构造过程中我们可以看出，MPT中节点之间，是通过哈希值来确定的。由于哈希值的特性，只要数据有了微小改动，就会导致根节点改变，所以我们可以用树的根节点来代表整个树中数据的状态，这样就不用保存整个树的数据。

在以太坊中，默克尔树有着大量的应用，比如保存和验证系统中的所有账户状态、所有合约的存储状态、区块中的所有交易和所有收据数据的状态等。



参考：<https://learnblockchain.cn/article/5321>