数学基础（椭圆曲线加密）

椭圆曲线加密

6. EC ElGamal——原理

1. 密钥生成过程：

选一条椭圆曲线，得$E_p(a,b)$，将明文消息m嵌入到曲线上的点P_m，再对点P_m做加密变换.
取$E_p(a,b)$的一个生成元G， $E_p(a,b)$和G作为公开参数。 用户A选

n_A作为秘密钥，以$P_A = n_A G$作为公开钥. (1)

2. 加密过程：

用户B向A发送消息P_m，选取一个随机正整数k，产生以下点对作为密文：

$$C_m = \{kG, P_m + kP_A\}. \quad (2)$$

3. 解密过程：

以密文点对中的第二个点减去用自己的秘密钥与第一个点的倍乘，即

$$P_m + kP_A - n_A kG = P_m + k(n_A G) - n_A kG = P_m. \quad (3)$$

椭圆曲线签名ECDSA

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。

第一步，你需要知道签名本身是40字节，由各20字节的两个值来进行表示，第一个值叫作 R，第二个叫作 S。值对 (R, S) 放到一起就是你的 `ECDSA` 签名。

然后来看看为了进行签名如何创建这一值对：

- 产生一个随机数 k，20字节
- 利用点乘法计算 $P = k \times G$
- 点 P 的 x 坐标即为 R
- 利用SHA1计算信息的哈希，得到一个20字节的巨大的整数 z
- 利用方程 $S = k^{-1}(z + dA \times R) \mod p$ 计算 S

其中 k 是用来生成 R 的随机数，k^{-1} 是 k 的**模的乘法逆元**。

验证过程（dA=私钥，Qa=公钥=dA*G）

$$P = S^{-1} \times z \times G + S^{-1} \times R \times Qa$$

由于 $Qa = dA \times G$，代入上式，则有：

$$P = S^{-1} \times z \times G + S^{-1} \times R \times dA \times G = S^{-1}(z + dA \times R) \times G$$

再由点 P 的 x 坐标必须与 R 匹配，且 R 是点 $k \times P$ 的 x 坐标，即 $P = k \times$ 于是有：

$$k \times G = S^{-1}(z + dA \times R) \times G$$

两边将 G 拿掉，有：

$$k = S^{-1}(z + dA \times R)$$

两边求逆，即：

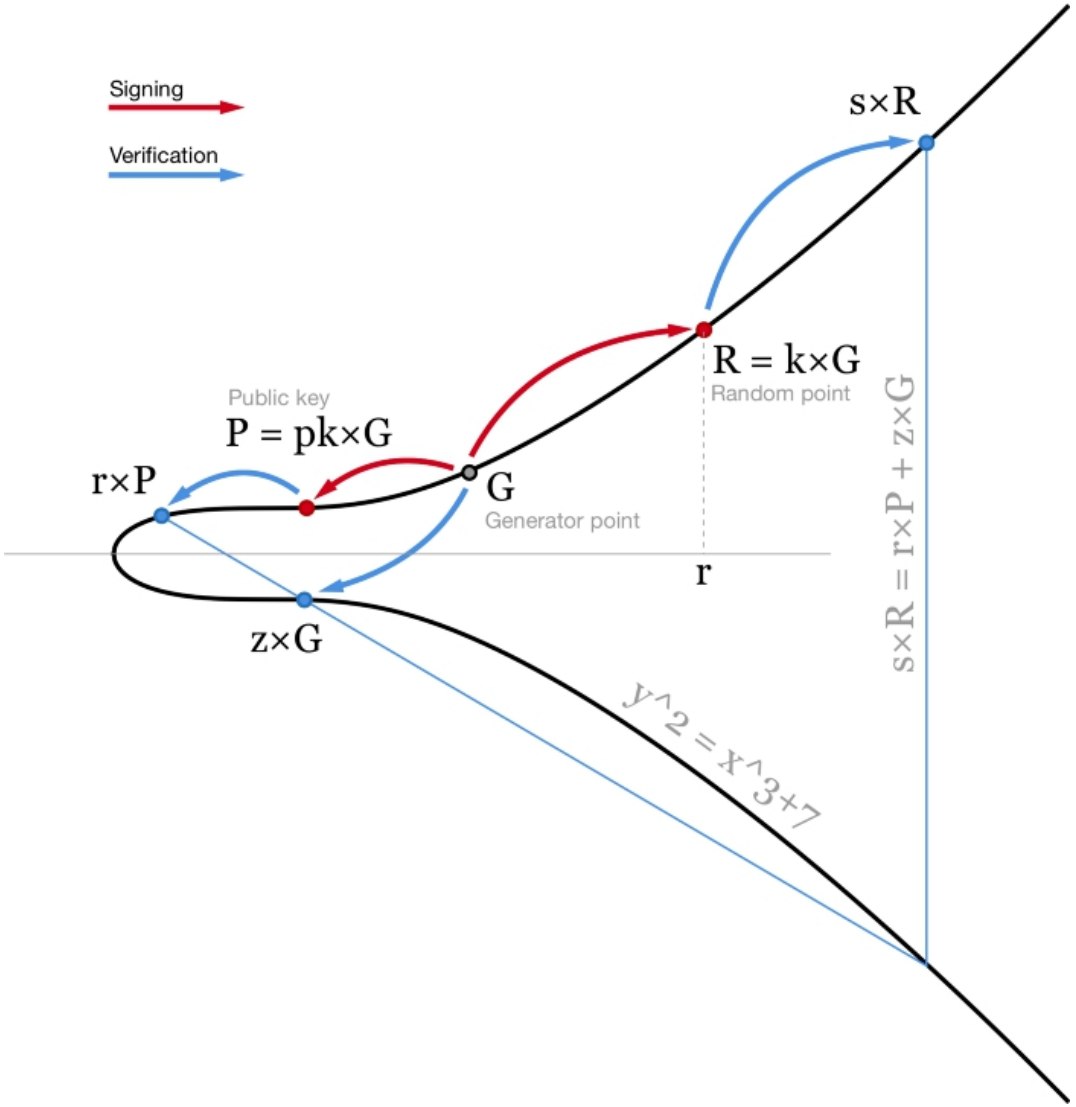
$$S = k^{-1}(z + dA \times R)$$

椭圆曲线签名ECDSA图解

在签名一条消息 mm 时，我们先哈希这条消息，得出一个哈希值，即 $z=hash(m)$z=hash(m) 。我们也需要一个随机数（或者至少看似随机的数）kk 。在这里，我们不希望信任随机数生成器（有太多的错误和漏洞都与不合格的随机数生成器有关），所以我们通常使用 RFC6979，基于我们所知的一个秘密值和我们要签名的消息，计算出一个**确定性的 k**。

使用私钥 pkpk ，我们可以为消息 mm 生成一个签名，签名由两个数组成：rr（随机点 $R=k*G$R=k∗G 的 x 坐标）和 $s=(z+r*pk)/k$s=(z+r∗pk)/k。

然后，使用我们的公钥 $P=pk*G$P=pk∗G ，任何人都可以验证我们的签名，也就是检查 $(z/s)\times G+(r/s)\times P$ (z/s)×G+(r/s)×P 的 x 坐标确为 r。



Sony ps3 破解门

话又说回来，算法的安全是基于其实现的。确保随机数 k 确实是随机产生的变得非常重要，并且没有人能够猜测、计算或者其它任何类型的攻击来得到随机数。但是索尼在它们的实现当中犯了一个巨大的错误，它们在任何地方都采用了同一个随机数，然后它们将具有相同的 R，这意味着你可以使用两个分别具有散列 z 和 z' 和签名 S 和 S' 的两个文件的 S 签名来计算随机数 k：

$$S - S' = k^{-1}(z + dA \times R) - k^{-1}(z' + dA \times R) = k^{-1}(z + dA \times R - z' - dA \times R) = k^{-1}(z - z')$$

于是有：

$$k = \frac{z - z'}{S - S'}$$

一旦你知道了随机数 k，求解 S 的方程就变成了一个只含一个未知数的方程，然后可以很容易地解出 dA：

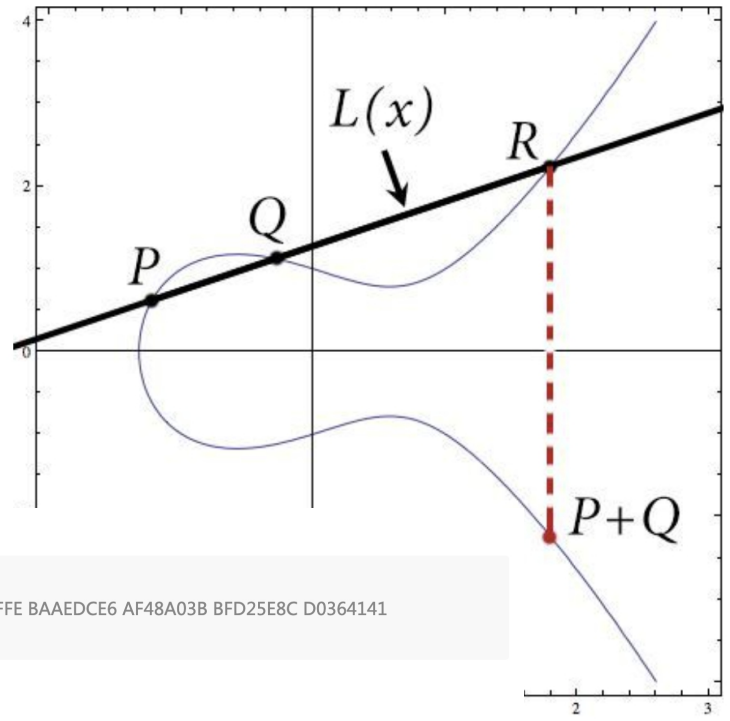
$$dA = \left(S \times k - z\right) / R$$

secp256k1

Secp256k1椭圆曲线形如:

$$y2 = x3 + ax + b$$

椭圆曲线域参数由单元T =（p，a，b，G，n，h）指定



最后，G的阶为：

- n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

协因子：

h = 01

- p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
- =

$$2^{256} - 2^{32} - 2^{9} - 2^{8} - 2^{7} - 2^{6} - 2^{4} - 1$$

Fp上的曲线 E:

$$y2 = x3 + ax + b$$

由下式定义：

- a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
- b = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

压缩形式的基点G是：

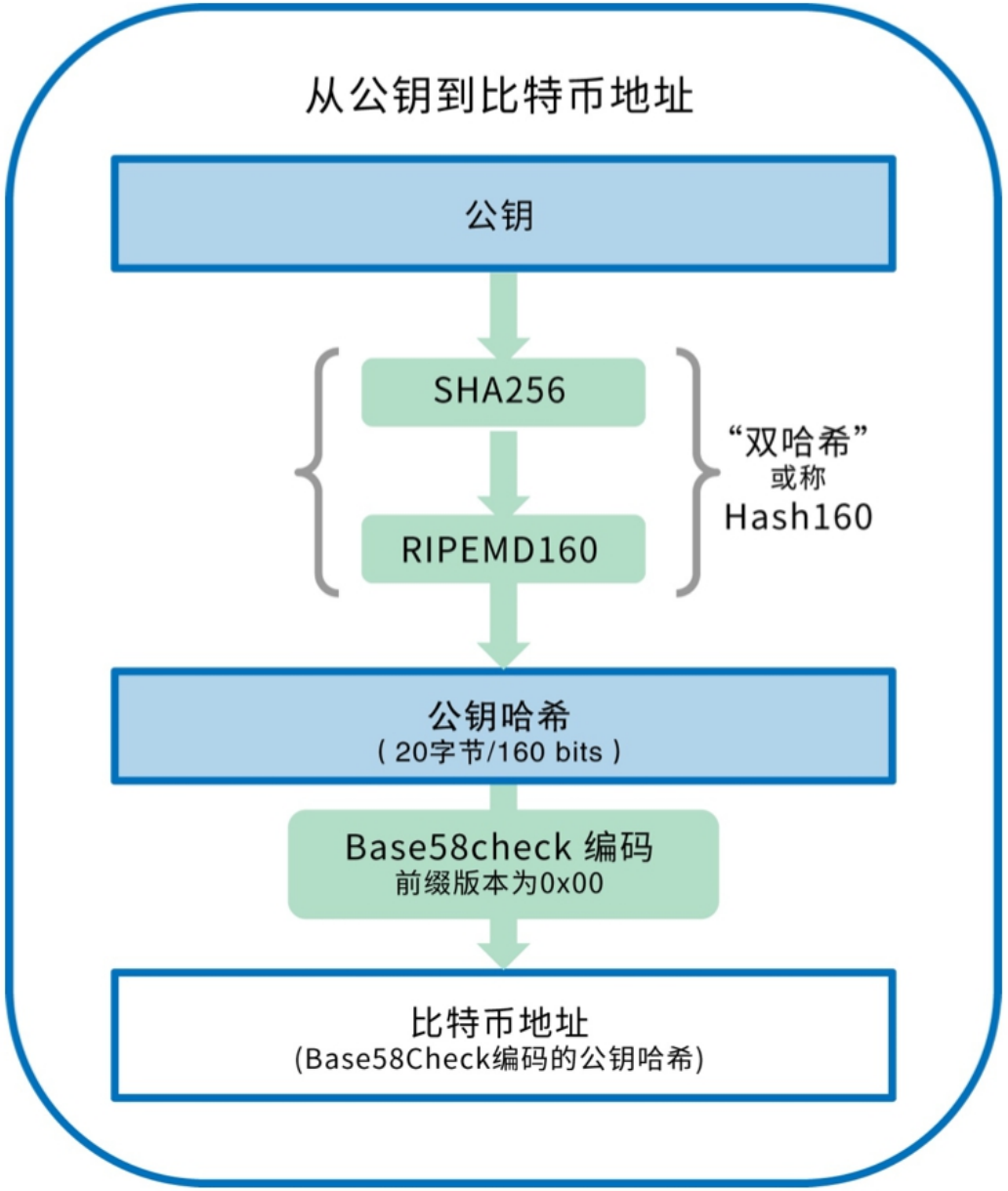
- G = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798

G在未压缩的形式是：

- G= 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

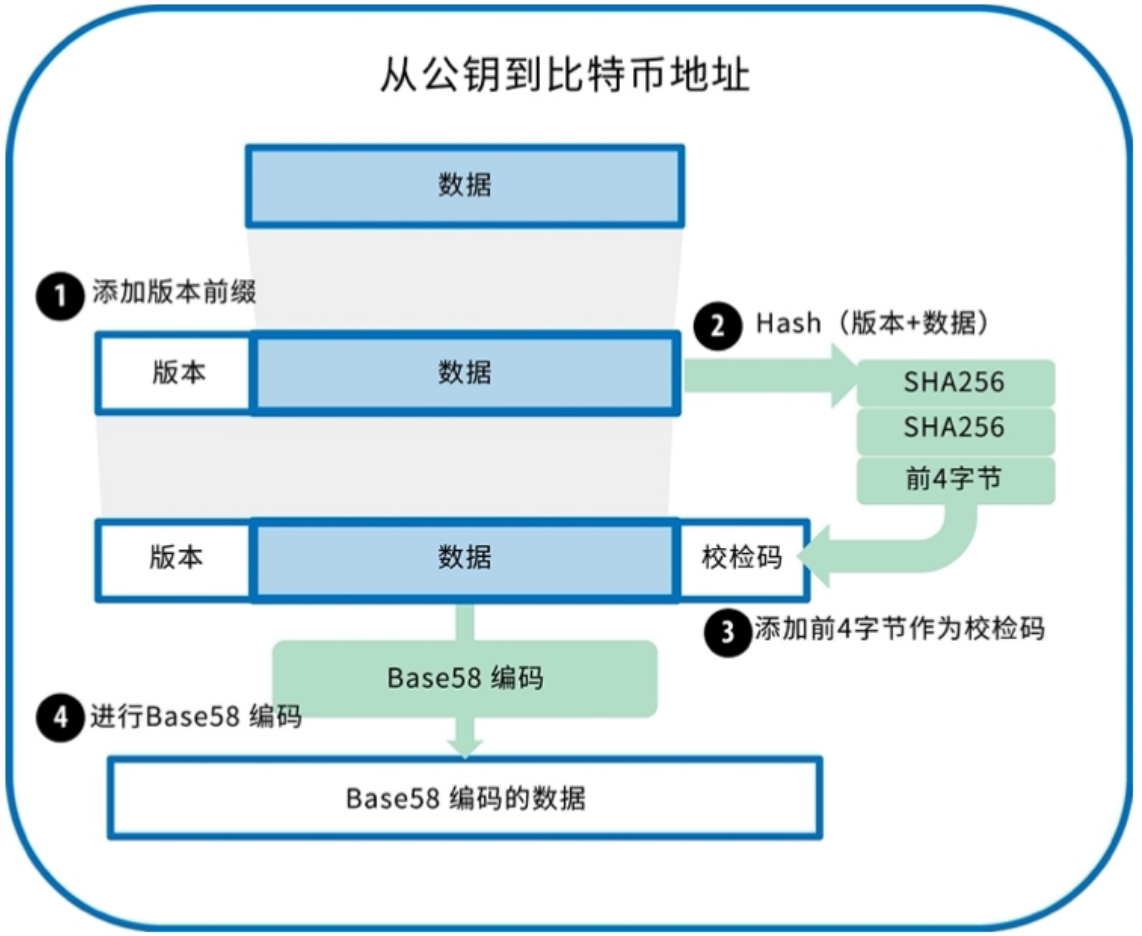
公钥到地址

通常用户见到的比特币地址是经过“Base58Check”编码的，这种编码使用 了58个字符(一种Base58数字系统)和校验码，提高了可读性、避免歧义并有效防止了在地址转录和输入中产生的错 误。Base58Check编码也被用于比特币的其它地方，例如比特币地址、私钥、加密的密钥和脚本哈希中，用来提高可读性和 录入的正确性。



Base58/Base58check编码

为了更简洁方便地表示长串的数字，许多计算机系统会使用一种以数字和字母组成的大于十进制的表示法。例如，传统的十 进制计数系统使用0-9十个数字，而十六进制系统使用了额外的 A-F 六个字母。一个同样的数字，它的十六进制表示就会比十 进制表示更短。更进一步，Base64使用了26个小写字母、26个大写字母、10个数字以及两个符号(例如“+”和“/”)，用于在 电子邮件这样的基于文本的媒介中传输二进制数据。Base64通常用于编码邮件中的附件。Base58是一种基于文本的二进制 编码格式，用在比特币和其它的加密货币中。这种编码格式不仅实现了数据压缩，保持了易读性，还具有错误诊断功 能。Base58是Base64编码格式的子集，同样使用大小写字母和10个数字，但舍弃了一些容易错读和在特定字体中容易混淆 的字符。具体地，Base58不含Base64中的0(数字0)、O(大写字母o)、l(小写字母L)、I(大写字母i)，以 及“+”和“/”两个字符。简而言之，Base58就是由不包括(0，O，l，I)的大小写字母和数字组成。



私钥的格式

表4-2 私钥表示法（编码格式）

种类	版本	描述
Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding: Base58 with version prefix of 128 and 32-bit checksum
WIF-compressed	K or L	As above, with added suffix 0x01 before encoding

表4-3 示例：同样的私钥，不同的格式

格式	私钥
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

私钥/公钥的压缩和非压缩形式

地址格式	地址形式	交易手续费	编码	支持的钱包	支持的交易所
Legacy (P2PKH)	"1"开头	高	Base58	多数	多数
Nested SegWit (P2SH)	"3"开头	中	Base58	多数	多数
Native SegWit	"bc1"开头	低	Bech32	少数	少数

压缩形式

该 *压缩的* 形式只给 *x*，你就应该解决 *y*。在 *未压缩的* 形式为您提供了 *x* 和 *y*。但是，这些数字是略微编码的。在压缩形式中，字符串以“o2”或“o3”开头，字符串的其余部分是 *x* 的十六进制表示 。将满足 *y* 的两个值

$$y^2 = x^3 + 7 \bmod p$$

并且“o2”或“03”告诉您选择哪一个。如果压缩形式以02开头，则选择最低有效位为偶数的根。如果压缩形式从03开始，则选择其最低有效位为奇数的根。（两个根将添加到 *p*，而 *p*是奇数，因此其中一个根将是偶数，一个将是奇数。）

未压缩的形式

未压缩的表单将始终以04开头。在此之后，按照 *x* 和 *y* 连接在一起的十六进制表示形式 。

无论哪种情况，我们都有

x = 79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798

和

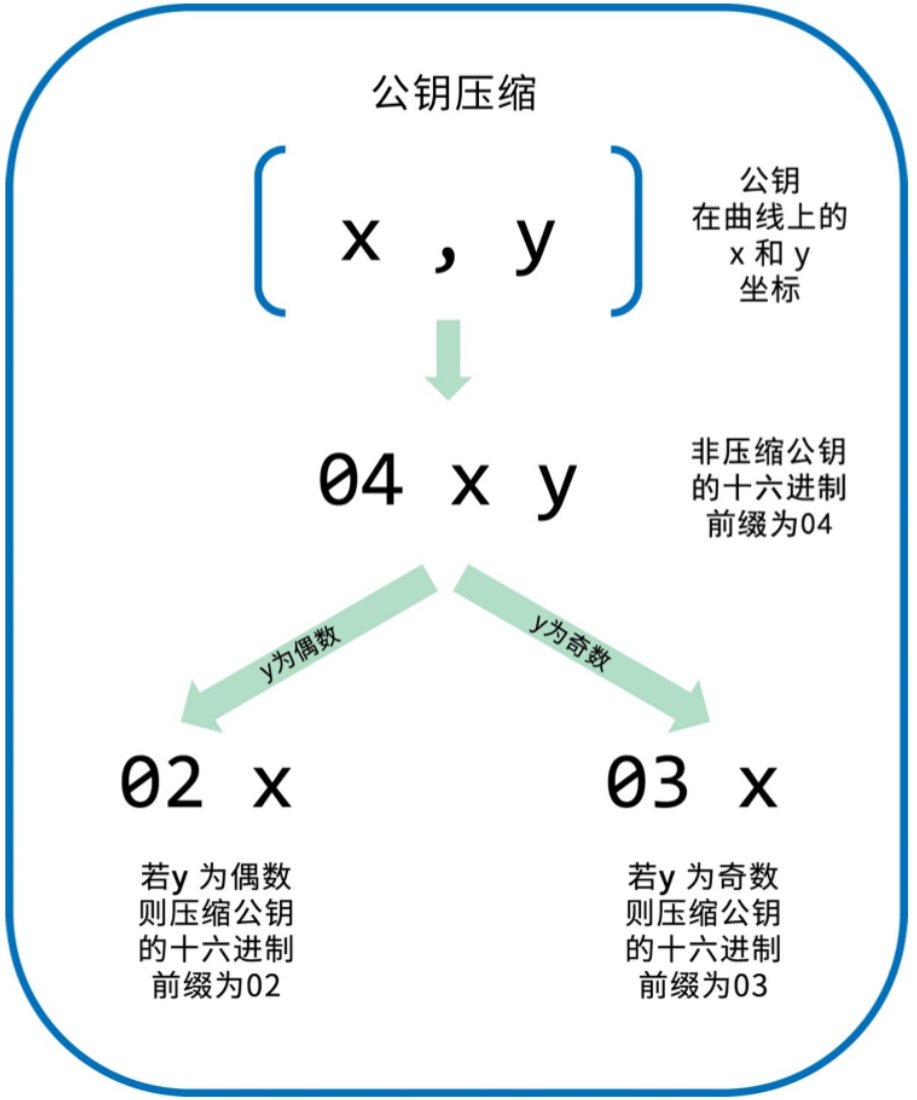
y = 483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8

公钥的压缩和非压缩形式

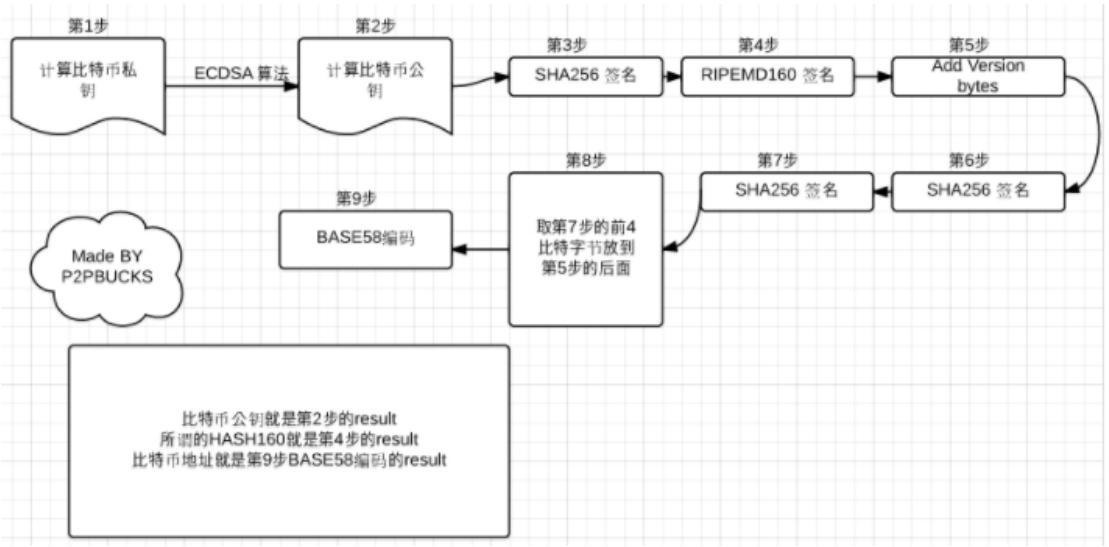
未压缩格式公钥使用04作为前缀，而压缩格式公钥是以02或03作为前缀。需要这两种不同前缀的原因是:因为椭圆曲线加密 的公式的左边是y2 ，也就是说y的解是来自于一个平方根，可能是正值也可能是负值。更形象地说，y坐标可能在x坐标轴的 上面或者下面。从图4-2的椭圆曲线图中可以看出，曲线是对称的，从x轴看就像对称的镜子两面。因此，如果我们略去y坐 标，就必须储存y的符号(正值或者负值)。换句话说，对于给定的x值，我们需要知道y值在x轴的上面还是下面，因为它们 代表椭圆曲线上不同的点，即不同的公钥。当我们在素数p阶的有限域上使用二进制算术计算椭圆曲线的时候，y坐标可能是 奇数或者偶数，分别对应前面所讲的y值的正负符号。因此，为了区分y坐标的两种可能值，我们在生成压缩格式公钥时，如 果y是偶数，则使用02作为前缀;如果y是奇数，则使用03作为前缀。这样就可以根据公钥中给定的x值，正确推导出对应的y 坐标，从而将公钥解压缩为在椭圆曲线上的完整的点坐标。

压缩格式公钥对应着同样的一个私钥，这意味它是由同样的私钥所生成。但是压缩格式公钥和非压缩格式公钥差别很 大。更重要的是，如果我们使用双哈希函数(RIPEMD160(SHA256(K)))将压缩格式公钥转化成比特币地址，得到的地址将会 不同于由非压缩格式公钥产生的地址。这种结果会让人迷惑，因为一个私钥可以生成两种不同格式的公钥——压缩格式和非 压缩格式，而这两种格式的公钥可以生成两个不同的比特币地址。但是，这两个不同的比特币地址的私钥是一样的。

压缩格式公钥渐渐成为了各种不同的比特币客戶端的默认格式，它可以大大减少交易所需的字节数，同时也让存储区块链所 需的磁盘空间变小。然而，并非所有的客戶端都支持压缩格式公钥，于是那些较新的支持压缩格式公钥的客戶端就不得不考 虑如何处理那些来自较老的不支持压缩格式公钥的客戶端的交易。这在钱包应用导入另一个钱包应用的私钥的时候就会变得 尤其重要，因为新钱包需要扫描区块链并找到所有与这些被导入私钥相关的交易。比特币钱包应该扫描哪个比特币地址呢? 新客戶端不知道应该使用哪个公钥:因为不论是通过压缩的公钥产生的比特币地址，还是通过非压缩的公钥产生的地址，两 个都是合法的比特币地址，都可以被私钥正确签名，但是他们是完全不同的比特币地址。



V神的python bitcoin，演示私钥-公钥生成机制




```
>>> from cryptos import *
>>> c=Bitcoin(testnet=True)
>>> priv=sha256('a big big big hello world')
>>> priv
'a79fc536a174ab691079d1d477f4173e0843f2401a49b89c027d62937babd5b1'
>>> len(priv)
64
>>> pub=c.privtopub(priv)
>>> len(pub)
130
>>> pub
'0477400f136523e8d73022f4e16def8707c34a7679139c52dd5ddcb72e99673ba8adea417f8b0d
7de5587db4c27d7b0076f4646227d164758c1a4fc2629435fe30'
>>> x=pub[2:66]
>>> x
'77400f136523e8d73022f4e16def8707c34a7679139c52dd5ddcb72e99673ba8'
>>> len(x)
64
>>> pubx=0x77400f136523e8d73022f4e16def8707c34a7679139c52dd5ddcb72e99673ba8
>>> y=pub[66:130]
>>> y
'adea417f8b0d7de5587db4c27d7b0076f4646227d164758c1a4fc2629435fe30'
>>> len(y)
64
>>> puby=0xadea417f8b0d7de5587db4c27d7b0076f4646227d164758c1a4fc2629435fe30
>>> p=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
>>> assert((puby*puby-pubx*pubx*pubx-7)%p == 0)
>>> addr=c.pubtoaddr(pub)
>>> addr
'mo4PtfHBsUeegcBbfbVzLPg9rPVYBg8XfT'
>>> len(addr)
34
```