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

山东大学网络空间安全学院

周 宇 帆　　202100460108

目录

1 相关知识

1.1 椭圆曲线

域 k(特征 0) 上的椭圆曲线可看成由下面方程的解全体再加上一个无穷远点：$y^2 = x^3 + ax + b, (x, y) \in k^2$，a,b 为常数，并且判别式 $\Delta = -16(4a^3 + 27b^2) \neq 0$ 不等于 0。(即为了光滑性要求无重根)。

椭圆曲线上的点全体构成一个加法群，点与点之间的“加法”运算。正因为椭圆曲线存在加法结构，所以它包含了很多重要的数论信息。

对于密码学，椭圆曲线是连续的，并不适合用于加密。因此，椭圆曲线密码学的第一要务就是把椭圆曲线定义在有限域上，（有限域 F_p ，p 为素数），并提出一条适于加密的曲线：$y^2 = x^3 + ax + b \ (mod\ p)$。

椭圆曲线通常与离散对数问题关联。椭圆曲线上的离散对数问题是指，对于一个曲线上的点 $P = k * G$，G 为基点。已知 P 和 G，计算 k 是困难的，这就引出了公钥密码学的应用。

相比起在商业中被广泛采用的 RSA 加密算法，ECC 优势是可以使用更短的密钥，来实现与 RSA 相当或更高级别的安全。通过下图我们清楚的发现，160 位 ECC 加密安全性相当于 1024 位 RSA 加密，而 210 位 ECC 加密安全性甚至相当于 2048 位 RSA 加密。

1.2 ECDSA

即椭圆曲线数字签名算法，是 DSA 算法在椭圆曲线上的变形应用。

最原始的算法过程如下：

1.2.1 公私钥生成

随机取整数 $d_A \in [1, n-1]$ 作为私钥。n 为椭圆曲线群的阶。

计算 $Q = d_A * G$ 作为公钥，G 为基点。

1.2.2 计算签名

生成随机数 k。

计算 $P = k * G$，得到曲线上一个点

取 P 的坐标 x，令 $r = x \ (mod\ n)$

计算消息 m 的 hash 值 H(m)

计算 $s = k^{-1} * (H(m) + d_A * r)\ (mod\ n)$

输出 (r, s) 作为签名。

验证者需要得到签名对应的消息和公钥才可以验证签名合法性。

2 ESDSA 签名恢复原理

根据 ECDSA 签名算法中 s 的计算公式，利用适当的等式变形可以推导出公钥 P 的表达式：

$$s = k^{-1}(e + dr)$$

$$skG = eG + drG$$

$$sR = eG + rP$$

$$P = r^{-1}(sR - eG)$$

那么问题则从公钥表达式转移至还原 R：

在 ECDSA 签名算法中，r 等于点 R 的横坐标；已知 r，且点 R 在椭圆曲线：

$$y^2 = x^3 + Ax + B$$

上，我们可以通过横坐标 r 计算点 R 的纵坐标 y，即求解二次剩余；

利用 Tonelli Shanks 算法，我们能够得到两个 y 值，即可计算点 R 与点 R'，且 R 与 R' 关于 x 轴对称。

显然，R 与 R' 中，只有一个点与 ECDSA 签名时使用的 R 点相同。

那么问题从还原 R 转移至选择 R 点：

若得到签名 (r, s) 的同时得到签名前缀 v，且 v 对于选择哪个 R 点有明确的指向性，那么就可以轻松地恢复 R，并带入 式即可还原公钥 P；

参考的资料中，对于签名前缀 v 没有具体的介绍，但如果签名前缀的作用仅是对恢复点 R 有明确指向性，那设计签名前缀 v 的算法是容易的：

首先，当 Tonelli Shanks 算法的输入一定时，它的输出即是固定的；

那么我们在 ECDSA 签名算法中，可以将 r 值代入 Tonelli Shanks 算法，判断一下关系式：

$$tonellishanks(r, P) == R_y$$

并设置一定的 v 值，使 v 与关系式是否成立有明确的联系；

恢复公钥时，一样是利用 Tonelli Shanks 算法计算纵坐标，并联系 v 值，选择：

1.Tonelli Shanks 算法的输出 y 所计算的点 R:(x, y)；

2. 选择另一个点 R':(x, P - y)。

3 ESDSA 签名恢复的过程

3.1 产生密钥 GenKey

选择一条椭圆曲线 $E_P(a, b)$，选择基点 G，G 的阶数为 n

选择随机数 $d \in n$ 为私钥，计算公钥 Q = d G

3.2 签名算法 Sign

对消息 m 使用消息摘要算法，得到 z=hash(m)

生成随机数 $k \in n$，计算点 (x, y)=k G

取 r=x mod n，若 r=0 则重新选择随机数 k

计算 $s = k^{-1}(z + rd)modn$，若 s=0 则重新选择随机数 k

上述 (r,s) 即为 ECDSA 签名

3.3 验证算法 Verify

使用公钥 Q 和消息 m，对签名 (r,s) 进行验证。

验证 $r, s \in n$

计算 $z = hash(m)$

计算 $u_1 = zs^{-1}modn$ $u_2 = rs^{-1}modn$

计算 $(x, y) = u1\ G + u2\ Qmodn$

判断 r == x，若相等则签名验证成功

3.4 恢复算法 Recover

已知消息 m 和签名 (r,s)，恢复计算出公钥 Q。

验证 $r, s \in n$

计算 R=(x, y)，其中 x=r,r+n,r+2n⋯，代入椭圆曲线方程计算获得 R

计算 z = hash(m)

计算 $u_1 = -zr^{-1} modn$ $u_2 = sr^{-1} modn$

计算公钥 $Q = (x', y') = u_1\, G + u_2\, R$

4 以太网公钥恢复的意义

在区块链系统中，客户端对每笔交易进行签名，节点对交易签名进行验证。如果采用「验证算法 Verify」，那节点必须首先知道签发该交易所对应的公钥，因此需要在每笔交易中携带公钥，这需要消耗很大带宽和存储。如果采用「恢复算法 Recover」，并且在生成的签名中携带 recoveryID，就可以快速恢复出签发该交易对应的公钥，根据公钥计算出用户地址，然后在用户地址空间执行相应操作。

这里潜藏了一个区块链设计哲学，区块链上的资源（资产、合约）都是归属某个用户的，如果能够构造出符合该用户地址的签名，等同于掌握了该用户的私钥，因此节点无需事先确定用户公钥，仅从签名恢复出公钥，进而计算出用户地址，就可以执行这个用户地址空间的相应操作。

5 代码演示

文件中提供了 1 份 python 代码，展示了通过 ESDSA 签名恢复公钥。运行结果如下：

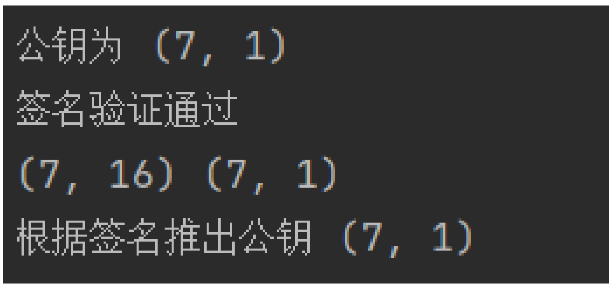

Figure 1: 运行结果