ECDSA 签名机制在区块链领域中的应用

longcpp

longcpp9@gmail.com

July 22, 2019

1 椭圆曲线 secp256k1 与 secp256r1

由于 Bitcoin 中的采纳, 曾经未曾得到广泛部署的椭圆曲线 secp256k1 成为了大多数区块链项目中默认的椭圆曲线选择. 曲线 secp256k1 的名字来自于密码学标准文档 SEC2[1], 其中 “sec” 是 “Standards For Efficient Cryptography” 缩写, “p” 表示椭圆曲线参数定义在有限域 $\mathbb{F}_p$ 上, “256” 表示该有限域中元素的比特长度为 256, “k” 表示这是一条 Koblitz 曲线, 而 “1” 表示这是满足前述条件的第一条 (实际上也是唯一的) 推荐的曲线. Koblitz 曲线在密码学文献中通常指代定义在特征为 2 的有限域上 $\mathbb{F}_{2^m}, m \in \mathbb{Z}$ 的椭圆曲线, Gallant, Lamber 和 Vanstone1 在 CRYPTO 2001 的论文[2] 中泛化了 Koblitz 曲线的含义, 也包括定义在大素数上 $\mathbb{F}_p$ 上具备高效可计算自同态特性的椭圆曲线.

Satoshi 在最开始选择 secp256k1 曲线的原因仍不可知, 尤其是在当时得到广泛部署的是一条名为 secp256r1 椭圆曲线的背景之下. 选用 secp256k1 的原因可能是该曲线具备的自同态映射可以加速 ECDSA 签名验证过程的特性在区块链场景中尤为合适, 但是以 OpenSSL 为代表的各个密码学库的实现中并没有利用这一属性. 虽然 libsecp256k1 中的实现成功利用这一属性使得基于 secp256k1 的 ECDSA 签名验证速度达到了 21000 次每秒 (测试平台的芯片型号为 Intel(R) Core(TM) i7-6700HQ CPU), 速度上超过了 OpenSSL 1.1 版本中深度优化的基于 secp256r1 曲线的 ECDSA 的 12000 次每秒的验签速度, 最终证实了 secp256k1 曲线在 ECDSA 验签操作中的效率优势, 但是在选定这条曲线时, 并没有相应的实现可以证实关于验签效率的推断.

[1] Cerrticom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters. 2010. https://www.secg.org/sec2-v2.pdf

[2] Gallant, Robert P., Robert J. Lambert, and Scott A. Vanstone. ”Faster point multiplication on elliptic curves with efficient endomorphisms.” In Annual International Cryptology Conference, pp. 190-200. Springer, Berlin, Heidelberg, 2001. https://www.iacr.org/archive/crypto2001/21390189.pdf

然而后来的斯诺登泄露的文档中显示的 NSA 可能在 NIST 标准中的埋藏算法级后门的信息, 尤其经过 Dual_EC_DRBG 事件[3] 验证之后, Satoshi 当初的曲线选择在后来看来有了先见之明的意味. 得到广泛部署的 secp256r1 曲线中的 “r” 表示曲线参数是从随机种子派生而来. secp256r1 (NIST P-256) 曲线的参数是从随机种子

$$c49d360886e704936a6678e1139d26b7819f7e90$$

中派生而来, 而该随机种子的来源 NIST 并没有解释, 鉴于 Dual_EC_DRBG 事件的教训, 难免会有其中存在后门的疑虑 [4]. 相比之下, secp256k1 曲线的参数选择有合理的解释, 也就有助于消除对存在算法级后门的担忧 [5]. 后续介绍基于 secp256k1 的 ECDSA 签名机制在区块链领域中的应用以及在区块链场景下的面临的特殊问题.

2 基于椭圆曲线 secp256k1 的 ECDSA 签名

定义在有限域 $\mathbb{F}_p$ 上的曲线 secp256k1 的方程为 $y^2 = x^3 + 7$, 其中

$$p = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f.$$

椭圆曲线上的点的个数为 $\#E(\mathbb{F}_p) = h \cdot n$, 其中 $h = 1$ 为余因子 (Cofactor), n 为 $E(\mathbb{F}_p)$ 的最大素子群的阶:

$$n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141.$$

子群 $\mathbb{G} = \langle G \rangle$ 的基点 G 的坐标为:

$$G_x = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798$$

$$G_y = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8$$

值得提及的是, $\mathbb{G}$ 的阶 n 是素数, 也即 $\mathbb{F}_n$ 是有限域, 非零元构成的乘法群表示为 $\mathbb{F}_n^*$.

假设待签名消息为 m, 私钥为 d, 公钥为 $P = dG$, 哈希算法为 $H : \{0,1\}^* \to \mathbb{F}_n^*$. **ECDSA 签名值** $\sigma = (r, s), r, s \in \mathbb{F}_n^*$ **的计算过程**为:

1. 选择随机数 $k \in_R \mathbb{F}_n^*$, 计算 $R = (x, y) = kG, x, y \in \mathbb{F}_p$, 计算 $r = x \mod n \in \mathbb{F}_n^*$,
2. 计算消息 m 的哈希值 $e = H(m) \in \mathbb{F}_n^*$, 计算 $s = k^{-1}(e + rd) \mod n$.

注意 $k, r, s \in \mathbb{F}_n^*$, 也即 k, r, s 均不得为 0, 如果为 0, 则重新选择 k 进行计算.

给定消息 m, 公钥 P, 签名值 $\sigma = (r, s)$, **签名验证的过程**为:

[3] Bernstein, D.J., Lange, T. and Niederhagen, R., 2016. Dual EC: A standardized back door. In The New Codebreakers (pp. 256-281). Springer, Berlin, Heidelberg. https://projectbullrun.org/dual-ec/documents/dual-ec-20150731.pdf

[4] SafeCurves:choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to/rigid.html

[5] NSA and ECC. https://bitcointalk.org/index.php?topic=289795.msg3183975#msg3183975

1. 验证 r, s 确实是 $\mathbb{F}_n^*$ 中的元素, 也即 $r, s \in [1, n-1]$, 否则签名值无效,

2. 计算哈希值 $e = H(m) \in \mathbb{F}_n^*$, 并计算 $s \in \mathbb{F}_n^*$ 的逆 s^{-1},

3. 计算 $R' = (x', y') = s^{-1}(hG + rP)$,

4. 判断 $R' \neq \mathcal{O}$, 否则验签失败, 判断 $x' \mod n = r$, 相等则验签成功, 否则验签失败.

合法的签名能够验证通过是因为

$$R' = s^{-1}(eG + rP) = s^{-1}(eG + r(dG)) = s^{-1}(e + rd)G = kG = R.$$

上述基于曲线 secp256k1 的 ECDSA 签名机制, 总共涉及 3 种数学结构上的计算: 有限域 $\mathbb{F}_n$ 上加法和乘法运算 (求逆运算可以由加法和乘法运算构造), $E(\mathbb{F}_p)$ 中的加法点群 $\mathbb{G}$ 中点的加法运算 (点的倍乘), 由于 $\mathbb{G}$ 中的点的坐标为有限域 $\mathbb{F}_p$ 中的元素, 则点的加法运算中也涉及到有限域 $\mathbb{F}_p$ 上的加法和乘法运算 (求逆运算). 编码实现 ECDSA 签名机制时, 需注意区分不同的运算, 尤其要注意不要混淆有限域 $\mathbb{F}_n$ 和 $\mathbb{F}_p$ 上的运算.

3 ECDSA 签名机制应用中的安全隐患

ECDSA 签名机制在编码实现和应用时, 特别是在区块链场景中应用时很容易引入安全问题, 列举如下, 随后依次介绍每个安全隐患的原理.

1. 如果 k 值泄露, 则任何知道该随机数值的人可以使用该随机数产生签名值恢复私钥

2. 用相同私钥和 k 对两个消息进行签名, 则任何人都可以通过两个签名值恢复出私钥

3. 两个用户使用相同的 k 分别对不同的消息进行签名, 则任一方可推算出对方的私钥

4. 相同私钥和 k 同时用于 ECDSA 签名和 Schnorr 签名时, 任何人都能够恢复出私钥

5. ECDSA 签名值的可锻造性

6. 签名值通常采用的 DER 编码由于编码值并不唯一也会造成区块链网络的分裂

7. 不需要提供签名消息的情况下, 任何人可以根据任意签名值伪造对应私钥的签名值

3.1 随机数 k 值泄露导致私钥泄露

签名过程中使用的随机数 $k \in \mathbb{F}_n^*$ 一定要及时删除以免泄露. 这是因为知道随机数 k 的任何人, 都可以根据签名值 $\sigma = (r, s)$ 计算出相应的生成该签名值的私钥 d. 随机数 k 的值

与签名值 $\sigma = (r, s)$ 之间的对应关系可以根据等式 $x = (kG)_x \mod n = r$ 来进行判断. 由于 k, r 以及 $e = H(m)$ 已知, 则可以根据下面的公式计算私钥 d:

$$s = k^{-1}(e + rd) \mod n \to ks = (e + rd) \mod n \to d = r^{-1}(ks - e) \mod n.$$

基于上述理由, 在通过随机数发生器 (Random Number Generator, RNG) 生成随机数 k 的时候, 首先要确保选用的 RNG 生成的随机数的质量, 然后要确保生成的随机数没有被记录下来, 还要确认 RNG 实现的安全性 (生成 k 的能量消耗等侧信道信息不会泄露 k 的值), 最后在使用完随机数 k 之后要将其及时删除.

3.2 同一用户重用随机数 k 导致私钥泄露

如果同一个用户使用自己的私钥对不同的消息进行签名时, 使用了相同的 k 值, 则任何人可以根据消息和签名值推算出用户的私钥. 假设用户用私钥 d 和 k 对消息 m_1 和 m_2 进行签名, 生成了签名值 $\sigma_1 = (r, s_1)$ 和 $\sigma_2 = (r, s_2)$, 可以看到两个签名值的 r 部分相同, 这是因为使用了相同的 k, 而 $r = (kG)_x \mod n$, 根据签名计算过程则有:

$$\begin{cases} s_1 = k^{-1}(e_1 + rd) \mod n, & e_1 = H(m_1) \\ s_2 = k^{-1}(e_2 + rd) \mod n, & e_2 = H(m_2) \end{cases}$$

上面的方程组中, 可以看到未知的变量仅有 k 和 d, 两个方程两个未知量, 通过解方程组可以推算出私钥 d. 根据上面的方程组就有

$$\begin{cases} k = s_1^{-1}(e_1 + rd) \mod n \\ k = s_2^{-1}(e_2 + rd) \mod n \end{cases} \to \frac{e_1 + rd}{s_1^{-1}} = \frac{e_2 + rd}{s_2^{-1}} \mod n$$

根据 $s_2(e_1 + rd) = s_1(e_2 + rd) \mod n$, 可以推断出私钥

$$d = \frac{s_2 e_1 - s_1 e_2}{(s_1 - s_2)r} \mod n.$$

3.3 不同用户重用随机数 k 导致私钥泄露

前一小节论述了同一个用户签名时重用随机数 k 的安全隐患, 然而如果两个用户之间重用了随机数 k, 同样也会私钥泄露的隐患. 如果两个用户 Alice 和 Bob 在用自己的私钥签名时, 选用了同样的随机数 k, 则 Alice 可以根据签名值推算出 Bob 的私钥, Bob 也可以根据签名值推算出 Alice 的私钥. 假设 Alice 的私钥为 d_1, 要签名的消息为 m_1, 选用随机数 k 时计算的签名值为 $\sigma_1 = (r, s_1)$, Bob 的私钥为 d_2, 要签名的消息为 m_2, 选用同样随机数 k 时计算的签名值为 $\sigma_2 = (r, s_2)$. 则根据签名值的计算过程有

$$\begin{cases} s_1 = k^{-1}(e_1 + rd_1) \mod n, & e_1 = H(m_1) \\ s_2 = k^{-1}(e_2 + rd_2) \mod n, & e_2 = H(m_2) \end{cases}$$

根据上面的方程组就有

$$\begin{cases} k = & s_1^{-1}(e_1 + rd_1) \mod n \\ k = & s_2^{-1}(e_2 + rd_2) \mod n \end{cases} \rightarrow \frac{e_1 + rd_1}{s_1^{-1}} = \frac{e_2 + rd_2}{s_2^{-1}} \mod n$$

则有 $s_2(e_1 + rd_1) = s_1(e_2 + rd_2) \mod n$. 在 Alice 看来, 等式中未知的变量只有 Bob 的私钥 d_2, Alice 可以计算 Bob 私钥:

$$d_2 = \frac{s_2e_1 - s_1e_2 + s_2rd_1}{s_1r} \mod n,$$

而在 Bob 看来, 等式中未知的变量只有 Alice 的私钥 d_1, Bob 可以计算 Alice 的私钥:

$$d_1 = \frac{s_1e_2 - s_2e_1 + s_1rd_2}{s_2r} \mod n.$$

Bitcoin 历史上由于 3.2 小节和 3.3 小节中介绍的关于随机数 k 的问题引发的多起私钥泄露的事件. 由于安全的随机数发生器实现的困难性与程序员正确使用随机数的困难性, 业界由随机产生 k 逐渐切换为利用 RFC 6979[6]中推荐的方式. RFC 6979 中通过利用待签名消息 m 和私钥 d 等信息给出了一种确定性派生 k 的方式, 通过这种方式, 只要 k 和 m 不同, 在 k 值不同, 也就避免了3.2 小节和 3.3 小节的安全隐患.

3.4 ECDSA 与 Schnorr 共用 k 导致私钥泄露

前面两个小节可以看到, 同一用户重用 k, 不同用户重用 k 都会导致用于签名的私钥泄露的问题. 然而除了用户自己/用户之间重用 k 之外, 如果不同的签名机制在签名时重用了 k 也同样会导致私钥泄露. Bitcoin 和 Bitcoin Cash 网络中正在围绕在链上采纳 Schnorr 签名而做大量的准备活动, 其中 Bitcoin Cash 已经在 19 年 5 月份的网络协议升级中为交易签名过程添加 Schnorr 签名机制.

首先介绍定义在椭圆曲线 secp256k1 上的 Schnorr 签名机制可以与 ECDSA 签名机制共用相同的公私钥对, 也即用于 ECDSA 签名的公私钥对 $d \in \mathbb{F}_n^*, Q = dG \in \mathbb{G}$, 同时也可以是 Schnorr 签名机制的公私钥对. 假设待签名消息为 m, 有哈希函数 $H : \{0,1\}^* \rightarrow \mathbb{F}_n^*$, 则 **Schnorr 签名 $\sigma = (R, s), R \in \mathbb{G}, s \in \mathbb{F}_n^*$ 的计算过程**为:

1. 选取随机数 $k \in_R \mathbb{F}_n^*$, 计算 $R = (x, y) = kG$,
2. 计算哈希值 $e = H(R||P||m) \in \mathbb{F}_n^*$, 计算 $s = r + ed \mod n \in \mathbb{F}_n^*$.

给定消息 m, 公钥 P, Schnorr 签名值 $\sigma = (R, s)$, **Schnorr 签名验证过程**为:

[6] RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). https://tools.ietf.org/html/rfc6979

1. 验证 R 确实是 $\mathbb{G}$ 中的元素, 验证 s 确实是 $\mathbb{F}_n^*$ 中的元素, 否则验签失败,

2. 计算哈希值 $e = H(R||P||m) \in \mathbb{F}_n^*$,

3. 计算 $R + eP$ 并验证是否与 sG 相等, 如果相等验签通过, 否则验签失败.

合法的签名值能够验证通过是因为:

$$sG = (k + ex)G = kG + exG = R + e(dG) = R + eP.$$

值得提及的是 Schnorr 签名还有另外一种变体, 此处仅介绍 Bitcoin 和 Bitcoin Cash 网络中采用的 Schnorr 签名机制, 对于展示 ECDSA 和 Schnorr 签名共用 k 导致私钥泄露的问题来说已经足够.

用 d, P 表示 ECDSA 和 Schnorr 签名所共用的公私钥对, ECDSA 签名用随机数 k 对消息 m_1 的签名值为 $\sigma_1 = (r, s_1)$, Schnorr 签名用相同的随机数对消息 m_2 的签名值为 $\sigma_2 = (R, s_2)$, 下面展示如何通过签名值恢复私钥 d. 根据签名计算规则有

$$\begin{cases} s_1 \equiv & k^{-1}(e_1 + rd) \mod n, & e_1 = H(m_1) \\ s_2 \equiv & k + e_2 d \mod n, & e_2 = H(R||P||m_2) \end{cases}$$

则有 $k = s_2 - e_2 d \mod n$, 带入上述方程组的第一个等式得到

$$s_1 = (s_2 - e_2 d)^{-1}(e_1 + rd) \mod n \to d = \frac{s_1 s_2 - e_1}{s_1 e_2 + r} \mod n.$$

前面我们提到利用 RFC 6979 中的方法避开3.2 小节和 3.3 小节重用 k 的问题, 如何在不同的签名算法的维度上避免重用 k? 答案是同样利用 RFC 6979. RFC6979 中输入参数可以包含 “addition data”, 在派生 k 时, 可以将签名算法的信息填入该字段, 如此可以在算法的维度上重用 k. 可以参考 Bitcoin Cash 网络中支持 Schnorr 签名值的做法:

We suggest using the RFC6979 sec 3.6 'additional data' mechanism, by appending the 16-byte ASCII string "Schnorr+SHA256␣␣" (here ␣ represents 0x20 – ASCII space). The popular library libsecp256k1 supports passing a parameter `algo16` *to* `nonce_function_rfc6979` *for this purpose.* [7]

3.5 ECDSA 签名值的可锻造性问题

给定消息 m 的 ECDSA 签名值 $\sigma = (r, s)$, 任何人都可以构造关于消息 m 的额外的合法签名值, 例如 $\sigma' = (r, -s)$ 也是消息 m 的合法签名值, 也即 σ 和 σ' 均可利用公钥 P 验证通

[7] 2019-MAY-15 Schnorr Signature specification. https://github.com/bitcoincashorg/bitcoincash.org/blob/master/spec/2019-05-15-schnorr.md

过. 这是因为验签是通过计算 $R' = (x', y') = s^{-1}(eG + rP)$ 并判断 $x' \mod n$ 是否等于签名值中的 $r = x \mod n, (x, y) = R = kG$. 然而根据椭圆曲线的性质有 $-R' = (x', -y')$, 也即 R' 和 $-R'$ 具有相同的横坐标. 根据费马小定理有 $s^{-1} \mod n \equiv s^{n-2} \mod n$ 以及有限域 $\mathbb{F}_n$ 上 $-s \equiv (n - s) \mod n$, 根据多项式的二项展开 (Binomial Expansion) 有:

$$\begin{aligned}(-s)^{-1} \mod n &\equiv (n-s)^{n-2} \mod n \\ &\equiv \textstyle\sum_{i=0}^{n-2} \binom{n-2}{i} n^{n-2-i}(-s)^i \mod n \\ &\equiv (-s)^{n-2} \mod n \\ &\equiv (-1)^{n-2} s^{-1} \mod n \\ &\equiv -s^{-1} \mod n.\end{aligned}$$

则验证签名值 $\sigma' = (r, -s)$ 时

$$(-s)^{-1}(eG + rP) = -s^{-1}(eG + rP) = -R' = (x', -y'),$$

由于 $x' \mod n \equiv r \mod n$, 因此 $\sigma' = (r, -s)$ 验签成功.

两个签名值 $\sigma = (r, s)$ 和 $\sigma' = (r, -s)$ 同时为消息 m 的合法签名, 在很多场景中不会造成问题. 然而在区块链场景下, 网络中的任何一方可以利用该性质, 修改一笔交易中的签名值. 由于区块链中的交易 ID 通常是交易的哈希值, 则 ECDSA 签名值的可锻造性会造成同一笔交易在网络上有不同的交易 ID. 交易 ID 的不唯一会导致根据交易 ID 追踪交易状态时出现安全隐患. 假设追踪的是携带签名值 σ 的交易的 ID, 而上链的交易中包含的 σ' (具有不同的交易的 ID), 则在数字货币交易所处理用户的提币操作时, 当一笔提币已经成功时, 根据交易 ID 进行交易状态跟踪的业务逻辑会认为提币操作失败. 倘若还设置了超时重试机制, 会导致交易所资产的损失.

由于 ECDSA 签名值的可锻造性可能带来的安全隐患, BIP-146 中针对 ECDSA 签名值做了额外的固定, 规定验签时签名值 (r, s) 中的 s 的值必须满足 $1 \le s \le n/2$, 根据曲线 secp256k1 的参数有

$$n/2 = 0x7fffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0.$$

由于模运算的性质, 当 $s < n/2 \to (n - s) > n/2$ 并且有 $s = n/2 \to (n - s) = n/2$, 也即 BIP-146 中对 s 取值范围的限定, 使得 $\sigma = (r, s)$ 和 $\sigma' = (r, -s)$ 中仅有一个为合法的签名值, 从而避开 ECDSA 签名值的可锻造性带来的安全隐患.

3.6 ECDSA 签名值 DER 编码的不唯一问题

在用 Go 语言重新实现 Bitcoin Cash 协议的 Copernicus 项目时 [8] 遇到了这样一个问题, Listing 1 中的 DER 编码的 ECDSA 签名用函数 `secp256k1_ecdsa_signature_parse_der`

[8] An alternative implementation of the Bitcoin Cash protocol, written in Golang. https://github.com/copernet/copernicus

解析时 (来自 libsecp256k1) 失败, 但是用另外一个函数 `ecdsa_signature_parse_der_lax` 可以成功解析.

Listing 1: 解析失败的 DER 编码的 ECDSA 签名

```
const char *sig_der =
    "3046"
    "0221"
    "002e6f0e8b515b5f25e837592e5e8a834cbe3fabaf98973edf88b19502e0180c2d"
    "0221"
    "00d03cc64f35fb277fe1b69270b542aca5620394ed7b7fae7a3546934dd6fe4288";
```

函数 `secp256k1_ecdsa_signature_parse_der` 实现内部的逻辑关系在 Figure 1中展示. 跟踪函数执行过程, 可以发现错误发生在对函数 `secp256k1_ecdsa_sig_parse` 的调用, 更具体的说是该函数中调用 `secp256k1_der_parse_integer` 函数尝试对签名值的 r 解析时返回 0 值 (执行失败).

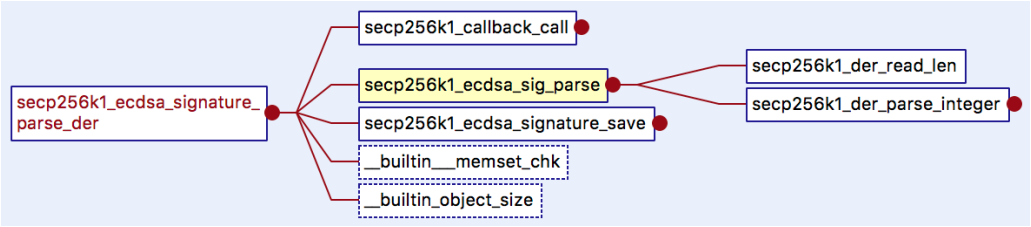

Figure 1: 函数 `secp256k1_ecdsa_signature_parse_der` 内部实现逻辑

继续追踪 `secp256k1_der_parse_integer` 的内部实现, 可以发现导致失败的原因是上述签名值中表示 r 的部分在最开始的多余额全零字节 00, 对应函数内部的代码注释: `/* Excessive 0x00 padding. */`. 如下面代码展示, 删掉多余的全零字节, 并相应调整 DER 编码中指示长度的字段, 则可用函数 `secp256k1_ecdsa_signature_parse_der` 成功解析签名值.

Listing 2: 可成功解析的 DER 编码的 ECDSA 签名

```
const char *sig_der =
    "3045"
    "0220"
    "2e6f0e8b515b5f25e837592e5e8a834cbe3fabaf98973edf88b19502e0180c2d"
    "0221"
    "00d03cc64f35fb277fe1b69270b542aca5620394ed7b7fae7a3546934dd6fe4288";
```

为了明白上述问题的产生原因, 接下来介绍 DER (Distinguished Encoding Rules) 编码规则, DER 编码是 ASN.1 (Abstract Syntax Notation One) 编码中的一种编码规则. ASN.1 中另外支持的编码规则包括 BER (Basic Encoding Rules) 和 CER(Canonical Encoding Rules). 其中 DER 和 CER 编码规则完全指定了编码规则, 也即对于特定的数据仅有一个编码值是合法的. DER 针对数据长度的不同, 有两种方式来编码数据载荷的长度: Short Encoding 和 Long Encoding, 并规定当数据载荷的长度小于 128 个字节时, 应当使用 Short Encoding. 基于 secp256k1 曲线的 ECDSA 签名值中的 r, s 都是 256 比特的整数, 所以采用的是 Short Encoding 的 DER 编码方式.

Bitcoin 中从 OpenSSL 中继承了签名值的 DER 编码方式, 这是因为早期的 Bitcoin 的密码能力通过 OpenSSL 实现, 而 OpenSSL 中 ECDSA 签名操作的返回的是使用 DER 编码后的签名值. 由于签名值中的 r, s 都是证书, 接下来仅考虑对 INTEGER 的编码方式. 对 INTEGER 的编码基本上是以字节为基本单位将整数表达成 256 进制表示, 编码规则同时要求对于正数 (r, s 是正数), 第一个字节的最高位必须为 0, 如果第一个字节的最高位为 1, 则需要在第一个字节之前添加一个全零字节. 另外 DER 编码规则同时规定, 对于所有可能的编码值中, 选用最短的编码值作为合法的签名值:

The Distinguished Encoding Rules specify we must choose the shortest encoding that fully represents the length of the element.The encoded lengths do not include the ASN.1 header or length bytes, simply the payload. [9]

理论或者设想总是美好的, 但是在具体实现时经常出现偏离规则的情况, 正如 OpenSSL 的实现中 DER 解码时候也会接受没有严格遵循前述规则的 DER 编码:

One specifically critical area is the encoding of signatures. Until recently, OpenSSL's releases would accept various deviations from the DER standard and accept signatures as valid. When this changed in OpenSSL 1.0.0p and 1.0.1k, it made some nodes reject the chain. [10]

在区块链场景中会引发的问题, 如果各个节点运行的 OpenSSL 版本不同, 则对于一个没有遵循 DER 规则的签名值编码, 可能会出现某些节点认为包含了这样签名的区块为合法区块, 而另一些节点则将该节点视为非法的区块, 从而导致网络的分裂. 因此, Bitcoin 中在 BIP-66 中对 Bitcoin 网络应该接收的签名的 DER 编码做了更为严格的限制.

另外的问题在于, 签名值的 DER 编码在区块链的场景中是否是必要的? ECDSA 签名值的 DER 编码长度根据具体的签名值, 长度会有所不同, 按照 BIP-66 中的约束, 编码之后的签名值最多可以为 73 个字节, 相比之下直接采用 r, s 的二进制表示的话, 签名值则

[9] 摘录自 St Denis, Tom. Cryptography for developers. Elsevier, 2006.

[10] 摘录自 Wuille, Pieter. BIP-66: Strict DER signatures. 2015. https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki

是固定的 64 个字节. 也因此, 在 Bitcoin 和 Bitcoin Cash 计划中的采用 Schnorr 签名的升级, 没有采用 DER 形式编码 Schnorr 签名值, 而是直接采用签名值的固定 64 个字节的二进制表示形式[11].

在 Section 3.4 中, 提到了 Bitcoin 和 Bitcoin Cash 采用的 Schnor 签名机制的签名值为 $(R, s), R \in \mathbb{G}, s \in \mathbb{F}_n^*$. 注意到签名值中 R 是一个点, 而 s 是 32 字节的整数. 如何利用 32 个字节表达椭圆曲线上的一个点 R? 由于公钥 P 也是一个椭圆曲线上的一个点, 不压缩形式的需要占用 65 个字节, 而目前的公钥的压缩表示需要占用 33 个字节, 32 个字节表示点的横坐标, 另外一个字节指示点的纵坐标的取值 (如果 (x, y) 满足 $y^2 = x^3 + 7$, 则 $(x, -y)$ 也满足). 可是如何用 32 个字节表达椭圆曲线上的一个点? 可以看到, 33 个字节的压缩形式的公钥表示是为了表达 1 比特的信息: y 还是 $-y$. 如果通过椭圆曲线的某种性质, 对点的纵坐标取值进行限定, 使得两种可能的纵坐标中仅有一个能够满足条件, 则可以用 32 个字节表示点. 可以借鉴 Section 3.5 中通过限制取值范围来处理 ECDSA 签名值中可锻造性问题的方案, 限制纵坐标的取值必须 $\leq p/2$ 或者限制纵坐标的取值必须为奇数 (或者必须为偶数). 限制奇数或者偶数的原因在于 p 为奇数, 则 $y \mod p$ 和 $-y \mod p$ 必然一个为奇数一个为偶数. 另外也可以通过约定 y 必须为是 $\mathbb{F}_p$ 上的二次剩余 (Quadratic Residue) 的方式添加限制. Pieter 的关于 Schnorr 签名的 BIP 草稿 "Schnorr Signatures for secp256k1" 中采用的是通过二次剩余的方式添加对 y 值的约束. 选取的原因涉及椭圆曲线的实现细节 (如 Jacobi 符号) 以及效率因素, 将在另外的文档再具体介绍.

3.7 不需要提供签名值对应的消息 m 时可伪造签名值

ECDSA 签名过程中, 首先利用哈希函数对待签名消息 m 计算散列值, 然后针对散列值进行后续的计算. 验签操作也同样先对消息 m 进行哈希, 并利用哈希值执行后续的验证操作. 然而一个较少提及的应用注意事项是, 验证签名时一定要提供消息 m 本身, 而不能仅提供消息 m 的哈希值. 假设 $\sigma = (r, s)$ 是使用私钥 d 对消息 m 计算的签名值, 如果验签时不要求提供消息 m 本身 (仅提供消息的哈希值 e), 则任何人都可以伪造关于私钥 d 的签名值 $\sigma' = (r', s')$.

根据 ECDSA 签名验证过程, 验证一个利用私钥 d 对消息 m 签署的签名值 (r, s) 时, 需要验证的条件为:

$$s^{-1}(eG + rP) = (x', y') = R', \ x' \mod n \equiv r, \ \text{其中} \ r \equiv x \mod n, (x, y) = R = kG.$$

任何人都可以根据一个已知的合法签名值 $\sigma = (r, s)$ 利用上式直接构建满足上式的

[11] *"Signature encoding: Instead of DER-encoding for signatures (which are variable size, and up to 72 bytes), we can use a simple fixed 64-byte format."* 摘录自 Wuille, Pieter. BIP-draft: Schnorr Signatures for secp256k1. 2018. https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki

r', s', 而无需知道私钥 d. 构建时首先选择随机数 $u, v \in_R \mathbb{F}_n^*$ 并计算 $R' = (x', y') = uG + vP$, 令 $r' \equiv x' \mod n$. 为了使最终的签名 $\sigma' = (r', s')$ 在验签时, 能够验证通过, 需要 $s'^{-1}(e'G + r'P) = uG + vP$, 则有

$$\begin{cases} s'^{-1}e' & \equiv u \mod n \\ s'^{-1}r' & \equiv v \mod n, \end{cases} \rightarrow \begin{cases} s' & \equiv r'v^{-1} \mod n \\ e' & \equiv r'uv^{-1} \mod n \end{cases}$$

则 $\sigma = (r', s')$ 和哈希值 h' 利用公钥 $Q = dG$ 也能够验证通过:

$$s'^{-1}(e'G + r'P) = s'^{-1}e'G + s'^{-1}r'P = (r'^{-1}vr'uv^{-1})G + r'^{-1}vr'P = uG + vP.$$

如果在验证 ECDSA 签名时, 要求提供哈希值所对应的消息 m, 则无法执行上述伪造签名的过程. 这是由于哈希函数的抗碰撞特性保证了从 e' 推导 m' 并且满足 $e' = H(m')$ 是不可能的. 尝试先选定消息 m' s.t. $e' = H(m')$ 来伪造签名也是不可能行, 因为为了成功构造签名 (r', s'), 则需要

$$\begin{cases} s'^{-1}e' & \equiv u \mod n \\ s'^{-1}r' & \equiv v \mod n, \end{cases} \rightarrow e'u^{-1} \equiv s' \equiv r'v^{-1} \mod n \rightarrow r' \equiv e'u^{-1}v \mod n$$

选择 u, v 的值, 使得下面的方程组成立是不现实的.

$$\begin{cases} r' = & e'u^{-1}v \mod n \\ r' = & x' \mod n, (x', y') = uG + vP \end{cases}$$

这是因为上述方程组等价于要求相同的 u, v 值按照不同的计算路径有相同的输出结果. 如果将两个不同的计算路径看做伪随机函数, 则上述方程组等价于要求 2^{256} 空间上的两次相互独立的随机取值相同 (碰撞). 根据生日悖论的结论, 构建这样的碰撞大概需要 2^{128} 次操作, 这是不现实的, 也即尝试通过先选定消息 m' s.t. $e' = H(m')$ 来伪造签名是不可能行的.

Craig Wright (澳本聪) 曾经利用本节介绍的原理通过伪造 Satoshi Nakamoto 的签名值, 进而宣称自己为中本聪. 其实任何人都可以做类似的宣称, 甚至有网站[12] 提供了自动化的工具来伪造中本聪的签名来帮助你宣称自己是中本聪.

4 从 ECDSA 签名值逆推公钥

尽量减少每一笔交易/每一个区块的大小, 对于缩减区块体积, 提升区块的传播速度等方面大有裨益. Bitcoin 网络中一个典型的 P2PKH 交易的主要内容在 Figure 2 中展示其中解

[12] Prove to the world you are Satoshi Nakamoto. https://albacore.io/faketoshi

锁脚本 (Unlocking Script) 中签名值 sig 按照 DER 编码长度大约为 70 个字节, 压缩的公钥 PubK 需要 33 个字节 (推荐使用压缩形式), 锁定脚本 (Locking Script) 中字节码 DUP, HASH160, EQUALVERIFY, CHECKSIG 各占用一个字节, 而 PubKHash 为 20 个字节, 则 Figure 2 展示的脚本占用的链上存储空间大约为 130 个字节.

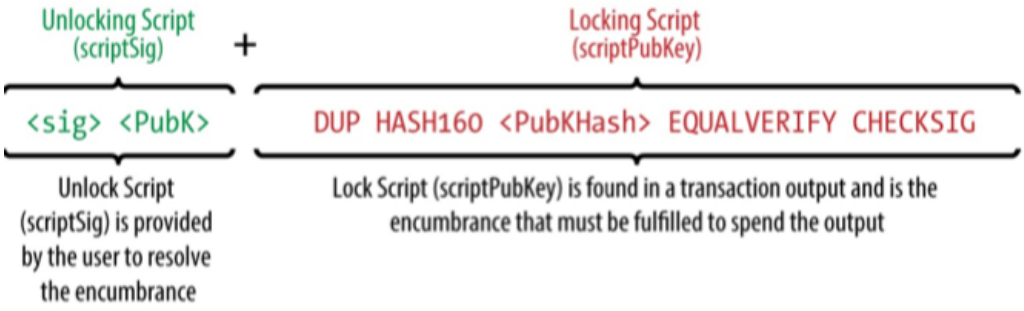

Figure 2: Bitcoin 中的 P2PKH 交易: 解锁脚本与锁定脚本

ECDSA 签名机制的一个在区块链场景中非常有用但是 Bitcoin 中没有利用的特性是可以根据签名值 sig 推算出公钥 PubK, 下文中用可恢复签名来指代这一特性. 这意味着解锁脚本中的 33 个字节的压缩公钥 PubK 字段是冗余的. 利用从 ECDSA 签名值可以恢复公钥的特性, 解锁脚本中不再需要字段 PubK, 则 Figure 2 所示的交易只需要占用 100 个字节左右的存储空间, 大约为 23% 的存储空间节省. 如果 Bitcoin 从最开始就利用了这一特性, 则同样大小的区块中可以存放更多交易, 历史区块占用的存储空间也可得到大幅缩减. Ethereum 项目中利用了 ECDSA 签名机制的这一特性, Ethereum 交易中没有包含公钥信息, 验签时从签名值推算即可.

实际上, 为了能够从签名值恢复出唯一的公钥值, 还需要存储额外的信息, 也因此为了支持可恢复签名的特性, libsecp256k1 库中定义了如 Listing 3 中所示的可恢复签名结构体, 根据注释可以看到该结构体的大小为 65 个字节.

Listing 3: libsecp256k1 中的 secp256k1_ecdsa_recoverable_signature

```
/** Opaque data structured that holds a parsed ECDSA signature,
 *  supporting pubkey recovery.
 *
 *  The exact representation of data inside is implementation defined and not
 *  guaranteed to be portable between different platforms or versions. It is
 *  however guaranteed to be 65 bytes in size, and can be safely copied/moved.
 *  If you need to convert to a format suitable for storage or transmission, use
 *  the secp256k1_ecdsa_signature_serialize_* and
 *  secp256k1_ecdsa_signature_parse_* functions.
```

```
 *
 *  Furthermore, it is guaranteed that identical signatures (including their
 *  recoverability) will have identical representation, so they can be
 *  memcmp'ed.
 */
typedef struct {
    unsigned char data[65];
} secp256k1_ecdsa_recoverable_signature;
```

对比 Listing 4中 libsecp256k1 中的普通 ECDSA 签名值 (根据注释大小保证为 64 个字节), 可以注意到 `secp256k1_ecdsa_recoverable_signature` 需要额外的一个字节来存储额外的信息. libsecp256k1 也提供了相应的签名接口 `secp256k1_ecdsa_sign_recoverable`.

Listing 4: libsecp256k1 中的 secp256k1_ecdsa_signature

```
/** Opaque data structured that holds a parsed ECDSA signature.
 *
 *  The exact representation of data inside is implementation defined and not
 *  guaranteed to be portable between different platforms or versions. It is
 *  however guaranteed to be 64 bytes in size, and can be safely copied/moved.
 *  If you need to convert to a format suitable for storage, transmission, or
 *  comparison, use the secp256k1_ecdsa_signature_serialize_* and
 *  secp256k1_ecdsa_signature_parse_* functions.
 */
typedef struct {
    unsigned char data[64];
} secp256k1_ecdsa_signature;
```

接下来介绍可恢复签名的原理, 以及可恢复签名结构体中额外的一个字节中存储的信息和必要性. 首先关注 ECDSA 签名值中的 $r \equiv x \mod n, (x, y) = kG, x, y \in \mathbb{F}_p$, 根据曲线 secp256k1 的参数可知, $n < p < 2n$, 则当 $x < n$ 时, $r = x$, 而当 $x >= n$, 有 $r = x + n$. 也即根据 r 以及 x 是否大于 n 这 1 比特的信息可以唯一确定 $R = kG$ 的横坐标 x. 进一步根据椭圆曲线的方程可以从 x 的值计算出纵坐标 y 的值, 前述已经论述过 y 和 $-y$ 都是对应 x 的合法值, 也即根据 x 的值以及 y 为奇数还是偶数这 1 比特信息可以唯一确定点 $R = kG$. 有了 R 信息之后, 对于合法的签名可以通过如下推算公钥 P:

$$R = s^{-1}(eG + rP) \rightarrow rP = sR - eG \rightarrow P = r^{-1}(sR - eG).$$

Listing 5: libsecp256k1 中的 secp256k1_ecdsa_sig_sign

```
1  static int secp256k1_ecdsa_sig_sign(const secp256k1_ecmult_gen_context *ctx,
       secp256k1_scalar *sigr, secp256k1_scalar *sigs, const secp256k1_scalar *
       seckey, const secp256k1_scalar *message, const secp256k1_scalar *nonce, int
       *recid) {
2      unsigned char b[32];
3      secp256k1_gej rp;
4      secp256k1_ge r;
5      secp256k1_scalar n;
6      int overflow = 0;
7
8      secp256k1_ecmult_gen(ctx, &rp, nonce);
9      secp256k1_ge_set_gej(&r, &rp);
10     secp256k1_fe_normalize(&r.x);
11     secp256k1_fe_normalize(&r.y);
12     secp256k1_fe_get_b32(b, &r.x);
13     secp256k1_scalar_set_b32(sigr, b, &overflow);
14     /* These two conditions should be checked before calling */
15     VERIFY_CHECK(!secp256k1_scalar_is_zero(sigr));
16     VERIFY_CHECK(overflow == 0);
17
18     if (recid) {
19         /* The overflow condition is cryptographically unreachable as hitting it
               requires finding the discrete log
20          * of some P where P.x >= order, and only 1 in about 2^127 points meet
               this criteria.
21          */
22         *recid = (overflow ? 2 : 0) | (secp256k1_fe_is_odd(&r.y) ? 1 : 0);
23     }
24     secp256k1_scalar_mul(&n, sigr, seckey);
25     secp256k1_scalar_add(&n, &n, message);
26     secp256k1_scalar_inverse(sigs, nonce);
27     secp256k1_scalar_mul(sigs, sigs, &n);
28     secp256k1_scalar_clear(&n);
29     secp256k1_gej_clear(&rp);
30     secp256k1_ge_clear(&r);
31     if (secp256k1_scalar_is_zero(sigs)) {
32         return 0;
33     }
34     if (secp256k1_scalar_is_high(sigs)) {
35         secp256k1_scalar_negate(sigs, sigs);
36         if (recid) {
37             *recid ^= 1;
```

```
38          }
39      }
40      return 1;
41  }
```

有了上述理论, 可以通过 libsecp256k1 中函数 `secp256k1_ecdsa_sig_sign` 的实现来理解额外的一个字节中存储的信息, 参见 Listing 5. 函数参数 `recid` 对应可恢复签名结构体 `secp256k1_ecdsa_recoverable_signature` 中字节数组的最后一个字节. 根据第 22 行可知, 如果 `recid` 第 2 最低比特为 1, 表示 $r > n$, 如果最低比特比特为 1, 则 R 的纵坐标是奇数. 值得注意的是, 前述的理论论述部分, 我们其实依赖了曲线 secp256k1 的余因子 (Cofactor) 为 1 这个事情, 当余因子不为 1 的时候, 从 r 计算 x 时, 需要考虑更多可能的 x 值, 在介绍 Ed25519 签名算法时, 再详细介绍余因子的概念以及在椭圆曲线计算中的影响.