深入理解 Ed25519: 原理与速度

longcpp

longcpp9@gmail.com

September 30, 2019

1 Ed25519 概述

Edwards-curve Digital Signature Algorithm (EdDSA) 是定义在 (扭曲) 爱德华曲线上 Schnorr 签名的变种签名机制. Ed25519 是 Bernstein 等人 2011 年在扭曲爱德华椭圆曲线 Edwards25519 (与蒙哥马利曲线 Curve25519 双向有理等价) 上构建的签名机制[1], 显著特点是高效安全, 在保证 128 比特的安全强度的前提下在 2.4GHz 的 Intel Westmere (Xeon E5620) CPU 上可以达到 10 万/秒的签名速度和 7 万/秒的验签速度. RFC 8032[2] 中给出了 EdDSA 签名的具体规范, 并且给出了基于两条具体曲线 Edwards25519 和 Edwards448 的签名机制 Ed25519 和 Ed448 测试向量. 其中 Edwards448 是 Mike Humberg 构建的椭圆曲线, 旨在提供 224 比特的安全强度. 值得注意的是, 2015 年 Bernstein 对 EdDSA 签名机制进行了推广[3] 以使 EdDSA 签名机制可以适用于更多的椭圆曲线. 本文中, 我们重点关注基于 Edwards25519 曲线的 EdDSA 的变种形式 PureEdDSA 和 HashEdDSA 的签名机制:Ed25519 和 Ed25519ph. RFC 8032 中为了统一两个变种的定义, 引入了预哈希函数 (Prehash) PH 的参数. HashEdDSA 是经典的先计算哈希值然后对哈希值计算签名的模式, 也即对于任意长度的消息 m, PH 都会输出固定长度的哈希值, 例如 PH 可以定义为 SHA-512: $\mathsf{PH}(m) = \text{SHA-512}(m)$. PureEdDSA 则直接对消息本身进行签名, 此时 PH 为恒等函数 (Identity Function), 也即 $\mathsf{PH}(m) = m$.

[1] Bernstein, Daniel J., Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures." In International Workshop on Cryptographic Hardware and Embedded Systems, pp. 124-142. Springer, Berlin, Heidelberg, 2011. https://link.springer.com/content/pdf/10.1007/978-3-642-23951-9_9.pdf

[2] RFC 8032. Edwards-Curve Digital Signature Algorithm (EdDSA). https://tools.ietf.org/html/rfc8032

[3] Bernstein, Daniel J., Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "EdDSA for more curves." Cryptology ePrint Archive 2015 (2015). https://eprint.iacr.org/2015/677.pdf

EdDSA 签名机制具有诸多良好的特性: 1) 在各种平台上都可以高速实现; 2) 签名过程不需要外部随机数; 3) 能够有效抵抗侧信道攻击; 4) 公钥和签名值都较小, 对于 Ed25519 而言公钥为 32 个字节签名值为 64 个字节; 5) 曲线上的点群运算是完备 (Complete) 的, 也即对于所有的点群中元素都成立, 计算时无需做额外的判断, 意味着运算时不需要对不受信的外部值做昂贵的点的验证; 6) EdDSA 签名机制本身安全性不受哈希碰撞的影响, 而 ECDSA 在出现哈希碰撞时会出现安全问题.

2 EdDSA 签名机制

根据 Bernstein 等人在 2015 年对 EdDSA 签名机制的推广和 RFC 8032 中的规范, EdDSA 签名机制有 11 个参数:

1. 奇素数 p: EdDSA 所依赖的椭圆曲线构建在有限域 $\mathbb{F}_p$ 上.
2. 整数 b 满足 $2^{b-1} > p$: EdDSA 公钥为 b 比特, 签名值为 $2b$ 比特, b 应为 8 的整数倍.
3. 有限域 $\mathbb{F}_p$ 中元素的 $b-1$ 比特的编码.
4. 可以产生 $2b$ 比特输出的具有密码学安全强度的哈希函数 H.
5. $\mathbb{F}_p$ 中的二次非剩余 d, d 是椭圆曲线方程的参数, 推荐选择尽可能接近零的值.
6. $\mathbb{F}_p$ 中非零元素 a, a 是曲线方程参数, 推荐 $p \mod 4 = 1$ 取 $a = -1$, 否则取 $a = 1$.
7. 基点 $B \neq (0,1)$ 并且 $B \in E = \{(x,y) \in \mathbb{F}_p \times \mathbb{F}_p \; s.t. \; ax^2 + y^2 = 1 + dx^2y^2\}$.
8. 整数 $c = 2$ 或 $c = 3$, 2^c 是椭圆曲线的余因子 (cofactor), EdDSA 私钥为 2^c 的倍数.
9. 整数 n 满足 $c \leq n < b$, 私钥为 $n+1$ 比特, 最高位为 1 (2^n 位), 最低 c 位置零.
10. 奇素数 ℓ 满足 $\ell B = (0,1)$ 并且 $2^c \times \ell = \#E$, 即 ℓ 为椭圆曲线点群的阶 (Order).
11. 预哈希函数 PH, PureEdDSA 和 HashEdDSA 对 PH 的定义不同.

点群中的单位元为 $(0,1)$, 并且点群上的加法运算是完备的 (Complete), 也即对于任意的点 $(x_1,y_1),(x_2,y_2)$ 都有

$$(x_1,y_1) + (x_2,y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}\right)$$

整数 $s : 0 < s < \ell - 1$ 用小端法编码为 b 比特的字符串的过程记为 $\mathsf{Encode}(s)$. 而 $(x,y) \in E$ 被编码为 b 比特的字符串, 记为 $\mathsf{Encode}(x,y)$, b 比特的编码包含 y 的 $(b-1)$

比特的编码和 1 比特的符号位: 如果 x 是负数, 则符号位为 1, 否则为 0. 根据这种编码方式可以立即确定 y 的值, x 的值则需要通过方程 $x = \pm\sqrt{(y^2-1)/(dy^2-a)}$ 和符号位进行确定 (由于 d 是 $\mathbb{F}_p$ 中的二次非剩余, 所以分母 dy^2-a 不为零). $x, y \in \mathbb{F}_p$, EdDSA 签名体制中对有限域 $\mathbb{F}_p$ 中的负数的定义为: 如果 x 的 $(b-1)$ 比特的编码字符串比 $-x$ 的 $(b-1)$ 比特的编码字符串字典序更大 (Lexicographically Larger), 则 $x \in \mathbb{F}_p$ 是负数. 对于 p 是大的奇素数并且采用小端法编码的情形, $\mathbb{F}_p$ 中负数是所有的奇数 $\{1, 3, 5, \ldots, q-2\}$.

EdDSA 签名机制的私钥是 b 比特的值 k, 记 $\mathsf{H}(k) = (h_0, h_1, \ldots, h_{2b-1})$. $h_0, h_1, \ldots, h_{b-1}$ 确定了一个整数值 s:

$$s = 2^n + \sum_{c \le i < n} 2^i h_i$$

整数值 s 决定公钥 $\mathsf{Encode}(A)$, 其中 $A = sB$. $h_b, h_{b+1}, \ldots, h_{2b-1}$ 用在签名值的计算过程中. 前面有提到, RFC 8032 中根据预哈希函数 PH 的定义了给出了 EdDSA 的两个变种: PureEdDSA 和 HashEdDSA, 由于两者的差异仅在于用 PH 处理待签名消息 m 的结果不同, 因此 EdDSA 可描述为: 用 PureEdDSA 对 $\mathsf{PH}(m)$ 进行签名.

PureEdDSA 对消息 m 计算签名值的结果 $2b$ 比特的值 $\mathsf{Encode}(R)||\mathsf{Encode}(S)$:

$$R = rB, S = (r + \mathsf{H}\left(\mathsf{Encode}(R)||\mathsf{Encode}(A)||\mathsf{PH}(m)\right) \cdot s) \mod \ell$$

其中 $r = \mathsf{H}(h_b||\ldots||h_{2b-1}||m)$ 并且被解释为小端法表示的 $2b$ 比特的整数值. 用公钥值 $\mathsf{Encode}(A)$ 验证关于消息 m 的签名值 $\mathsf{Encode}(R)||\mathsf{Encode}(S)$ 时, 首先需要从中解析出 A, R, S 的值, 并判定 A 和 R 是 E 中的元素并且 S 是集合 $\{0, 1, \ldots, \ell-1\}$ 中的值, 然后判断如下等式是否成立

$$(2^c \cdot S)B = 2^c R + (2^c \cdot h)A, \text{ 其中 } h = \mathsf{H}(\mathsf{Encode}(R)||\mathsf{Encode}(A)||\mathsf{PH}(m))$$

如果解析失败或者上述等式不成立, 则判定为非法的签名值, 否则判定为合法的签名值. 对消息 m 的 EdDSA 签名值的验证过程也即用 PureEdDSA 对 $\mathsf{PH}(m)$ 的签名值的验证过程.

Ed25519/Ed25519ph 是基于扭曲爱德华曲线 Edwards25519 实例化的 EdDSA 签名机制, 对应前述的 EdDSA 签名机制的 11 个参数分别为:

1. 有限域 $\mathbb{F}_p$ 的奇素数 $p = 2^{255} - 19$.
2. 整数 $b = 256$, 也即 Ed25519 的公钥为 256 比特, 签名值为 512 比特, 注意到 $2^{255} > p$.
3. $\mathbb{F}_p$ 中元素 $\{0, 1, \ldots, p-1\}$ 的 255 比特编码是用小端法表示的整数值, 最高位为零.

4. 基于 SHA-512[4] 的生成 512 比特输出的哈希函数 SHA-512(dom2(f, c)$||x$). f 是 $flag$ 缩写, 而 c 是 $context$ 的缩写. 对于 Ed25519, dom2(f, c) 是空字符串, 也因此 f 和 c 的值无关紧要, 这样可以保证 RFC 8032 中定义的 Ed25519 与已有的 Ed25519 实现之间保持兼容. Ed25519ctx 和 Ed25519ph 都可以带有额外的上下文参数 c, 至多为 255 个字节, 对于 Ed25519ctx 而言, $f = 0$, 对 Ed25519ph 而言, 则有 $f = 1$.

5. $\mathbb{F}_p$ 中的二次非剩余 $d = -121665/121666$.

6. 由于 $2^{255} - 19 \mod 4 = 1$, 因此选取 $\mathbb{F}_p$ 中的非零元素 $a = -1$.

7. 基点 B 的横坐标 0x216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a, 纵坐标为 0x6666666666666666666666666666666666666666666666666666666666666658.

8. Edwards25519 的余因子为 8, 也因此整数参数 $c = 3$.

9. $n = 254$.

10. Edwards25519 的阶 $\ell = 2^{252} + 27742317777372353535851937790883648493$.

11. Ed25519 的预哈希函数定义为 PH 为恒等函数, Ed25519ph 的 PH 定义为 SHA-512.

由于 Edwards25519 的点坐标的值位于 $\mathbb{F}_p$ 中, 根据 $\mathbb{F}_p$ 中元素的 255 比特, 可以用 32 字节的值编码点 (x, y): 用 255 比特的值编码 y 的值, 由于采用小端法编码, 则 32 字节中的最后一个字节的最高位为零) 然后将 x 的最低比特赋值给最后一个字节的最高位即可. 这是因为最后一个字节的最高比特位是符号位, 符号位为 1 表示 x 为负值, 根据前述的关于 $\mathbb{F}_p$ 中负数的定义可知, $\mathbb{F}_p$ 中的奇数均为负数, 由于奇数的最低位为 1, 所以将 x 的最低位赋值给符号位正好符合关于编码和负值的约定.

从 32 字节的编码中解码出点的值的步骤较编码复杂一些. 根据 255 比特的小端法编码可以直接得到 y 的值, 同时检查 y 的值在合理的范围之内并拒绝非法的值, 而根据符号位可知 x 的最低位 x_0. 根据 y 和曲线方程得到的 $x^2 = (y^2 - 1)/(dy^2 + 1) \mod p$ 可以恢复 x 的值, 计算中会涉及到域上的求逆运算和域上的开平方运算. 求逆可以利用费马小定理 $x^{-1} = x^{p-2} \mod p$ 或者扩展的欧几里得算法完成.

记 $u = y^2 - 1, v = dy^2 + 1$, 考虑根据 $x^2 = u/v \mod p$ 计算 x 的值. 根据欧拉准则 (Euler's Criterion), 如果 u/v 是 $\mathbb{F}_p$ 中的二次剩余, 则有 $(u/v)^{(p-1)/2} \equiv 1 \mod p$; 如果 $(u/v)^{(p-1)/2} \equiv -1 \mod p$, 则 $x^2 = u/v \mod p$ 无解. 考虑 u/v 是 $\mathbb{F}_p$ 中二次剩余的情况, 注意到 $2^{255} - 19 \equiv 5 \mod 8$, 也即存在整数 k 使得 $p = 8k + 5$. 由于 $(u/v)^{(p-1)/2} \equiv 1$

[4]RFC 6234: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). https://tools.ietf.org/pdf/rfc6234.pdf

$\mod p$, 则有 $(u/v)^{(p-1)/4} \equiv \pm 1 \mod p$. 如果 $(u/v)^{(p-1)/4} \equiv 1 \mod p$, 则 $x = (u/v)^{k+1} \mod p$ 是一个解:

$$x^2 \equiv (u/v)^{2(k+1)} \equiv (u/v)^{(p+3)/4} \equiv (u/v)^{(p-1)/4} \cdot (u/v) \equiv (u/v) \mod p$$

如果 $(u/v)^{(p-1)/4} \equiv -1 \mod p$, 则 $x = 2^{2k+1}(u/v)^{k+1} \mod p$ 是一个解:

$$x^2 \equiv 2^{4k+2}(u/v)^{2k+2} \equiv 2^{\frac{p-1}{2}}(u/v)^{\frac{p+3}{4}} \equiv 2^{\frac{p-1}{2}}(u/v)^{\frac{p-1}{4}} \cdot (u/v) \equiv -1 \cdot -1 \cdot (u/v) \mod p$$

上式成立的原因在于 2 是 $\mathbb{F}_p$ 中的二次非剩余以及欧拉准则 $2^{(p-1)/2} \equiv -1 \mod p$.

根据上述过程, 继续描述从 32 字节的编码中解码出点坐标值的过程, 从 y 的值可以计算出 u, v 的值. 当 u/v 是 $\mathbb{F}_p$ 中的二次剩余时, 计算 $x \equiv (u/v)^{(p+3)/8} \mod p$, 如果 $x^2 \equiv (u/v) \mod p$, 则 x 是平方根, 如果 $x^2 \equiv -(u/v) \mod p$, 则 $2^{(p-1)/4} \cdot x$ (也即 $\sqrt{-1} \cdot (u/v)^{(p+3)/8}$) 是平方根, 否则 u/v 不是 $\mathbb{F}_p$ 中的二次剩余, 意味着解码失败. 下一步则根据 x_0 的值选取正确的平方根, 如果 $x = 0$ 而 $x_0 = 1$, 则解码失败. 如果 $x_0 \neq x \mod 2$, 则 $x = p - x$, 解码得到的点即为 (x, y). 注意在具体实现解码操作时, 可以将求逆和求平方根的操作进行融合以简化运算, $(u/v)^{(p+3)/8} \mod p$ 可以等价变换为:

$$(u/v)^{(p+3)/8} = u^{\frac{p+3}{8}} v^{-\frac{p+3}{8}} = u^{\frac{p+3}{8}} v^{(p-1)-\frac{p+3}{8}} = u^{\frac{p+3}{8}} v^{\frac{7p-11}{8}} = uv^3(uv^7)^{(p-5)/8}$$

而判断 $x^2 \equiv (u/v) \mod p$ 和 $x^2 \equiv -(u/v) \mod p$ 可以等价变换为 $v \cdot x^2 \equiv u \mod p$ 和 $v \cdot x^2 \equiv -u \mod p$.

与 secp256k1 或者 secp256r1 等曲线的实现时常采用的 Jacobi 坐标系 $(X, Y, Z), x = X/Z, y = Y/Z$ 等不同, Hisil 等人指出[5]对于扭曲爱德华曲线 Edwards25519, 采用扩展的扭曲爱德华坐标系 (Extended Twisted Edwards Coordinates) 表示 $(x, y) \to (X : Y : Z : T), x = X/Z, y = Y/Z, T = XY/Z, Z \neq 0$ 更有利于点运算的高效实现. 在扩展的扭曲爱德华坐标系下, 单位元为 $(0 : 1 : 1 : 0)$, $(X : Y : Z : T)$ 的逆元为 $(-X : Y : Z : -T)$. 判断两个点 $(X_1, Y_1, Z_1, T_1), (X_2, Y_2, Z_2, T_2)$ 相等等价于判断 $X_1/Z_1 = X_2/Z_2$ & $Y_1/Z_1 = Y_2/Z_2$, 由于素数域上的求逆运算需要较多的计算量, 为了避免求逆运算并且注意到 $Z_1 \neq 0, Z_2 \neq 0$, 前述判断可以等价变换为 $X_1 \cdot Z_2 = X_2 \cdot Z_1$ & $Y_1 \cdot Z_2 = Y_2 \cdot Z_1$. 而两个点的加法运算 $(X_1, Y_1, Z_1, T_1) + (X_2, Y_2, Z_2, T_2) = (X_3, Y_3, Z_3, T_3)$, 可以按照如下步骤进行, 值得

[5]Hisil, Huseyin, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. "Twisted Edwards curves revisited." In International Conference on the Theory and Application of Cryptology and Information Security, pp. 326-343. Springer, Berlin, Heidelberg, 2008. https://eprint.iacr.org/2008/522.pdf

注意的是如下计算步骤对于两个点相同的情况同样适用.

$$A \leftarrow (Y_1 - X_1) \cdot (Y_2 - X_2), B \leftarrow (Y_1 + X_1) \cdot (Y_2 + X_2)$$
$$C \leftarrow 2 \cdot T_1 \cdot T_2, D \leftarrow 2 \cdot Z_1 \cdot Z_2$$
$$E \leftarrow B - A, F \leftarrow D - C, G \leftarrow D + C, H \leftarrow B + A$$
$$X_3 \leftarrow E \cdot F, Y_3 \leftarrow G \cdot H, T_3 \leftarrow E \cdot H, Z_3 \leftarrow F \cdot G$$

Listing 1 中展示了基于扩展的扭曲爱德华坐标系实现的点的加法, 加倍以及倍乘运算, 参见函数 `point_add`, `point_double` 以及 `point_mul`. 同时展示了前述的点的编解码过程 (参见函数 `point_compress` 和 `point_decompress`) 以及判断两个点是否相同的函数 `point_equal`. `point_decompress` 过程中所需的恢复横坐标的运算, 在函数 `recover_x` 中实现. 函数 `point_mul` 内部实现点的倍乘运算时, 采用了经典的 double-and-add 方法.

Listing 1: `Edwards25519` 点运算与编解码

```
import hashlib

def sha512(m):
    return hashlib.sha512(m).digest()

p = 2**255 - 19 # characteristic of base field

def fp_inv(x):
    return pow(x, p-2, p)

d = -121665 * fp_inv(121666) % p # Edwards25519 parameter

l = 2**252 + 27742317777372353535851937790883648493
def sha512_fp(m):
    return int.from_bytes(sha512(m), "little") % l

# Points are represented as tuples (X, Y, Z, T) of extended coordinates
# with x = X/Z, y = Y / Z, x * y = T/Z

def point_add(P, Q):
    A = (P[1]-P[0]) * (Q[1]-Q[0]) % p
    B = (P[1]+P[0]) * (Q[1]+Q[0]) % p
    C = 2 * P[3] * Q[3] * d % p
    D = 2 * P[2] * Q[2] % p
    E, F, G, H = (B-A)%p, (D-C)%p, (D+C)%p, (B+A)%p
    return (E*F % p, G*H % p, F*G % p, E*H % p)

```

```
28 def point_double(P):
29     return point_add(P, P)
30 
31 def point_mul(s, P):
32     Q = (0, 1, 1, 0) # Netural element
33     while s > 0:
34         if s & 1:
35             Q = point_add(Q, P)
36         P = point_double(P)
37         s >>= 1
38     return Q
39 
40 def point_equal(P, Q):
41     # X1 / Z1 == X2 / Z2 --> X1 * Z2 = X2 * Z1
42     # Y1 / Z1 == Y2 / Z2 --> Y1 * Z2 = Y2 * Z1
43     if (P[0] * Q[2] - Q[0] * P[2]) % p != 0:
44         return False
45     if (P[1] * Q[2] - Q[1] * P[2]) % p != 0:
46         return False
47     return True
48 
49 # square root of -1
50 fp_sqrt_m1 = 0x2b8324804fc1df0b2b4d00993dfbd7a72f431806ad2fe478c4ee1b274a0ea0b0
51 
52 # recover x-coordinate
53 def recover_x(y, sign):
54     if y >= p:
55         return None
56     x2 = (y*y - 1) * fp_inv(d*y*y + 1)
57     if x2 == 0:
58         if sign:
59             return None
60         else:
61             return 0
62     # Compute square root of x2
63     x = pow(x2, (p+3) // 8, p)
64     if (x*x - x2) % p != 0:
65         x = x * fp_sqrt_m1 % p
66     if (x*x - x2) % p != 0:
67         return None
68 
69     if (x & 1) != sign:
```

```
        x = p - x
    return x

# Base point
g_x = 0x216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a
g_y = 0x6666666666666666666666666666666666666666666666666666666666666658
G = (g_x, g_y, 1, g_x * g_y % p)

def point_compress(point):
    zinv = fp_inv(point[2])
    x = point[0] * zinv % p
    y = point[1] * zinv % p
    return int.to_bytes(y | ((x & 1) << 255), 32, "little")

def point_decompress(bytes):
    if len(bytes) != 32:
        raise Exception("Invalid input length for decompression")
    y = int.from_bytes(bytes, "little")
    sign = y >> 255 # get the sign bit
    y &= (1 << 255) - 1 # clear the sign bit

    x = recover_x(y, sign)
    if x is None:
        return None
    else:
        return (x, y, 1, x * y % p)
```

与 ECDSA 签名机制中直接用私钥和基点进行点倍乘运算得到公钥值不同, EdDSA 签名机制中私钥和公钥之间的关系更为复杂, 按照以下步骤进行 (参见 Figure 1): 1) 用 SHA-512 计算 32 字节的私钥的 512 比特 (64 个字节) 的哈希值, 低 32 字节用于生成公钥; 2) 将 32 字节中第一个字节的最低 3 比特清零, 最后一个字节的最高位清零, 将最后一个字节的第二最高位置 1; 3) 将设置之后的 32 字节的看做是小端法表示的整数, 记为 s, 计算 $A = sB$; 4) A 的编码 $\mathsf{Encode}(A)$ 是最终的公钥.

签名过程先利用 SHA-512 对私钥进行哈希运算, 得到 64 字节的输出, 记为 $h = (h[0], h[1], \ldots, h[63])$, 其中 $h[i], i \in \{0, 1, \ldots, 63\}$ 表示第 i 个字节. 如前所述 $(h[0], \ldots, h[31])$ 用于生成公钥值, 记 $prefix = h[32]||h[33]||\ldots||h[63]$, $prefix$ 在后续的 EdDSA 签名过程中扮演了随机数的角色. 计算

$$\mathsf{SHA\text{-}512}(\mathsf{dom2}(F, C)||prefix||\mathsf{PH}(m)),$$

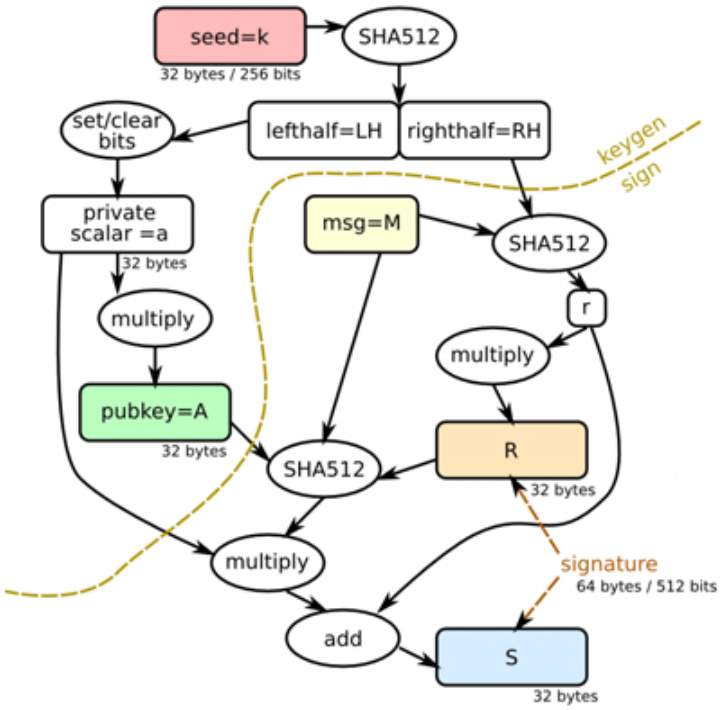

Figure 1: Ed25519 签名过程

其中根据 Ed25519, Ed25519ctx, Ed25519ph 的不同, dom2(F, C) 和 PH(m) 的定义有所不同, 得到的 64 字节的哈希值解释为小端法表示的整数 r 并计算

$$R = rB.$$

计算

$$\text{SHA-512}\left(\text{dom2}(F, C)||\text{Encode}(R)||\text{Encode}(A)||\text{PH}(m)\right).$$

将得到的 64 字节的哈希值解释为小端法表示的整数 k 并计算

$$S = (r + k * s) \mod \ell.$$

最终的签名值为 32 字节的 Encode(R) 以及 32 字节的 Encode(S), 总计为 64 字节.

签名验证时从签名值 Encode(R)||Encode(S) 解码得到点 R 和整数 S, 解码 Encode(A) 得到公钥点 A. 如果签名值或者公钥值解码失败或者 S 的值不在合理的范围内, 则判定为非法的签名值. 计算

$$\text{SHA-512}\left(\text{dom2}(F, C)||\text{Encode}(R)||\text{Encode}(A)||\text{PH}(m)\right).$$

将得到的 64 字节的哈希值解释为小端法表示的整数 k 并检查下面的等式是否成立:

$$(8S)B = 8R + (8k)A.$$

若等式成立, 则判定为合法的签名值; 否则判定为非法的签名值. Ed25519 签名验签的 Python 示例代码参见 Listing 2, 其中函数 `secret_to_pubkey` 实现从私钥生成公钥的过程, 函数 `ed25519_sign` 和 `ed25519_verify` 则分别实现了 Ed25519 的签名和验签.

Listing 2: Ed25519 签名机制

```
def secret_expand(secret):
    if len(secret) != 32:
        raise Exception("Bad size of private key")
    h = sha512(secret)
    a = int.from_bytes(h[:32], "little")
    a &= (1 << 254) - 8
    a |= (1 << 254)
    return (a, h[32:])

def secret_to_pubkey(secret):
    (a, dummy) = secret_expand(secret)
    A = point_mul(a, G)
    return point_compress(A)

def ed25519_sign(secret, msg):
    a, prefix = secret_expand(secret)
    A = point_compress(point_mul(a, G))
    r = sha512_fp(prefix + msg)
    R = point_compress(point_mul(r, G))
    h = sha512_fp(R + A + msg)
    s = (r + h * a) % p
    return R + int.to_bytes(s, 32, "little")

def ed25519_verify(pubkey, msg, signature):
    if len(pubkey) != 32:
        raise Exception("Bad public key length")
    if len(signature) != 64:
        raise Exception("Bad signature length")

    A = point_decompress(pubkey)
    if not A:
        return False

```

```
    Rbytes = signature[:32]
    R = point_decompress(Rbytes)
    if not R:
        return False

    s = int.from_bytes(signature[32:], "little")
    if s >= p:
        return False

    k = sha512_fp(Rbytes + pubkey + msg)
    sG = point_mul(8*s, G)
    kA = point_mul(8*k, A)
    return point_equal(sG, point_add(point_mul(8, R), kA))
```

根据 Listing 1 和 Listing 2 中的 Python 示例代码, 配合 RFC 8032 中给出的测试向量, 可以验证 PoC 代码的正确性, 参见 Listing 3.

Listing 3: 验证 Ed25519 实现正确性

```
>>> import ed25519
>>> import base64
>>>
>>> secret_str = (
... '9d61b19deffd5a60ba844af492ec2cc4'
... '4449c5697b326919703bac031cae7f60')
>>> pubkey_str = (
... 'd75a980182b10ab7d54bfed3c964073a'
... '0ee172f3daa62325af021a68f707511a')
>>> msg_str = ""
>>> sig_str = (
... 'e5564300c360ac729086e2cc806e828a'
... '84877f1eb8e5d974d873e06522490155'
... '5fb8821590a33bacc61e39701cf9b46b'
... 'd25bf5f0595bbe24655141438e7a100b')
>>>
>>> secret = base64.b16decode(secret_str, True)
>>> pubkey = base64.b16decode(pubkey_str, True)
>>>
>>> msg    = base64.b16decode(msg_str, True)
>>> sig    = base64.b16decode(sig_str, True)
>>>
>>> pubkey == ed25519.secret_to_pubkey(secret)
True
```

```
25 >>> base64.b16encode(ed25519.ed25519_sign(secret, msg)).lower()
26 b'e5564300c360ac729086e2cc806e828a84877f1eb8e5d974d873e0652249015
27 58d87fc207b0549fb6bc61bf6b550c7875e5f2a43320abd01ee6f2ef6965e5829'
28 >>>
29 >>> ed25519.ed25519_verify(pubkey, msg, sig)
30 True
```

预哈希函数 PH 定义的不同导致了两个变体 Ed25519 和 Ed25519ph 的存在. 为什么需要两种形式? RFC 8032 中解释到应该根据具体的应用需求来选用. 具体来说如果应用需要签名机制能够抗哈希碰撞, 也即签名机制即使在哈希碰撞的情形下仍然保持安全, 则应该选用 Ed25519; 如果应用要求生成签名的接口是 single-pass 的, 也即创建只需要遍历一次输入的消息, 则选用 Ed25519ph, 这是因为 Ed25519 在签名过程中需要对输入消息 m 执行两个哈希运算, 参见 Listing 2 中函数的 `ed25519_sign` (第 18 行和第 20 行).

dom2(F, C) 的值在 Ed25519 版本的签名计算和验签时要求为空, 否则为字节数组:

`"SigEd25519 no Ed25519 collisions" || octet(F) || octet(OLEN(C)) || C`,

其中 F 是单字节取值为 0~255, 而 C 表示上下文环境 *context* 并且最多可以为 255 字节. 也因此参数 C 的存在定义了带上下文环境实例化的扩展 Ed25519 签名机制. 在实际应用中, 上下文环境参数 C 可以用来为不同的协议之间或者在同一个协议内部对不同用途的 Ed25519 签名机制进行区分 (没有该参数的话类似的功能比较难以实现). 值得提及的是, 32 字节的`"SigEd25519 no Ed25519 collisions"` 不会被解码为曲线上的点. 这样的考虑是因为 Ed25519 签名机制的内部哈希函数总是以一个点的编码开始, 这样能够将 Ed25519 与 Ed25519ph, Ed25519ctx 区分开, 而 Ed25519ph, Ed25519ctx 可以通过 dom2(F, C) 中参数 F 的不同区分开. 因此不会出现一种 Ed25519 签名机制生成的签名值可以在另一种 Ed25519 签名机制下验证通过的情况, 也因此一个密钥对可以同时用于 Ed25519, Ed25519ph 以及 Ed25519ctx.

根据 RFC 8032 最后提供的 Ed25519, Ed25519ph 以及 Ed25519ctx 三种签名机制的示例代码, 可以认为 Ed25519ctx 是带上下文环境的 Ed25519, 也即这两个变种的预哈希函数都是恒等函数, 而 Ed25519ph 是采用 SHA-512 作为预哈希函数的变种, 也可以带上下文环境参数.

3 几种签名机制的关系

EdDSA 签名机制中的验签中判断的核心等式是 $(2^c \cdot S)B = 2^c R + (2^c \cdot h)A$, 其中每个点倍乘的标量都乘以 2^c 的原因是在 RFC 8032 中有相应的解释: *The given verification formulas for both Ed25519 and Ed448 multiply points by the cofactor. While this is not*

strictly necessary for security (in fact, any signature that meets the non-multiplied equation will satisfy the multiplied one), in some applications it is undesirable for implementations to disagree about the exact set of valid signatures. Such disagreements could open up, e.g., fingerprinting attacks. 如果不考虑标量中的 2^c, 则这个等式与 Schnorr 签名机制中验签判定的核心等式基本一致. 关于这点 Bernstein 等人在提出 EdDSA 签名机制的论文中就 ElGamal, (EC-)Schnorr, (EC-)DSA 以及 EdDSA 签名机制之间的联系有着精彩的论述.

ElGamal 签名机制是在椭圆曲线密码学诞生之前提出的, 工作在大素数乘法群 $\mathbb{Z}_p^*$ 上, 阶为 $\ell = p - 1$, 本原元记为 B, 私钥记为 a, 公钥记为 $A \equiv B^a \mod p$, 则对消息 m 的在随机数 $k \in \mathbb{Z}_\ell^*$ 下的签名值为 (R, S), 其中

$$R \equiv B^k \mod p,\ S \equiv k^{-1} \cdot (\mathsf{H}(m) - Ra) \mod \ell.$$

签名验证涉及到的核心的判断是

$$B^{\mathsf{H}(m)} \equiv A^R \cdot R^S \mod p.$$

对于正确的签名值有 $A^R \cdot R^S \equiv B^{aR} B^{kS} \equiv B^{aR+kS} \mod p$, 又由于

$$S \equiv k^{-1} \cdot (\mathsf{H}(m) - Ra) \mod \ell \implies aR + kS \equiv \mathsf{H}(m) \mod \ell,$$

因此对于正确的签名值有 $B^{\mathsf{H}(m)} \equiv A^R \cdot R^S \mod p$ 成立.

为了满足安全性, 现在通常推荐素数 p 需要为 2048 比特, 则对应的 ElGamal 签名值为 4096 比特, 签名值较大. 1989 年 Schnorr 提出了可以看做是 ElGamal 签名机制变种的 Schonrr 签名机制, 能够大大缩短签名值的长度. 而数字签名算法 DSA 是吸收了 Schnorr 签名设计思想的另一种 ElGamal 签名机制的变种签名机制. DSA 在 1994 年被采纳为 FIPS 标准: FIPS-186. 基于椭圆曲线的 DSA 方案即为熟知的 ECDSA.

根据拉格朗日定理可知, 大素数乘法群 $\mathbb{Z}_p^*$ 存在阶为 q, $q|(p-1)$ 的乘法子群. 对于 $p \approx 2^{1024}$ 的情况, 一般取 $q \approx 2^{160}$. Schnorr 签名机制对 ElGamal 签名机制进行了改动, 使得最终的签名值只有 $2\log_2(q)$ 比特而非 $2\log_2(p)$. Schnorr 签名工作在 $\mathbb{Z}_p^*$ 中的 q 阶子群, 方案的安全性基于这样的事实: 在特定的 $\mathbb{Z}_p^*$ 子群上求解离散对数问题是困难的.

ElGamal 的签名验证过程中 R 扮演了两种角色: $\mathbb{Z}_p^*$ 中的元素以及 A 的指数. 由于群的元素不一定可以直接作为指数 (标量), 因此更广义的群结构通常需要一个将群中元素映射成为标量的映射 $\mathbf{x}$. 例如 ECDSA 签名机制依赖的群结构是大素数域 $\mathbb{F}_p$ 上的椭圆曲线加法点群中阶为 ℓ 的子群, 而映射 $\mathbf{x}(R)$ 定义为点 R 的 x 坐标. 对比 ElGamal 签名机制, ECDSA 签名机制中还用 $-A$ 替换了 A, 则签名过程中的减法相应变换为加法, 对于基点 B, 私钥 a, 公钥 $A = aB$, 消息 m 在随机数 k 下的签名值为 $(\mathbf{x}(R), S)$, 其中:

$$R = kB,\ S = k^{-1}(\mathsf{H}(m) + \mathbf{x}(R)a) \mod \ell.$$

并且签名验证中判断的主要方程变为:

$$\mathsf{H}(m)B + \mathbf{x}(R)A = SR.$$

通过添加约束要求 S 模 ℓ 可逆 (ℓ 为素数时, 非零 S 模 ℓ 都可逆), 则前述方程变换为

$$S^{-1}\left(\mathsf{H}(m)B + \mathbf{x}(R)A\right) = R,$$

对比之下, 可以发现验证所需的计算从原先的 3 次点倍乘运算减少为 2 次点倍乘运算.

对于映射 $\mathbf{x}$, Schnorr 签名则采用了密码学安全的哈希函数, 借助于哈希函数的高效率, 用最少的计算代价消除了可能隐藏在简单 $\mathbf{x}$ 映射中的数学结构. 与前述的 ECDSA 签名类似, Schnorr 签名的签名值为 $(\mathbf{x}(R), S)$ 而非 (R, S). 给定签名值 $(\mathbf{x}(R), S)$, 验证者验证下面的等式是否成立:

$$\mathbf{x}(R') = \mathbf{x}(R), \text{ 其中 } R' = S^{-1}(\mathsf{H}(m)B + \mathbf{x}(R)A).$$

Schnorr 同时合并了对 R 和消息 m 的哈希计算 $\mathsf{H}(R||m)$, Bernstein 等人指出合并 R 和消息 m 的哈希计算可以理解成用 $\mathbf{x}(R)S$ 替换 S. 对于合法的 Schnorr 签名值, 有等式 $SR = \mathsf{H}(m)B + \mathbf{x}(R)A$ 成立. 用 $\mathbf{x}(R)S$ 替换 S 并假设 $\mathbf{x}(R) \neq 0$ 则有:

$$\mathbf{x}(R)SR = \mathsf{H}(m)B + \mathbf{x}(R)A \implies SR = \mathbf{x}(R)^{-1}\mathsf{H}(m)B + A.$$

注意到这里 $\mathbf{x}(R)^{-1}\mathsf{H}(m)$ 只是将两个哈希值做了乘法, 考虑 $\mathsf{H}(R||m)$ 时, $\mathbf{x}(R)^{-1}\mathsf{H}(m)$ 是多余的, 则上式, 也即签名验证的方程, 可简化为 $SR = \mathsf{H}(R||m)B + A$. 签名过程中 S 的计算方式也调整成 $S \equiv k^{-1}(\mathsf{H}(R||m) + a) \mod \ell$, 仍然需要在签名时计算素数域上元素的逆. Schnorr 签名中真正利用的验证等式实际为 $SB = R + \mathsf{H}(R||m)A$, 通过调整为该等式, 签名中 $S = r + \mathsf{H}(R||m)a$, 则签名和验签中都不需要计算素数域的逆运算, 对于减少实现的代码体积和运行时运算都有益处.

值得指出的是, 在哈希函数输入中包含 R, 使得签名机制在面对哈希碰撞的情况下也能保持安全性. Schnorr 签名的实际部署受到了相关专利的掣肘, 但是其构造方式被理论专家熟知, 因为在哈希函数输入中包含 R 的做法使得各种安全证明成为可能. EdDSA 签名机制的签名验证方程与 Schnorr 签名的签名验证方程一致 (不考虑余因子的影响). 另外在哈希函数的输入中还包括了公钥 A, 对于提升 EdDSA 签名机制在 multi-user 场景下的安全性有益处[6]. 而签名值的表示中, 由于对底层素数域的特征的巧妙选取, 用较少的空间即可完整编码点. 例如利用 32 字节即可编码 Edwards25519 上的点, 因此无需使用哈

[6]Bernstein, Daniel J. "Multi-user Schnorr security, revisited." IACR Cryptology ePrint Archive 2015 (2015): 996. https://ed25519.cr.yp.to/multischnorr-20151012.pdf

希函数对 R 进行压缩, 不使用哈希函数对 R 进行压缩也带来了可以批量验证签名的益处. 在 BTC/BCH 网络中升级计划中的 Schnorr 签名机制也采用了不压缩 R 的 Schnorr 签名形式, 以利用批量验证特性来加速区块验证[7].

4 EdDSA 批量验签与速度实测

区块链场景中, 一个区块中通常包含大量的签名并且网络上的每个节点都要对所有的签名值进行校验. 也因此, 签名验证的速度对于区块的传播速度有影响. 虽然也可以通过在内存池中对每笔交易的签名值验证进行进行缓存来加速区块验证速度, 但改进验签操作本身的效率对于效率的整体提升有益处. 如前所述, EdDSA 签名机制的设计允许 EdDSA 签名机制的实现利用批量验证来加速多个签名的验证过程.

假设我们有一组签名值 (m_i, A_i, R_i, S_i), 其中 (R_i, S_i) 是关于消息 m_i 的对应公钥 A_i 的签名值. 为了进行批量验证, 随机选择 128 比特的随机数 z_i, 其中每个 z_i 值相互独立, 然后计算 $H_i = \mathsf{H}(\mathsf{Encode}(R_i)||\mathsf{Encode}(A_i)||m_i)$, 并用多标量乘法验证如下方程是否成立:

$$\left(-\sum_i z_i S_i \mod \ell\right) B + \sum_i z_i R_i + \sum_i (z_i H_i \mod \ell) A_i = 0$$

关于多标量乘法可以用 Pippenger 的方法[8], 也可以用 Bos-Coster 方法[9]. Bernstein 等人推荐使用 Bos-Coster 方法.

如果上式验证通过, 则认为 $8S_iB = 8R_i + 8H_iA_i$ 对每个 i 都成立, 也即每个签名值都合法. 原因在于, $P_i = 8R_i + 8H_iA_i - 8S_iB$ 是阶为 ℓ 的循环子群中的一个元素, 并且经由上式验证满足 $\sum_i z_i P_i = 0$, 除非所有的 $P_i = 0$, 否则对于随机选择的相互独立的 z_i 的值 $\sum_i z_i P_i = 0$ 成立的概率不超过 2^{-128}. 例如, 假设 P_4 非零, 则 $z_1, z_2, z_3, , z_5, z_6, \ldots$ 的值选定之后, 仅有一个 z_4 的值能够使得 $\sum_i z_i P_i = 0$ 成立, 随机选择的 z_4 恰好等于该值的概率为 2^{-128}.

如果上式验证失败, 则认为 (m_i, A_i, R_i, S_i) 集合中, 至少有一次非法的签名值, 此时重新开始逐个验证签名值. 如果签名值集合中仅有一小部分签名值非法, 则存在一些技术可以批量鉴别这些签名值. 但是所有现存的批量鉴别技术都会随着非法签名值的增多而越来越慢. 另外考虑到可能存在的 DoS 攻击, 逐个验证签名值是较好的策略.

[7] Pieter Wuille. "Schnorr Signatures for secp256k1." https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki

[8] Pippenger, Nicholas. "On the evaluation of powers and related problems." In 17th Annual Symposium on Foundations of Computer Science (sfcs 1976), pp. 258-263. IEEE, 1976.

[9] De Rooij, Peter. "Efficient exponentiation using precomputation and vector addition chains." In Workshop on the Theory and Application of of Cryptographic Techniques, pp. 389-399. Springer, Berlin, Heidelberg, 1994. https://link.springer.com/content/pdf/10.1007/BFb0053453.pdf

libsodium[10] 中实现了 Ed25519 的签名机制, 利用在 Listing 4 中展示的代码测试 libsodium 1.0.18 版本中 Ed25519 签名和验签的速度. 在 Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 上执行得到的结果展示在 Listing 5 中, 也即 46500 sign/s 以及 15300 verify/s.

Listing 4: Ed25519 Speed in Libsodium

```
#include <iostream>
#include <chrono>
#include <cstdio>
#include "sodium.h"

template <typename Func, typename Clock = std::chrono::steady_clock,
        typename Unit = std::chrono::milliseconds>
inline double perf(Func f) {
    auto t1 = Clock::now();
    f();
    auto t2 = Clock::now();
    return std::chrono::duration_cast<Unit>(t2 - t1).count();
}

const int counts = 10000;
const int msglen = 32;

struct libsodium_ed25519_context {
    unsigned char secret[crypto_sign_SECRETKEYBYTES];
    unsigned char pubkey[crypto_sign_PUBLICKEYBYTES];
    unsigned char message[msglen];
    unsigned char signature[crypto_sign_BYTES];
};

void libsodium_bench_ed25519_sign_verify() {
    libsodium_ed25519_context ctx{};
    crypto_sign_keypair(ctx.pubkey, ctx.secret);

    auto sign = [&]() {
        for(int i = 0; i < counts; ++i) {
            crypto_sign_detached(ctx.signature, NULL,
                ctx.message, msglen, ctx.secret);
        }
    };
```

[10]libsodium: A modern, portable, easy to use crypto library. https://github.com/jedisct1/libsodium

```
35
36      auto milliseconds = perf(sign);
37      printf("avg time of per ed25519_sign: %.2f us, ",
38              milliseconds * 1000 / counts);
39      printf("about %.2f sign/s\n", counts / (milliseconds / 1000));
40
41      auto verify = [&]() {
42          for(int i = 0; i < counts; ++i) {
43              if(crypto_sign_verify_detached(ctx.signature, ctx.message,
44                  msglen, ctx.pubkey) != 0) {
45                  fprintf(stderr, "signature verification failed\n");
46                  exit(1);
47              }
48          }
49      };
50
51      milliseconds = perf(verify);
52      printf("avg time of per ed25519_verify: %.2f us, ",
53             milliseconds * 1000 / counts);
54      printf("about %.2f verify/s\n", counts / (milliseconds / 1000));
55  }
56
57  int main() {
58      libsodium_bench_ed25519_sign_verify();
59      return 0;
60  }
```

Listing 5: Ed25519 Speed in libsodium

```
1  long@LMBP $ ./libsodium-ed25519-bench.exe
2  avg time of per ed25519_sign: 21.50 us, about 46511.63 sign/s
3  avg time of per ed25519_verify: 65.40 us, about 15290.52 verify/s
```

OpenSSL 自 1.1.1 版本中也增加了对 Ed25519 签名机制的支持, 利用 OpenSSL 自带的速度测试工具在 Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 上执行得到的结果展示在 Listing 6 中, 也即 26900 sign/s 和 7200 verify/s. 相比 libosdium 1.0.18 版本中的 Ed25519 实现, OpenSSL 中的 Ed25519 在签名和验签速度方面都较慢.

Listing 6: Ed25519 Speed in OpenSSL

```
1  long@LMBP $ openssl speed eddsa
2  Doing 253 bits sign Ed25519's for 10s: 268115 253 bits Ed25519 signs in 9.97s
```

```
Doing 253 bits verify Ed25519's for 10s: 79327 253 bits Ed25519 verify in 9.98s
Doing 456 bits sign Ed448's for 10s: 71858 456 bits Ed448 signs in 9.97s
Doing 456 bits verify Ed448's for 10s: 19427 456 bits Ed448 verify in 9.98s
version: 3.0.0-dev
built on: built on: Sun Sep 29 11:12:41 2019 UTC
options:bn(64,64) rc4(16x,int) des(int) aes(partial) idea(int) blowfish(ptr)
compiler: cc -fPIC -arch x86_64 -O3 -Wall -DL_ENDIAN -DOPENSSL_PIC -D_REENTRANT
    -DNDEBUG
CPUINFO: OPENSSL_ia32cap=0x7ffaf3bfffebffff:0x29c6fbf
                              sign    verify    sign/s verify/s
 253 bits EdDSA (Ed25519)   0.0000s   0.0001s  26892.2   7948.6
 456 bits EdDSA (Ed448)   0.0001s   0.0005s   7207.4   1946.6
```

Tendermint Core 项目[11] 的共识投票算法选用了 Ed25519 签名机制, 项目中的 Ed25519 签名机制实现采用了 Go 语言标准库的实现, Listing 7 中给出了在同样的 CPU 上执行速度测试的结果. 62133 ns/sign 等价于 16000 sign/s, 而 158603 ns/verify 等价于 6300 verify/s.

Listing 7: Ed25519 Speed in Tendermint

```
 long@LMBP $ pwd
/Users/long/Go/src/github.com/tendermint/tendermint/crypto/ed25519
 long@LMBP $  go test -bench .
goos: darwin
goarch: amd64
pkg: github.com/tendermint/tendermint/crypto/ed25519
BenchmarkKeyGeneration-8          19882             61446 ns/op
BenchmarkSigning-8                18825             62133 ns/op
BenchmarkVerification-8            7472            158603 ns/op
PASS
ok      github.com/tendermint/tendermint/crypto/ed25519 4.903s
```

至此, 已经讨论了 Ed25519 签名机制在 libsodium, OpenSSL 以及 Go 语言密码学库中的实现, 由于 OpenSSL 和 Go 语言库中的实现显然是未经优化的, 仅考虑 libsodium 中的实现. libsodium 中 Ed25519 的签名和验签的实测速度大约为 46500 sign/s 和 15300 verify/s, 虽然签名速度相比 libsecp256k1 中的 ECDSA 实现确实有较大的改进, 但是区块链场景下更关切的验签的速度相比 libsecp256k1 中开启优化选项 `--enable-endomorphism` 和 `--with-bignum=gmp` 之后 ECDSA 在同样的 CPU 下可以达到的速度 19800 verify/s 相比并没有什么优势. Bernstein 等人在提出 Ed25519 的论文中, 提及

[11] Tendermint Core (BFT Consensus) in Go. https://github.com/tendermint/tendermint

Ed25519 在 Intel Westmere (Xeon E5620) CPU @ 2.4GH 下可以达到 71000 verify/s, 推测应该是 libsodium 中的 Ed25519 的实现也没有优化到极致. 幸运的是 Rust 语言实现的项目 ed25519-dalek[12] 对 Ed25519 的实现做了深度优化并且实现了批量验证的接口. Listing 8 中展示了利用项目自带的速度测试程序在同样的 CPU 上运行得到的结果 (为了方便展示, 输出做了精简). Listing 8 中还展示了开启 CPU 支持的 AVX2 指令集之后的速度测试结果, 可以注意到的验签的速度相比没有开启之前有了大幅提升. 值得注意的是, 做批量验证的时候展示的时间是完成一批签名验证所需的时间. 例如开启 AVX2 选项之后, 4 个签名的一次批量验证耗时的平均时间为 103.67us.

Listing 8: Ed25519 Speed in ed25519-dalek

```
long@LMBP $ pwd
/Users/long/Git/ed25519-dalek
long@LMBP $ cargo bench
    Finished release [optimized] target(s) in 0.06s

... signing                             time:   [23.215 us 23.275 us 23.336 us]
...  signing with  expanded secret key  time:   [22.693 us 22.773 us 22.854 us]
... signature verification              time:   [66.835 us 67.371 us 67.987 us]
... batch signature verification/4      time:   [172.20 us 173.04 us 173.90 us]
... batch signature verification/8      time:   [279.77 us 280.84 us 281.90 us]
... batch signature verification/16     time:   [527.72 us 529.46 us 531.20 us]
... batch signature verification/32     time:   [1.0178 ms 1.0298 ms 1.0457 ms]
... batch signature verification/64     time:   [1.9155 ms 1.9243 ms 1.9334 ms]
... batch signature verification/96     time:   [2.8346 ms 2.8556 ms 2.8811 ms]
... batch signature verification/128    time:   [3.5081 ms 3.5272 ms 3.5518 ms]
... batch signature verification/256    time:   [6.3983 ms 6.4198 ms 6.4399 ms]

long@LMBP $ export RUSTFLAGS=-Ctarget_cpu=native
long@LMBP $ cargo bench --features=avx2_backend
    Finished release [optimized] target(s) in 0.41s

...  signing                             time:   [21.195 us 21.309 us 21.417 us]
...  signing with  expanded secret key  time:   [20.705 us 20.808 us 20.918 us]
...  signature verification              time:   [36.696 us 36.793 us 36.920 us]
...  batch signature verification/4      time:   [103.40 us 103.67 us 103.98 us]
...  batch signature verification/8      time:   [175.04 us 175.71 us 176.54 us]
...  batch signature verification/16     time:   [338.26 us 342.17 us 346.44 us]
...  batch signature verification/32     time:   [616.49 us 617.83 us 619.39 us]
```

[12]Fast and efficient ed25519 signing and verification in Rust. https://github.com/dalek-cryptography/ed25519-dalek

```
29 ...  batch signature verification/64    time:   [1.2003 ms 1.2029 ms 1.2056 ms]
30 ...  batch signature verification/96    time:   [1.7099 ms 1.7137 ms 1.7180 ms]
31 ...  batch signature verification/128   time:   [2.1540 ms 2.1576 ms 2.1616 ms]
32 ...  batch signature verification/256   time:   [3.9470 ms 3.9520 ms 3.9571 ms]
```

为了方便比较, 将 Listing 8 中的 us/op 或者 ms/op 都换算成 op/s, 并且将批量验证时的测试结果也换算成做利用批量验证每秒可以验证多少个签名值, 转换后的结果参见 Table 1. 从中可以看到, 不用批量验证并且开启 AVX2 选项之后, 签名和验签速度分别可以达到 47000~48000 sign/s 和 27200 verify/s, 相比 libsodium 的速度, 在验签速度有了明显的提升, 也比开启优化选项之后的 libsecp256k1 中的 ECDSA 验签速度更快. 当利用批量验证时, 随着并行度的提升, 换算得到的验签速度也越来越快, 在一次批量验证 256 个签名时, 换算之后的验签速度大约为 64800 verify/s, 已经是优化选项全开之后 libsecp256k1 中 ECDSA 验签速度的 3 倍之多.

Table 1: Ed25519 Speed in ed25519-dalek

	us/op	op/s	us/op with avx2	op/s with avx2
sign	23.27	43000	21.31	46900
sign with extended secret key	22.77	43900	20.81	48100
verification	67.37	14800	36.79	27200
batch verification/4	173.04	23100	103.67	38600
batch verification/8	280.84	28500	175.71	45500
batch verification/16	529.46	30200	342.17	46800
batch verification/32	1029.80	31100	617.83	51800
batch verification/64	1924.30	33300	1202.90	53200
batch verification/96	2881.10	33300	1713.70	56000
batch verification/128	3551.80	36000	2157.60	59300
batch verification/256	6419.80	39900	3952.00	64800

可以发现 Table 1 中关于签名有两个数据, 一个是 sign, 一个是 sign with extended secret key. 两者的速度也有些许差异: 考虑开启 AVX2 选项的场景, 签名速度分别为 46900 sign/s, 48100 sign/s. 参考文件 `ed25519_benchmarks.rs` 和文件 `ed25519.rs` 中的代码, 参见 Listing 9, 参考前述的 Ed25519 签名机制计算过程, 可以看清两者之间的差异: Ed25519 机制中的私钥没有直接用于生成公钥, 而是取 SHA-512 对私钥的计算结果的一部分作为私钥值, 函数 `sign` 的速度测试中包含了 SHA-512 的哈希计算, 而 `fn sign_expanded_key` 中

不包含这部分计算, 参见函数 `pub fn sign(&self, message: &[u8])–> Signature`.

Listing 9: sign & sign with expanded secret key in ed25519-dalek

```
// file ed25519_benchmarks.rs
fn sign(c: &mut Criterion) {
    let mut csprng: ThreadRng = thread_rng();
    let keypair: Keypair = Keypair::generate(&mut csprng);
    let msg: &[u8] = b"";

    c.bench_function("Ed25519 signing", move |b| {
                      b.iter(| | keypair.sign(msg))
    });
}

fn sign_expanded_key(c: &mut Criterion) {
    let mut csprng: ThreadRng = thread_rng();
    let keypair: Keypair = Keypair::generate(&mut csprng);
    let expanded: ExpandedSecretKey = (&keypair.secret).into();
    let msg: &[u8] = b"";

    c.bench_function("Ed25519 signing with an expanded secret key", move |b| {
                      b.iter(| | expanded.sign(msg, &keypair.public))
    });
}
// file ed25519.rs
/// Sign a message with this keypair's secret key.
pub fn sign(&self, message: &[u8]) -> Signature {
    let expanded: ExpandedSecretKey = (&self.secret).into();

    expanded.sign(&message, &self.public)
}
```