基于 secp256k1 的自同态映射加速 ECDSA 验签

longcpp

longcpp9@gmail.com

August 12, 2019

由于 Bitcoin 的采纳, 基于椭圆曲线 secp256k1 的 ECDSA 签名机制在区块链领域得到广泛应用. Bitcoin 最开始采用的是 OpenSSL 中实现的基于 secp256k1 的 ECDSA 签名机制, 然而 OpenSSL 中没有针对曲线 secp256k1 进行针对性优化, 也就导致 OpenSSL 中基于 secp256k1 曲线的 ECDSA 效率很差. 值得注意的是, OpenSSL 项目中对基于 secp256r1 曲线的 ECDSA 进行了深度优化.

编译链接 OpenSSL 1.1 版本, 在 Intel(R) Core(TM) i7-6700HQ CPU 上的执行结果显示 OpenSSL 提供的基于 secp256k1 的 ECDSA 的执行速度大约为 2000 sign/s 和 2300 verify/s, 而基于 secp256r1 的 ECDSA 的执行速度大约为 33000 sign/s 和 12000 veriy/s. 可以说 OpenSSL 中基于 secp256k1 曲线的 ECDSA 实现比基于 secp256r1 曲线的 ECDSA 实现慢了一个数量级. 安全性方面, 根据 [1] 中结论显示 OpenSSL 1.1 中的基于 secp256k1 的 ECDSA 实现是不安全的.

OpenSSL 实现的速度安全问题以及 OpenSSL 版本变动可能引入的不一致问题 (参考 DER 编码和 BIP 66) 不能满足区块链场景下对速度, 安全以及稳定性的要求, Bitcoin 核心开发者在 libsep256k1 项目中重新实现了基于 secp256k1 的 ECDSA, 针对曲线 secp256k1 做了深度优化并同时保证了常量实现. 采纳基于 secp256k1 的 ECDSA 机制的区块链项目目前都默认采用 libsec256k1 中的实现.

libsep256k1 项目中采用了大量技巧以提升签名验签的执行效率, 这里我们仅关注利用 secp256k1 的自同态 (Endomorphism) 特性提升验签效率. 开始讨论技术之前, 首先感受下自同态特性的利用在 libsecp256k1 项目中带来的速度提升, 注意为了利用自同态特性加速验签过程, 需要在 `./configure` 时候指定 `--enable-endomorphism` 选项. 另外 libsecp256k1 还提供了选项 `--with-bignum=gmp|no`, 可以利用 GMP 中实现进一步提升速度. 为了利用 libsec256k1 自带的速度测试用例, 也同时需要指定 `--enable-benchmark`. Listing 1 中列出了指定不同的编译选项 (是否使用 GMP 库以及是否开启自同态选项) 时,

利用 libsecp256k1 自带的性能测试工具得到的测试数据.

Listing 1: ECDSA with secp256k1 benchmark, OpenSSL vs libsecp256k1

```
# no gmp, no endomorphism, 12330 veiry/s
secp256k1 git:(master) ./configure --enable-benchmark --with-bignum=no
secp256k1 git:(master) ./bench_verify
ecdsa_verify: min 80.0us / avg 81.1us / max 82.5us
ecdsa_verify_openssl: min 498us / avg 510us / max 522us

# with gmp, no endomorphism, 14320verify/s
secp256k1 git:(master) ./configure --enable-benchmark --with-bignum=gmp
secp256k1 git:(master) ./bench_verify
ecdsa_verify: min 68.6us / avg 69.8us / max 70.7us
ecdsa_verify_openssl: min 498us / avg 510us / max 522us

# no gmp, with endomorphism, ~15950 verify/s
secp256k1 git:(master) ./configure --enable-benchmark --with-bignum=no --enable-endomorphism
secp256k1 git:(master) ./bench_verify
ecdsa_verify: min 61.6us / avg 62.7us / max 66.5us
ecdsa_verify_openssl: min 498us / avg 510us / max 522us

# with gmp, with endomorphism, ~19800 verify/s
secp256k1 git:(master) ./configure --enable-benchmark --with-bignum=gmp --enable-endomorphism
secp256k1 git:(master) ./bench_verify
ecdsa_verify: min 49.9us / avg 50.5us / max 51.4us
ecdsa_verify_openssl: min 498us / avg 510us / max 522us
```

可以看到同时指定选项 `--enable-endomorphism` 和 `--with-bignum=gmp` 验签时, 验签速度最快, 大约 50.5us 即可完成一次验签操作, 相比不使用这两个选项时的 81.1us, 有了大约 37% 的速度提升. 仅使用 `--with-bignum=gmp` 选项时, 大约有 14% 的速度提升, 而仅使用 `--enable-endomorphism` 时, 大约有 22% 的速度提升. 值得注意的是, libsecp256k1 的速度测试工具中同时测试了 OpenSSL 中基于 secp256k1 的 ECDSA 的运行速度, 每次验签操作耗大约耗时 510us, 大约为 2000 verify/s. libsecp256k1 指定不同的编译选项时的速度, 以及 OpenSSL 中基于 secp256k1 和 secp256r1 的 ECDSA 速度对比在在 Table 1 中给出, 本文接下来关注自同态特性的原理.

早在 2011 年, Bitcoin 开发者 Hal Finney 就在 bitcointalk 论坛上指出可以利用 secp256k1 的自同态特性加速 ECDSA 的验签操作 [2], 并给出了原理示例代码. 浏览

Table 1: ECDSA 速度对比

	OpenSSL secp256r1	OpenSSL secp256k1	libsecp256k1 secp256k1 no end, no gmp	libsecp256k1 secp256k1 end, gmp
sign/s	31000	1900	20800	20800
verify/s	11600	2000	12300	19800

libsecp256k1 的代码实现可知, libsecp256k1 中的实现是以 Hal Finney 的示例为基础的, 而 Hal Finney 的原理示例则基于 [3] 中 Section 3.5 节中的内容, 而 Section 3.5 节中的内容来自论文 [6].

假设 E 是定义在域 (Field) K 上的椭圆曲线, 用 E 表示该椭圆曲线上的所有的点的集合, 则 E 上的自同态 (Endomorphism) 映射是满足

$$\phi(\mathcal{O}) = \mathcal{O}, \phi(P) = (g(P), h(P)), \forall P \in E$$

的映射 $\phi : E \to E$, 其中 g 和 h 是系数位于 K 的有理函数 (Rational Function). 如果 ϕ_1 和 ϕ_2 上自同态映射, 则定义两者的和 $\phi_1 + \phi_2$ 为 $(\phi_1 + \phi_2)(P) = \phi_1(P) + \phi_2(P)$. 定义两者的乘积 $\phi_1\phi_2$ 为 $(\phi_1\phi_2)(P) = \phi_1(\phi_2(P))$. 在这样的加法和乘法的定义下, E 上的所有的自同态映射构成一个环 (Ring), 称为域 K 上 E 的自同态环 (Endomorphism Ring of E over K). 值得注意的是, 自同态映射 ϕ 同时也是一个群同态映射 (Group Homorphism), 满足

$$\phi(P_1 + P_2) = \phi(P_1) + \phi(P_2), \forall P_1, P_2 \in E.$$

假设 p 是一个满足 $p \equiv 1 \mod 3$ 的大素数, E 则是定义在 $\mathbb{F}_p$ 上的椭圆曲线 $E : y^2 = x^3 + b$, 用 β 表示 $\mathbb{F}_p$ 上一个阶 (Order) 为 3 的元素, 则有 E 上的自同态映射

$$\phi(x, y) \to (\beta x, y), \phi(\mathcal{O}) = \mathcal{O}.$$

将 $(\beta x, y)$ 带入 E 的方程有 $y^2 = (\beta x)^3 + b = \beta^3 x^3 + b$ 可以看到 ϕ 确实是 E 到自身的映射. 由于 $order(\beta) = 3$ ($\beta^3 \equiv 1 \mod p$), 则 $y^2 = (\beta x)^3 + b = \beta^3 x^3 + b = x^3 + b$. 自同态映射 $\phi : E \to E$ 的特征多项式 (Characteristic Polynomial) 为 $X^2 + X + 1$. 一个自同态映射 ϕ 的特征多项式是 $\mathbb{Z}[X]$ 中满足 $f(\phi) = 0$(也即 $f(\phi)(P) = \mathcal{O}, \forall P \in E$) 的次数最小的首一多项式 $f(X)$. 接下来验证自同态映射 ϕ 确实满足 $f(\phi)(P) = \mathcal{O}, \forall P \in E$, 也即

$$(\phi^2 + \phi + 1)(P) = \mathcal{O}, \forall P \in E.$$

假设 $P = (x, y)$, 则根据自同态环的定义有

$$(\phi^2 + \phi + 1)(P) = \phi^2(P) + \phi(P) + 1(P) = (\beta^2 x, y) + (\beta x, y) + (x, y)$$

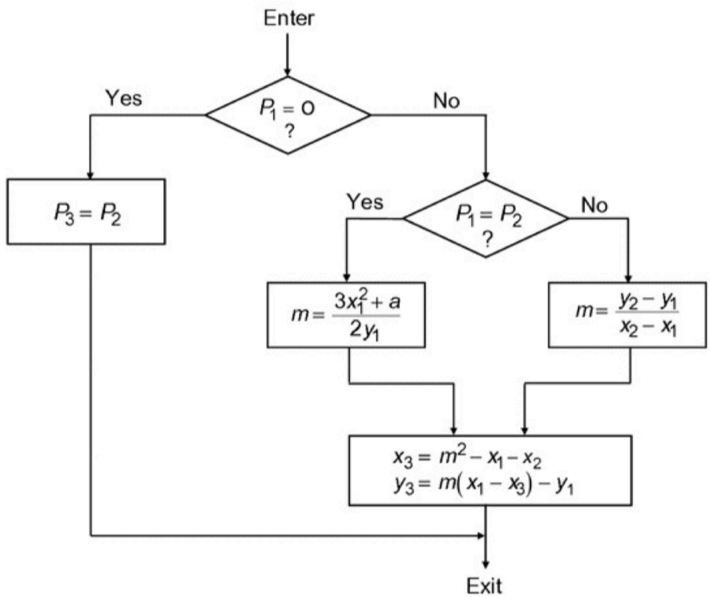

Figure 1: $y^2 = x^3 + b$ 的点加运算 $(x_3, y_3) = P_3 = P_1 + P_2 = (x_1, y_1) + (x_2, y_2)$

根据 [4] 中给出的点加运算示例, 参见 Figure 1, 有

$$(\beta^2 x, y) + (\beta x, y) + (x, y) = (-\beta^2 x - \beta x, -y) + (x, y).$$

由于 $\beta^3 \equiv 1 \mod p$, 就有 $\beta^2 + \beta + 1 = 0$, 则

$$(-\beta^2 x - \beta x, -y) + (x, y) = (-(\beta^2 + \beta)x, -y) = (x, -y) + (x, y) = \mathcal{O}.$$

用 $\#E(\mathbb{F}_p)$ 表示集合 $E(\mathbb{F}_p)$ 中点的个数, 其中 $p \equiv 1 \mod 3$ 是素数, 用 n 表示 $\#E(\mathbb{F}_p)$ 的最大素因子并且 $n^2 \nmid \#E(\mathbb{F}_p)$, 则点群 $E(\mathbb{F}_p)$ 仅包含一个阶为 n 的子群 (Subgroup), 记这个子群的基点 (Base Point) 为 $G \in E(\mathbb{F}_p)$, 记这个子群为 $\mathbb{G} = \langle G \rangle$. 假设 $\phi(G) \neq \mathcal{O}$, 则前述的映射 ϕ 相当于 $\mathbb{G}$ 上的一个点倍乘映射:

$$\phi(G) = \lambda G, \lambda^2 + \lambda + 1 \equiv 0 \mod n.$$

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

$$p = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f.$$

$\#E(\mathbb{F}_p) = h \cdot n$, 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 $\langle G\rangle$ 的基点 G 的坐标为:

$$G_x = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798$$

$$G_y = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8$$

注意到 secp256k1 的参数 $p \equiv 1 \mod 3, n|\#E(\mathbb{F}_p), n^2 \nmid \#E(\mathbb{F}_p)$, 则存在自同态映射 ϕ:

$$\phi(P) = \phi(x,y) = (\beta x, y) = \lambda P, \beta^3 \equiv 1 \mod p, \lambda^2 + \lambda + 1 \equiv 0 \mod n, \forall P \in \mathbb{G}.$$

由于

$$\lambda P = (\beta x, y) \rightarrow \lambda^2 P = (\beta^2 x, y) \rightarrow \lambda^3 P = (\beta^3 x, y) = (x,y),$$

也即 $\lambda^3 \equiv 1 \mod n$ $(\lambda^2+\lambda+1 \equiv 0 \mod n)$. 根据费马小定理 (Fermat's Little Theorem), 可以按照 Listing 2 计算 β, λ 的值.

Listing 2: `generate` β `and` λ `for endomorphism of secp256k1`

```
sage: p = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
sage: n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
sage: fp = GF(p)
sage: fn = GF(n)
sage: beta = fp(2)^((fp.characteristic()-1)/3)
sage: lamb = fn(3)^((fn.characteristic()-1)/3)
sage: hex(int(beta))
'0x7ae96a2b657c07106e64479eac3434e99cf0497512f58995c1396c28719501eeL'
sage: hex(int(lamb))
'0x5363ad4cc05c30e0a5261c028812645a122e22ea20816678df02967c1b23bd72L'
```

根据 β, λ 的具体取值以及 secp256k1 的参数, 可以用具体数值验证前述理论, e.g. $(\phi^2+\phi+1)(P) = \mathcal{O}$ 以及 $\lambda P = (\beta x, y)$, 参见 Listing 3.

Listing 3: `verify endomorphism` ϕ `with` β, λ `for secp256k1`

```
sage: secp256k1 = EllipticCurve(fp, [0,7])
sage: P = secp256k1.random_point()
sage: beta_P = int(lamb) * P
sage: beta2_P = int(lamb * lamb) * P
sage: secp256k1.is_on_curve(P.xy()[0], P.xy()[1])
True
sage: secp256k1.is_on_curve(beta_P.xy()[0], beta_P.xy()[1])
True
sage: secp256k1.is_on_curve(beta2_P.xy()[0], beta2_P.xy()[1])
True
```

```
sage: P + beta_P + beta2_P
(0 : 1 : 0)
sage: P
(41377969691400010933106372906063172361899533199914561951528196380352494104025 :
98022602531930804648533646821930292808872917654793309549219646753654843391634 :
    1)
sage: beta_P
(61065808270983873313809816182282167124533426070191311711991257449293116444501 :
98022602531930804648533646821930292808872917654793309549219646753654843391634 :
    1)
sage: beta2_P
(13348311274932311176654795920342568366837025395534690375938130178263224123137 :
98022602531930804648533646821930292808872917654793309549219646753654843391634 :
    1)
sage: beta * P.xy()[0]
61065808270983873313809816182282167124533426070191311711991257449293116444501
sage: beta * beta * P.xy()[0]
13348311274932311176654795920342568366837025395534690375938130178263224123137
```

secp256k1 上的自同态映射可以用来加速点的倍乘, 能够加速 ECDSA 的验签过程, 参见 [5] 中给出的 Figure 2. 注意签名过程中涉及到只有固定点 G 的倍乘运算 kG, 点的倍乘运算利用预计算方法可以高效计算, 无需利用自同态映射 [1]. 验证过程中, 需要计算 $es^{-1}G + rs^{-1}Q$, 其中的 $rs^{-1}Q$ 部分可以利用自同态映射进行加速, 其中 (r, s) 为签名值, e 为待签名消息的哈希值. 接下来讨论如何利用自同态 ϕ 加速非固定点的倍乘运算, 从而加速 ECDSA 的验签效率.

记待计算的点倍乘运算为 kP, 将标量 k 表示为 $k = k_1 + k_2\lambda \mod n$, 其中 k_1, k_2 的比特长度大约是 k 的一半, 则有

$$kP = k_1P + k_2\lambda P = k_1P + k_2\phi(P) = k_1P + k_2Q, Q = \phi(P)$$

其中 $Q = \phi(P)$ 仅需要一次域中的乘法运算 ($(x_Q, y_Q) = (\beta x_P, y_P)$), 而 $k_1P + k_2Q$ 的计算可以利用多点并行乘法 (Simultaneous Multiple Point Multiplication) 完成.

标量 k 的分解 $k \equiv k_1 + k_2\lambda \mod n$ 在 [6] 中有讨论, 完整的算法参见 [3] 中的 Algorithm 3.74. 分解 k 的算法, 输入为 $n, \lambda, k \in [1, n-1]$, 输出结果为 $(k_1, k_2) \in \mathbb{Z} \times \mathbb{Z}$ 满足 $k \equiv k_1 + k_2\lambda \mod n$, 并且 (k_1, k_2) 的欧几里得范数 (Euclidean Norm) 较小. 考虑定义为 $(i, j) \to i + j\lambda$ 的同态映射 $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_n$, 我们希望找到的是向量 $\mathbf{k} = (k_1, k_2) \in \mathbb{Z} \times \mathbb{Z}$ 并且满足 $f(\mathbf{k}) = k$. 注意 $\mathbf{k} = (k, 0)$ 满足上述要求, 但是这种值对我们加速点倍乘的初衷没有任何帮助, 我们需要的是具有小的欧几里得范数的向量 $\mathbf{k}$.

[1]这句需要进一步验证, 固定点计算应用自同态映射是否会更快? 直觉上不会

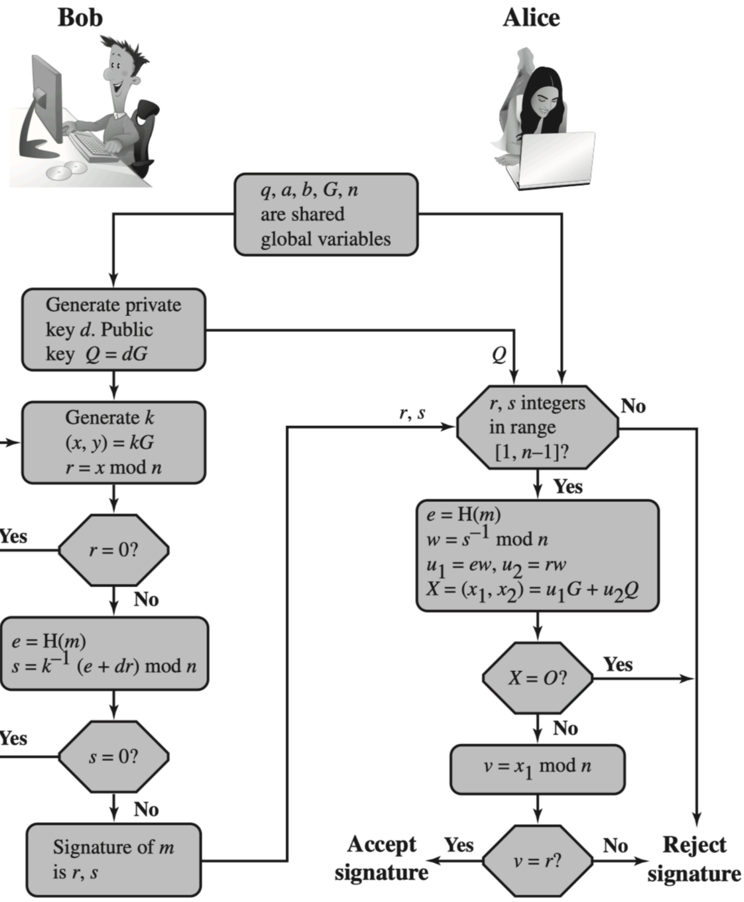

Figure 2: ECDSA 签名验签

[6] 中给出的求解思路是首先找到两个线性独立 (Linearly Independent) 的向量 $\mathbf{v_1}, \mathbf{v_2} \in \mathbb{Z} \times \mathbb{Z}$ 并且有 $f(\mathbf{v_1}) = f(\mathbf{v_2}) = 0$. 然后计算由 $\mathbf{v_1}, \mathbf{v_2}$ 生成的接近 $(k, 0)$ 的向量 $\mathbf{v} = \beta_1\mathbf{v_1} + \beta_2\mathbf{v_2}$, 则 $\mathbf{k} = (k, 0) - \mathbf{v}$ 是满足 $f(\mathbf{k}) = f((k, 0)) - f(\mathbf{v}) = k$ 的短向量.

可以利用扩展的欧几里得算法 (Extended Eculidean Algorithm) 寻找向量 $\mathbf{v_1}$ 和 $\mathbf{v_2}$. 将扩展的欧几里得算法应用到 n 和 λ, 则计算过程中会产生一系列的等式:

$$s_i n + t_i \lambda = r_i, i = 0, 1, 2, ... \tag{1}$$

其中 $s_0 = 1, t_0 = 0, r_0 = n, s_1 = 0, t_1 = 1, r_1 = \lambda$ 并且有 $r_i \geq 0$. 则有如下性质:

1. $r_i > r_{i+1} \geq 0$ for all $i \geq 0$

2. $|s_i| < |s_{i+1}|$ for all $i \geq 1$

3. $|t_i| < |t_{i+1}|$ for all $i \geq 0$

4. $r_{i-1}|t_i| + r_i|t_{i-1}| = n$ for all $i \geq 1$

第 2 点和第 3 点可以从扩展的欧几里得算法计算 s_i 和 t_i 的过程中看出.

$$s_i = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{if } i = 1, \\ s_{i-2} - q_{i-1}s_{i-1} & \text{if } i \geq 2 \end{cases} \quad t_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ t_{i-2} - q_{i-1}t_{i-1} & \text{if } i \geq 2 \end{cases}$$

可知 $s_2 = 1$, s_3 是负数, 则 $s_4 = s_2 - q_3 s_3$ 为正数, 可以看到随着 i 的增大, s_i 的正负号交替变换, 从而有 $|s_i| < |s_{i+1}|$ for all $i \geq 1$. 同理可证明 $|t_i| < |t_{i+1}|$ for all $i \geq 0$. 第 4 点性质可以通过归纳法证明. $i = 1$ 时, $r_0|t_1| + r_1|t_0| = n \cdot 1 + \lambda \cdot 0 = n$. 假设 $r_{i-1}|t_i| + r_i|t_{i-1}| = n$ 成立, 考察 $r_i|t_{i+1}| + r_{i+1}|t_i|$:

$$\begin{aligned} r_i|t_{i+1}| + r_{i+1}|t_i| =& r_i|t_{i-1} - q_i t_i| + (r_{i-1} - q_i r_i)|t_i| \\ =& r_i|t_{i-1}| + r_i q_i|t_i| + r_{i-1}|t_i| - q_i r_i|t_i| \\ =& r_i|t_{i-1}| + r_{i-1}|t_i| = n. \end{aligned}$$

记 m 为满足 $r_m \geq \sqrt{n}$ 的最大的 m, 根据上述第 4 条性质就有 $r_m|t_{m+1}| + r_{m+1}|t_m| = n$, 可推断 $|t_{m+1}| < \sqrt{n}$. 根据 Equaiton (1) 有

$$s_{m+1}n + t_{m+1}\lambda = r_{m+1} \rightarrow r_{m+1} - t_{m+1}\lambda = s_{m+1}n \equiv 0 \mod n.$$

取 $(a_1, b_1) = \mathbf{v_1} = (r_{m+1}, -t_{m+1})$, 就能满足 $f(\mathbf{v_1}) = 0$. 由于 $|t_{m+1}| < \sqrt{n}$ 以及 $|r_{m+1}| < \sqrt{n}$, 则 $\mathbf{v_1}$ 的欧几里得范数 $||\mathbf{v_1}|| < \sqrt{2n}$. 取 $(a_2, b_2) = \mathbf{v_2}$ 为 $(r_m, -t_m)$ 和 $(r_{m+2}, -t_{m+2})$ 中欧几里得范数较小的那个, 则根据 Equaiton (1) 有 $f(\mathbf{v_2}) = 0$. 直观上感觉,$\mathbf{v_2}$ 在所有满足条件的向量中也具有较小的欧几里得范数. 可以注意到 $\mathbf{v_1}$ 和 $\mathbf{v_2}$ 是相互独立的, 不失一般性假设 $\mathbf{v_2} = (r_m, -t_m)$, 则有

$$\frac{r_{m+1}}{r_m} = \frac{-t_{m+1}}{-t_m} = \frac{t_{m+1}}{t_m},$$

由于 $\frac{r_{m+1}}{r_m} < 1$ 以及 $|\frac{t_{m+1}}{t_m}| > 1$, 产生矛盾.

有了 $\mathbf{v_1}$ 和 $\mathbf{v_2}$ 之后, 构建接近 $(k, 0)$ 的向量 $\mathbf{v} = \beta_1\mathbf{v_1} + \beta_2\mathbf{v_2}$, 由于在 $\mathbb{Z} \times \mathbb{Z}$ 结构上并不一定存在 $\beta_1, \beta_2 \in \mathbb{Z}$ 能够使 $\mathbf{v} = \beta_1\mathbf{v_1} + \beta_2\mathbf{v_2}$ 成立. 所以在 $\mathbb{Q} \times \mathbb{Q}$ 结构上考虑, e.g. $\beta_1, \beta_2 \in \mathbb{Q}$, 求得 β_1, β_2 之后, 在 $\mathbb{Z}$ 中选取距离 β_1, β_2 最近的整数 c_1, c_2, 则 $\mathbf{v} = c_1\mathbf{v_1} + c_2\mathbf{v_2}$, 则期望构建的向量

$$\mathbf{k} = (k, 0) - \mathbf{v} = (k, 0) - c_1\mathbf{v_1} + c_2\mathbf{v_2},$$

其中, $k_1 = k - c_1a_1 - c_2a_2$, $k_2 = -c_1b_1 - c_2b_2$.

Figure 3: [3] 中的 Algorithm 3.74: 分解 k

Algorithm 3.74 Balanced length-two representation of a multiplier

INPUT: Integers n, λ, $k \in [0, n-1]$.
OUTPUT: Integers k_1, k_2 such that $k = k_1 + k_2\lambda \bmod n$ and $|k_1|, |k_2| \approx \sqrt{n}$.

1. Run the extended Euclidean algorithm (Algorithm 2.19) with inputs n and λ. The algorithm produces a sequence of equations $s_i n + t_i \lambda = r_i$ where $s_0 = 1$, $t_0 = 0$, $r_0 = n$, $s_1 = 0$, $t_1 = 1$, $r_1 = \lambda$, and the remainders r_i and are non-negative and strictly decreasing. Let l be the greatest index for which $r_l \geq \sqrt{n}$.
2. Set $(a_1, b_1) \leftarrow (r_{l+1}, -t_{l+1})$.
3. If $(r_l^2 + t_l^2) \leq (r_{l+2}^2 + t_{l+2}^2)$ then set $(a_2, b_2) \leftarrow (r_l, -t_l)$;
   Else set $(a_2, b_2) \leftarrow (r_{l+2}, -t_{l+2})$.
4. Compute $c_1 = \lfloor b_2 k/n \rceil$ and $c_2 = \lfloor -b_1 k/n \rceil$.
5. Compute $k_1 = k - c_1 a_1 - c_2 a_2$ and $k_2 = -c_1 b_1 - c_2 b_2$.
6. Return(k_1, k_2).

接下来考察如何计算 β_1, β_2. 由于 $(k, 0) = \beta_1(a_1, b_1) + \beta_2(a_2, b_2)$, 就有

$$\beta_1 a_1 + \beta_2 a_2 = k$$

$$\beta_1 b_1 + \beta_2 b_2 = 0$$

解方程组可以得到

$$\beta_1 = \frac{b_2 k}{a_1 b_2 - a_2 b_1}, \ \beta_2 = \frac{-b_1 k}{a_1 b_2 - a_2 b_1}.$$

由于 $(a_1, b_1) = (r_{m+1}, -t_{m+1})$, 并且不失一般性有 $(a_2, b_2) = (r_m, -t_m)$, 根据前述第 4 点性质, 以及 t_i 序列中相邻两个值的符号相反, 则有

$$|a_1 b_2 - a_2 b_1| = |-r_{m+1} t_m + t_{m+1} r_m| = n,$$

进而可以得到

$$\beta_1 = b_2 k/n, \ \beta_2 = -b_1 k/n, \ c_1 = \lfloor \beta_1 \rceil, \ c_2 = \lfloor \beta_2 \rceil.$$

根据上述讨论, 可以理解 [3] 中的 Algorithm 3.74, 参见 Figure 3.

Listing 4 中展示了用 Sage 实现的 Figure 3 中 k 分解算法, 为了验证正确性, 也同时输出了 $a_1, b_1, a_2, b_2, c_1, c_2$ 的值. 与 [2] 对比发现输出的 $a_1, b_1, a_2, b_2, c_1, c_2$ 值与 Finney 给出的一致.

Listing 4: `split_k with` λ `for secp256k1`

```
def split_k(a, b, k):
    s, old_s = 0, 1
    t, old_t = 1, 0
    r, old_r = b, a
```

```

    print "k", hex(int(k))

    # invariant r = sa + tb
    flag = false
    rt = []
    while r != 0:
        if old_r >= sqrt(a) and r < sqrt(a):
            rt.append((old_r, old_t))
            rt.append((r,t))
            flag = true

        q = floor(old_r / r)

        old_r, r = r, old_r - q * r
        old_s, s = s, old_s - q * s
        old_t, t = t, old_t - q * t

        if flag == true:
            rt.append((r,t))
            flag = false

    (r0, t0) = rt[0]
    (r1, t1) = rt[1]
    (r2, t2) = rt[2]

    a1, b1 = r1, -t1
    print "a1, b1", hex(int(a1)), hex(int(b1))

    if r0 * r0 + t0 * t0 <= r2 * r2 + t2 * t2:
        a2, b2 = r0, -t0
    else:
        a2, b2 = r2, -t2
    print "a2, b2", hex(int(a2)), hex(int(b2))

    c1, c2 = round(b2 * k / a), round(-b1 * k / a)
    print "c1, c2", hex(int(c1)), hex(int(c2))

    k1, k2 = k - c1 * a1 - c2 * a2, -c1 * b1 - c2 * b2
    print "k1, k2", hex(int(k1)), hex(int(k2))

    return k1, k2
```

为了进一步验证 k 分解的有效性, Listing 5 中对随机选取的 $k \in_R [1, n-1]$, 用 λ 进行了分解, 可以发现 `split_k` 返回的 k_1, k_2, 满足 $k_1 + k_2\lambda \equiv k \mod n$.

Listing 5: `verify multiplier split with` λ `for secp256k1`

```
sage: load("split-k.sage")
sage: k = int(fn.random_element())
sage: k1, k2 = split_k(n, int(lamb), k)
k 0x572d5beb0549d2ddcb1ff17bc516568c7f2a4d776e9c15d86e42ae55396f34f5L
a1, b1 0x3086d221a7d46bcde86c90e49284eb15L -0xe4437ed6010e88286f547fa90abfe4c3L
a2, b2 0x114ca50f7a8e2f3f657c1108d9d44cfd8L 0x3086d221a7d46bcde86c90e49284eb15L
c1, c2 0x10866a88d9692d0cb6a3e3188351f370L 0x4dbb61ed92bcc5654a28486a8793676aL
k1, k2 0x4cfc491368f8e9bb17180d76c4587555L 0x6bb2c62e9346d862c0edd4a2cf1e649eL
sage: (k1 + k2 * int(lamb)) % n
39431360339721158570738739562322678290810515502148100028669271620139548095733
sage: k
39431360339721158570738739562322678290810515502148100028669271620139548095733L
```

基于 $a_1, b_1, a_2, b_2, c_1, c_2$ 的值 Finney 在 [2] 中给出了基于 OpenSSL 的 PoC 代码. 由于 OpenSSL 版本变动导致的接口变化问题,Finney 的代码需要稍微修改才能在 OpenSSL 1.1 版本下编译通过. 调整之后的代码参见 Listing 6, 编译之后的执行的结果显示利用自同态特性有大约 16% 左右的速度提升.

Listing 6: `PoC of speeding up ECDSA verification with endomorphism`

```
static int secp256k1Verify(const unsigned char hash[32],
                           const unsigned char *dersig, size_t sigsize,
                           const EC_KEY *pkey) {
    int rslt = 0;
    const EC_GROUP *group = EC_KEY_get0_group(pkey);
    const EC_POINT *Y = EC_KEY_get0_public_key(pkey);
    const EC_POINT *G = EC_GROUP_get0_generator(group);
    EC_POINT *Glam = EC_POINT_new(group);
    EC_POINT *Ylam = EC_POINT_new(group);
    EC_POINT *R = EC_POINT_new(group);
    const EC_POINT *Points[3];
    const BIGNUM *bnexps[3];

    BN_CTX *ctx = NULL;
    BIGNUM *bnp = NULL, *bnn = NULL, *bnx = NULL, *bny = NULL;
    BIGNUM *bnk = NULL, *bnk1 = NULL, *bnk2 = NULL, *bnk1a = NULL;
```

```
    BIGNUM *bnk2a = NULL, *bnsinv = NULL, *bnh = NULL, *bnbeta = NULL;
    if (!(ctx = BN_CTX_new())) goto done;
    BN_CTX_start(ctx);
    bnp = BN_CTX_get(ctx), bnn = BN_CTX_get(ctx);
    bnx = BN_CTX_get(ctx), bny = BN_CTX_get(ctx);
    bnk = BN_CTX_get(ctx), bnk1 = BN_CTX_get(ctx), bnk2 = BN_CTX_get(ctx);
    bnk1a = BN_CTX_get(ctx), bnk2a = BN_CTX_get(ctx);
    bnsinv = BN_CTX_get(ctx), bnh = BN_CTX_get(ctx);
    if (!(bnbeta = BN_CTX_get(ctx))) goto done;

    BN_bin2bn(hash, 32, bnh);

    static unsigned char beta[] = {
        0x7a, 0xe9, 0x6a, 0x2b, 0x65, 0x7c, 0x07, 0x10, 0x6e, 0x64, 0x47,
        0x9e, 0xac, 0x34, 0x34, 0xe9, 0x9c, 0xf0, 0x49, 0x75, 0x12, 0xf5,
        0x89, 0x95, 0xc1, 0x39, 0x6c, 0x28, 0x71, 0x95, 0x01, 0xee,
    };
    BN_bin2bn(beta, 32, bnbeta);
    ECDSA_SIG *sig = d2i_ECDSA_SIG(NULL, &dersig, sigsize);

    if (sig == NULL) goto done;

    EC_GROUP_get_curve_GFp(group, bnp, NULL, NULL, ctx);
    EC_GROUP_get_order(group, bnn, ctx);

    if (BN_is_zero(ECDSA_SIG_get0_r(sig)) ||
        BN_is_negative(ECDSA_SIG_get0_r(sig)) ||
        BN_ucmp(ECDSA_SIG_get0_r(sig), bnn) >= 0 ||
        BN_is_zero(ECDSA_SIG_get0_s(sig)) ||
        BN_is_negative(ECDSA_SIG_get0_s(sig)) ||
        BN_ucmp(ECDSA_SIG_get0_s(sig), bnn) >= 0)
        goto done;

    // bnx = Gx, bny = Gy
    EC_POINT_get_affine_coordinates_GFp(group, G, bnx, bny, ctx);
    BN_mod_mul(bnx, bnx, bnbeta, bnp, ctx);  // bnx = bnx * beta mod p
    // Glam = (beta*x, y)
    EC_POINT_set_affine_coordinates_GFp(group, Glam, bnx, bny, ctx);
    EC_POINT_get_affine_coordinates_GFp(group, Y, bnx, bny, ctx);
    BN_mod_mul(bnx, bnx, bnbeta, bnp, ctx);
    EC_POINT_set_affine_coordinates_GFp(group, Ylam, bnx, bny, ctx);  // Ylam

```

```
    Points[0] = Glam;
    Points[1] = Y;
    Points[2] = Ylam;
    // sinv = s^-1 mod n
    BN_mod_inverse(bnsinv, ECDSA_SIG_get0_s(sig), bnn, ctx);
    BN_mod_mul(bnk, bnh, bnsinv, bnn, ctx);  // bnk = h * s^-1
    splitk(bnk1, bnk2, bnk, bnn, ctx);
    bnexps[0] = bnk2;
    BN_mod_mul(bnk, ECDSA_SIG_get0_r(sig), bnsinv, bnn, ctx);  // bnk = r * s^-1
    splitk(bnk1a, bnk2a, bnk, bnn, ctx);
    bnexps[1] = bnk1a;
    bnexps[2] = bnk2a;

    EC_POINTs_mul(group, R, bnk1, 3, Points, bnexps, ctx);
    EC_POINT_get_affine_coordinates_GFp(group, R, bnx, NULL, ctx);
    BN_mod(bnx, bnx, bnn, ctx);
    rslt = (BN_cmp(bnx, ECDSA_SIG_get0_r(sig)) == 0);

    ECDSA_SIG_free(sig);
done:
    if (ctx) {
        BN_CTX_end(ctx);
        BN_CTX_free(ctx);
    }
    EC_POINT_free(Glam);
    EC_POINT_free(Ylam);
    EC_POINT_free(R);

    return rslt;
}
```

libsecp256k1 中仅使用 `--enable-endomorphism` 选项时 22% 的速度提升整合了更多的优化技巧, 例如通过预计算避免在分解 k 的时候执行除法运算等, 留作以后分析.

References

[1] Genkin, Daniel, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ”ECDSA key extraction from mobile devices via nonintrusive physical side channels.” In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1626-1638. ACM, 2016.

[2] Hal Finney. bitcointalk - Speeding up signature verification. 2011. https://bitcointalk.org/index.php?topic=3238.0

[3] Hankerson, Darrel, Alfred J. Menezes, and Scott Vanstone. ”Guide to elliptic curve cryptography.” Computing Reviews 46, no. 1 (2005): 13.

[4] Blahut, Richard E. Cryptography and secure communication. Cambridge University Press, 2014.

[5] Stallings, William. Cryptography and network security: principles and practice. Upper Saddle River: Pearson, 2017.

[6] 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.