uECC解析:

uecc/uECC.c:

椭圆曲线满足的条件(韦尔斯特拉斯标准型):

y^2 = x^3 + a\*x + b where 4\*a^3 + 27\*b^2 != 0

群：

一个集合与一个二元运算符@构成的含逆元的群

交换群(阿贝尔群):

1封闭性: a,b 属于G，a@b也属于G

2结合律（即正常的对@操作符结合运算，可修改结合顺序）

3存在单位元（a@0 == 0@a == 0）

4所有元素均有逆元（任意的元素都有a@b==0）

5交换律（加上此条才满足交换群，否则就只算群）

群的更多定理:

如单位元唯一，逆元唯一等

那么因为椭圆曲线是以X轴对称的曲线（曲线而非椭圆，椭圆不是曲线），那么可建立在椭圆曲线上的群:

1单位元为无穷远点0（群定义3）

2逆元为关于x轴对称的点（群定义4）

3定义二元运算符+（群的描述，二元运算符@ == +，它现在并不是我们常规理解的 +运算符，仅仅是一个符号）

4三个非0点共线（a + b + c == 0）（群定义2,5）

(a+b)+c == a+(b+c) == b+(a+c) == 0

现在椭圆曲线群构造完毕，自定义的加法+如何确定？它并不是我们常规理解的几何加法+，那么现在认定这是一个几何+（这不是代数加法，不是1+1==2，只是逻辑关系的一种描述，你仍然可以看做@）

那么对于阿贝尔群（具有交换律）有任意 当abc三点共线时，那么根据群的特点我们可以知道，c + (-c) == 0（关于x轴对称）:

a + b == -c

也即在xy轴上任意ab俩点的连线过第三点（我们假设第三点在中间），那么它关于x轴对称的点-c的值为a + b

现在如果a == 0或b == 0时，根据群定义3，那么a + b == a或b，很显然依然符合群的完备性（任然满足椭圆群定义2）

如果a + b == 0时，那么根据椭圆曲线群的特点（椭圆群定义2），那么无法交于第三点，则，a + b == a + (-a)==》b == -a。此时不与第三点交合，满足椭圆群定义4

现在如果a与b当在条件 a + b == -c时，a与b无穷接近时，过a或b（切线的极限思想）的线会变成一条切线，此时 a+a == -c 或 b+b ==-c

若a!=b且不存在c？ 其实也是上述情况，是c与a或b的任意一点无穷接近，依然会形成切线

现在这个几何的+已经具备了完备性，假定的各种情况在几何上具备了完备性（逻辑层面），那么现在我们需要转化成代数+（因为我们需要到算法），但下面式子具体推演不知道，据说要用到三次方程的解算，但上述的式子是正确的，证明推演毕竟这不是我们需要了解的原理主体。

对于a!=b!=0时（一般情况）:

俩点之间的直线方程变换：

m == (Ya-Yb)/(Xa-Xb)

Xc==m ^ 2 - Xa - Xb

Yc==Ya + m \* (Xc - Xa) or

Yc==Yb + m \* (Xc - Xb)

对于a==b时：

(这里是椭圆的参数a，不是上面的变量a，命名失误)

m == (3\*Xa^2 + a)/(2\*Ya)

Ya==+-(Xa ^ 3 + a \* Xa + b)^(1 / 2)（开根号）

那么有了加法，我们可以定义常规的\*为n个A加等于n \* A

（快速加法）

！对数问题，上述乘法运算的出现自然而然的可产生对数运算，而这就是问题所在乘法运算已知任意的俩个量可随意求出第三个量，但对数不行，即使该运算被置入到椭圆曲线群这一关离散的域中。根据椭圆曲线方程所定义的平面曲线的有限域上，满足条件的特殊点集合构成的上述的椭圆曲线群，在n\*p = q的乘法运算情况下，已知p和q很难求出n的值

！这是非对称加密的核心，（参见RSA加密，它的难点是大质数分解极度困难或可能无法实现，所以可采用搜索俩个大质数，在他们满足一定条件下时，形成公钥和私钥）

//uECC.h中的宏定义最终配置了uECC\_BYTES 32

//定义的不同种类分别适应于不同的平台

//工程应使用uECC\_WORD\_SIZE == 4的配置

//uECC.c中定义的以下数据集合指定了椭圆曲线的离散点，它满足椭圆曲线群的特点

#define Curve\_P\_

#define Curve\_G\_

#define Curve\_N\_

//该宏实现了联立字符串，参见它对应的.h

#define uECC\_WORDS uECC\_CONCAT(uECC\_WORDS\_, uECC\_CURVE)

#define uECC\_N\_WORDS uECC\_CONCAT(uECC\_N\_WORDS\_, uECC\_CURVE)

//它记录了点的大小

typedef struct EccPoint {

uECC\_word\_t x[uECC\_WORDS];

uECC\_word\_t y[uECC\_WORDS];

} EccPoint;

//这里指定了该数组首地址被绑定到椭圆曲线群的哪个对应的集合中

//工程上是使用的Curve\_X\_3应该是四字节对应的

static const uECC\_word\_t curve\_p[uECC\_WORDS] = uECC\_CONCAT(Curve\_P\_, uECC\_CURVE);

static const uECC\_word\_t curve\_b[uECC\_WORDS] = uECC\_CONCAT(Curve\_B\_, uECC\_CURVE);

static const uECC\_word\_t curve\_n[uECC\_N\_WORDS] = uECC\_CONCAT(Curve\_N\_, uECC\_CURVE);

static const EccPoint curve\_G = uECC\_CONCAT(Curve\_G\_, uECC\_CURVE);

//确定随机函数Random从哪里来

#if (defined(\_WIN32) || defined(\_WIN64))

#define WIN32\_LEAN\_AND\_MEAN

#include <windows.h>

#include <wincrypt.h>

//使用Windows提供的系统接口获取随机数

static int default\_RNG(uint8\_t \*dest, unsigned size) ;

//使用UNIX提供的伪随机数参数表去获取随机数

#elif defined(unix) || defined(\_\_linux\_\_) || defined(\_\_unix\_\_) || defined(\_\_unix) || \

(defined(\_\_APPLE\_\_) && defined(\_\_MACH\_\_)) || defined(uECC\_POSIX)

/\* Some POSIX-like system with /dev/urandom or /dev/random. \*/

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

//查表获取伪随机数

static int default\_RNG(uint8\_t \*dest, unsigned size);

#else /\* Some other platform \*/

//其他平台需要默认自定义伪随机数

static int default\_RNG(uint8\_t \*dest, unsigned size) {

return 0;

}

#endif

static uECC\_RNG\_Function g\_rng\_function = &default\_RNG;

//但是可以在这里自己重定向自己编写的随机数算法

void uECC\_set\_rng(uECC\_RNG\_Function rng\_function) {

g\_rng\_function = rng\_function;

}

//程序自带了许多基础的快速运算的函数调用

//因为程序自定义的默认数据是兼容各种字节的数据操作

//所以这个系列的快速运算都是基于Byte去做这个事情的，因为还涉及到

//保存的大小端不同，根据不同的平台运算方式是不一样的，运算顺序也是不一样的

//同样的脱开原理外，里面出现了大量的公式的变现，主要的目的都是计算

//P=[k]Q的操作，也即是上文介绍的标量的乘法的操作，操作及其复杂

//多余的舍去

//获取一个钥匙对，公钥在工程中是64 \* 8位，私钥是32 \* 8位

int uECC\_make\_key(uint8\_t public\_key[uECC\_BYTES\*2], uint8\_t private\_key[uECC\_BYTES]) {

//主要因为C语言平台是没有private和public俩个关键字的

//但C++是存在俩个关键字的，所以在不同语言下需要注意

uECC\_word\_t private[uECC\_WORDS];

EccPoint public;

uECC\_word\_t tries;

for (tries = 0; tries < MAX\_TRIES; ++tries) {

//循环多次去生成，是为了增加随机性？

//私钥使用随机函数获取，要么查表要么使用类似时间值等

if (g\_rng\_function((uint8\_t \*)private, sizeof(private)) &&

//公钥调用公钥算法，见下，但注意到的是公钥需要私钥生成

EccPoint\_compute\_public\_key(&public, private)) {

vli\_nativeToBytes(private\_key, private);

vli\_nativeToBytes(public\_key, public.x);

vli\_nativeToBytes(public\_key + uECC\_BYTES, public.y);

return 1;

}

}

return 0;

}

//公钥生成（以项目为例，删除了一些不可能出现的配置）

static int EccPoint\_compute\_public\_key(EccPoint \*result, uECC\_word\_t \*private) {

#if (uECC\_CURVE != uECC\_secp160r1)

uECC\_word\_t tmp1[uECC\_WORDS];

uECC\_word\_t tmp2[uECC\_WORDS];

uECC\_word\_t \*p2[2] = {tmp1, tmp2};

uECC\_word\_t carry;

#endif

//你随机生成的私钥是否合法

if (vli\_isZero(private)) {

return 0;

}

//当curve\_n <= 私钥时

if (vli\_cmp(curve\_n, private) != 1) {

return 0;

}

//tmp1 == private + curve\_n，carry是进位

carry = vli\_add(tmp1, private, curve\_n);

//tmp2 == tmp1 + curve\_n

vli\_add(tmp2, tmp1, curve\_n);

//对离散域中一个点进行倍乘运算

//Q=[k]P，其他算法不需要太关注，只需要知道

//无论是倍乘前还是倍乘后，最终的各个值都必须要是域中的值

//最终计算 result == [p2[!carry]]curve\_G

//标量使用的是被private修饰过的curve\_n

EccPoint\_mult(result, &curve\_G, p2[!carry], 0, (uECC\_BYTES \* 8) + 1);

//确认公钥是否有效

if (EccPoint\_isZero(result)) {

return 0;

}

return 1;

}

//共享秘钥的生成

int uECC\_shared\_secret(const uint8\_t public\_key[uECC\_BYTES\*2],

const uint8\_t private\_key[uECC\_BYTES],

uint8\_t secret[uECC\_BYTES]) {

EccPoint public;

EccPoint product;

uECC\_word\_t private[uECC\_WORDS];

uECC\_word\_t random[uECC\_WORDS];

uECC\_word\_t \*initial\_Z = 0;

uECC\_word\_t tries;

#if (uECC\_CURVE != uECC\_secp160r1)

uECC\_word\_t tmp[uECC\_WORDS];

uECC\_word\_t \*p2[2] = {private, tmp};

uECC\_word\_t carry;

#endif

//获取一个随机的修饰值

for (tries = 0; tries < MAX\_TRIES; ++tries) {

if (g\_rng\_function((uint8\_t \*)random, sizeof(random)) && !vli\_isZero(random)) {

initial\_Z = random;

break;

}

}

vli\_bytesToNative(private, private\_key);

vli\_bytesToNative(public.x, public\_key);

vli\_bytesToNative(public.y, public\_key + uECC\_BYTES);

carry = vli\_add(private, private, curve\_n);

vli\_add(tmp, private, curve\_n);

//在本端计算公私钥未指定修饰值，因为公钥来源于外界，所以添加修饰

//该修饰值将会参与到矩阵变换与离散点的变换

EccPoint\_mult(&product, &public, p2[!carry], initial\_Z, (uECC\_BYTES \* 8) + 1);

vli\_nativeToBytes(secret, product.x);

return !EccPoint\_isZero(&product);

}

//数字签名

int uECC\_sign(const uint8\_t private\_key[uECC\_BYTES],

const uint8\_t message\_hash[uECC\_BYTES],

uint8\_t signature[uECC\_BYTES\*2]) {

uECC\_word\_t k[uECC\_N\_WORDS];

uECC\_word\_t tries;

for (tries = 0; tries < MAX\_TRIES; ++tries) {

//获取一个随机数作为标量乘法的”系数”

if(g\_rng\_function((uint8\_t \*)k, sizeof(k))) {

if (uECC\_sign\_with\_k(private\_key, message\_hash, k, signature)) {

return 1;

}

}

}

return 0;

}

//计算数字签名

static int uECC\_sign\_with\_k(const uint8\_t private\_key[uECC\_BYTES],

const uint8\_t message\_hash[uECC\_BYTES],

uECC\_word\_t k[uECC\_N\_WORDS],

uint8\_t signature[uECC\_BYTES\*2]) {

uECC\_word\_t tmp[uECC\_N\_WORDS];

uECC\_word\_t s[uECC\_N\_WORDS];

uECC\_word\_t \*k2[2] = {tmp, s};

EccPoint p;

uECC\_word\_t carry;

uECC\_word\_t tries;

//确认k的合法性

if (vli\_isZero(k) || vli\_cmp\_n(curve\_n, k) != 1) {

return 0;

}

//使用curve\_n修饰k

carry = vli\_add(tmp, k, curve\_n);

vli\_add(s, tmp, curve\_n);

//计算p = [k2[!carry]]curve\_G，无修饰

EccPoint\_mult(&p, &curve\_G, k2[!carry], 0, (uECC\_BYTES \* 8) + 1);

//继续修饰p.x

if (vli\_cmp(curve\_n, p.x) != 1) {

vli\_sub(p.x, p.x, curve\_n);

}

//合法性判断，关于点p

if (vli\_isZero(p.x)) {

return 0;

}

//挖一个随机数去帮助修饰k，防止外界猜到k的值

carry = 0;

for (tries = 0; tries < MAX\_TRIES; ++tries) {

if (!g\_rng\_function((uint8\_t \*)tmp, sizeof(tmp))) {

continue;

}

carry = 1;

//找一个tmp

if (!vli\_isZero(tmp)) {

break;

}

}

//实在不行指定一个非零值

if (!carry) {

vli\_clear(tmp);

tmp[0] = 1;

}

//用刚才出来的随机值tmp去修饰k

vli\_modMult\_n(k, k, tmp); /\* k' = rand \* k \*/

vli\_modInv\_n(k, k, curve\_n); /\* k = 1 / k' \*/

vli\_modMult\_n(k, k, tmp); /\* k = 1 / k \*/

//横坐标x填入数字签名中

vli\_nativeToBytes(signature, p.x);

//继续修饰k=》s

tmp[uECC\_N\_WORDS - 1] = 0;

vli\_bytesToNative(tmp, private\_key); /\* tmp = d \*/

s[uECC\_N\_WORDS - 1] = 0;

vli\_set(s, p.x);

vli\_modMult\_n(s, tmp, s); /\* s = r\*d \*/

vli\_bytesToNative(tmp, message\_hash);

vli\_modAdd\_n(s, tmp, s, curve\_n); /\* s = e + r\*d \*/

vli\_modMult\_n(s, s, k); /\* s = (e + r\*d) / k \*/

#if (uECC\_CURVE == uECC\_secp160r1)

if (s[uECC\_N\_WORDS - 1]) {

return 0;

}

#endif

//将使用随机值修饰好的s，写入数字签名

vli\_nativeToBytes(signature + uECC\_BYTES, s);

return 1;

}

//HMAC为K进行哈希散列修饰

//为哈希值生成数字签名，但需要具体的HASH实例操作

int uECC\_sign\_deterministic(const uint8\_t private\_key[uECC\_BYTES],

const uint8\_t message\_hash[uECC\_BYTES],

uECC\_HashContext \*hash\_context,

uint8\_t signature[uECC\_BYTES\*2]);

//确认数字签名

int uECC\_verify(const uint8\_t public\_key[uECC\_BYTES\*2],

const uint8\_t hash[uECC\_BYTES],

const uint8\_t signature[uECC\_BYTES\*2]) {

uECC\_word\_t u1[uECC\_N\_WORDS], u2[uECC\_N\_WORDS];

uECC\_word\_t z[uECC\_N\_WORDS];

EccPoint public, sum;

uECC\_word\_t rx[uECC\_WORDS];

uECC\_word\_t ry[uECC\_WORDS];

uECC\_word\_t tx[uECC\_WORDS];

uECC\_word\_t ty[uECC\_WORDS];

uECC\_word\_t tz[uECC\_WORDS];

const EccPoint \*points[4];

const EccPoint \*point;

bitcount\_t numBits;

bitcount\_t i;

uECC\_word\_t r[uECC\_N\_WORDS], s[uECC\_N\_WORDS];

r[uECC\_N\_WORDS - 1] = 0;

s[uECC\_N\_WORDS - 1] = 0;

vli\_bytesToNative(public.x, public\_key);//公钥中保存的被修饰的坐标

vli\_bytesToNative(public.y, public\_key + uECC\_BYTES);

vli\_bytesToNative(r, signature);//数字签名前一部分是P.x

vli\_bytesToNative(s, signature + uECC\_BYTES);//后一部分是被修饰的k

if (vli\_isZero(r) || vli\_isZero(s)) { /\* r, s must not be 0. \*/

return 0;

}

//反修饰，数学公式未知

/\* Calculate u1 and u2. \*/

vli\_modInv\_n(z, s, curve\_n); /\* Z = s^-1 \*/

u1[uECC\_N\_WORDS - 1] = 0;

vli\_bytesToNative(u1, hash);

vli\_modMult\_n(u1, u1, z); /\* u1 = e/s \*/

vli\_modMult\_n(u2, r, z); /\* u2 = r/s \*/

/\* Calculate sum = G + Q. \*/

vli\_set(sum.x, public.x);

vli\_set(sum.y, public.y);

vli\_set(tx, curve\_G.x);

vli\_set(ty, curve\_G.y);

vli\_modSub\_fast(z, sum.x, tx); /\* Z = x2 - x1 \*/

XYcZ\_add(tx, ty, sum.x, sum.y);

vli\_modInv(z, z, curve\_p); /\* Z = 1/Z \*/

apply\_z(sum.x, sum.y, z);

/\* Use Shamir's trick to calculate u1\*G + u2\*Q \*/

points[0] = 0;

points[1] = &curve\_G;

points[2] = &public;

points[3] = &sum;

numBits = smax(vli\_numBits(u1, uECC\_N\_WORDS), vli\_numBits(u2, uECC\_N\_WORDS));

point = points[(!!vli\_testBit(u1, numBits - 1)) | ((!!vli\_testBit(u2, numBits - 1)) << 1)];

vli\_set(rx, point->x);

vli\_set(ry, point->y);

vli\_clear(z);

z[0] = 1;

for (i = numBits - 2; i >= 0; --i) {

uECC\_word\_t index;

EccPoint\_double\_jacobian(rx, ry, z);

index = (!!vli\_testBit(u1, i)) | ((!!vli\_testBit(u2, i)) << 1);

point = points[index];

if (point) {

vli\_set(tx, point->x);

vli\_set(ty, point->y);

apply\_z(tx, ty, z);

vli\_modSub\_fast(tz, rx, tx); /\* Z = x2 - x1 \*/

XYcZ\_add(tx, ty, rx, ry);

vli\_modMult\_fast(z, z, tz);

}

}

vli\_modInv(z, z, curve\_p); /\* Z = 1/Z \*/

apply\_z(rx, ry, z);

/\* Accept only if v == r. \*/

return vli\_equal(rx, r);

}

//计算公钥的合法性，对椭圆曲线的计算流程的指导很有帮助

int uECC\_valid\_public\_key(const uint8\_t public\_key[uECC\_BYTES\*2]) {

uECC\_word\_t tmp1[uECC\_WORDS];

uECC\_word\_t tmp2[uECC\_WORDS];

EccPoint public;

vli\_bytesToNative(public.x, public\_key);

vli\_bytesToNative(public.y, public\_key + uECC\_BYTES);

// The point at infinity is invalid.

if (EccPoint\_isZero(&public)) {

return 0;

}

// x and y must be smaller than p.

if (vli\_cmp(curve\_p, public.x) != 1 || vli\_cmp(curve\_p, public.y) != 1) {

return 0;

}

vli\_modSquare\_fast(tmp1, public.y); /\* tmp1 = y^2 \*/

curve\_x\_side(tmp2, public.x); /\* tmp2 = x^3 + ax + b \*/

/\* Make sure that y^2 == x^3 + ax + b \*/

return (vli\_cmp(tmp1, tmp2) == 0);

}

//压缩公钥，进行偏移计算

void uECC\_compress(const uint8\_t public\_key[uECC\_BYTES\*2],

uint8\_t compressed[uECC\_BYTES+1]) {

wordcount\_t i;

for (i = 0; i < uECC\_BYTES; ++i) {

compressed[i+1] = public\_key[i];

}

//保留公钥的横坐标x和纵坐标的某一位

compressed[0] = 2 + (public\_key[uECC\_BYTES \* 2 - 1] & 0x01);

}

//通常来说，压缩与解压缩是互逆的动作

void uECC\_decompress(const uint8\_t compressed[uECC\_BYTES+1],

uint8\_t public\_key[uECC\_BYTES\*2]) {

EccPoint point;

//取出压缩后的公钥中的横坐标x，然后计算纵坐标y

vli\_bytesToNative(point.x, compressed + 1);

curve\_x\_side(point.y, point.x);

mod\_sqrt(point.y);

//校验

if ((point.y[0] & 0x01) != (compressed[0] & 0x01)) {

vli\_sub(point.y, curve\_p, point.y);

}

vli\_nativeToBytes(public\_key, point.x);

vli\_nativeToBytes(public\_key + uECC\_BYTES, point.y);

}