二元域椭圆曲线加密算法 (ECC)

C 语言实现

dajiangwan@gmail.com

2013-9-22

1. 前言

二元域的椭圆加密算法在工作中会遇到，由于该算法涉及到许多数学理论，要理解算法不是太容易。

ECC 在 PC 机上有很多实现，比如 openssl、tomcrypto 以及 gcrypto 等；以上的算法由于要考虑运行效率，使用了很多加速算法，导致阅读时不易理解。

为了理解该算法的关键部分，本文依照 RSA 提供的大数算法，对 TinyBECC 的源码进行整理修改,实现了 ECC 加密算法的 ECDSA。该代码可以在 Windows 下使用 Microsoft VC 6.0 进行编译，也可以在 Fedora Linux 下的使用 gcc 进行编译。为了验证算法的正确性，可以通过 openssl 进行对比测试。

作为一个示例性的程序，性能较差，只能用来了解 ECC 算法的原理。

2. BN 定义

ECC 使用大数进行运算，在计算机表示大数时，一般采用多个字节来存储。在 C 语言中使用整数数组来存储大数。以下代码假设计算机的整数宽度为 32Bit, 存储时先存储低端数据，后存储高端数据。以下代码是从 RSA 提供的 RSA 算法实现参考中(BN.h 和 BN.c)复制出来的，然后进行了一些修改。以下代码定义一些宏、数据类型以及函数原型。

```
#ifndef _BN_H
#define _BN_H

typedef unsigned char           uint8_t;
typedef unsigned short          uint16_t;
typedef unsigned int            uint32_t;
typedef signed char             int8_t;
typedef short                   int16_t;
typedef int                     int32_t;

#define NN_HALF_DIGIT_BITS      16
#define NN_DIGIT_BITS           32
#define MAX_NN_HALF_DIGIT       0xffff
#define MAX_NN_DIGIT            0xffffffff
#define NN_DIGIT_LEN            (NN_DIGIT_BITS / 8)
typedef uint16_t                NN_HALF_DIGIT;
typedef uint32_t                NN_DIGIT;

#define LOW_HALF(x)             ((x) & MAX_NN_HALF_DIGIT)
#define HIGH_HALF(x)            \
    (((x)>>NN_HALF_DIGIT_BITS)& AX_NN_HALF_DIGIT)
#define TO_HIGH_HALF(x)         (((NN_DIGIT)(x)) << NN_HALF_DIGIT_BITS)
#define DIGIT_MSB(x)            (NN_DIGIT)(((x) >> (NN_DIGIT_BITS - 1)) & 1)
#define DIGIT_2MSB(x)           (NN_DIGIT)(((x) >> (NN_DIGIT_BITS - 2)) & 3)

#define MAX_NN_DIGIT_BITS       1024
#define MAX_NN_DIGITS           \
    ((MAX_NN_DIGIT_BITS+NN_DIGIT_LEN-1)/NN_DIGIT_LEN + 1)

#ifndef MIN
    #define MIN(a, b) ((a) <= (b) ? (a) : (b))
    #define MAX(a, b) ((a) >= (b) ? (a) : (b))
#endif
```

图 1. 大数头文件 bn.h (第 1 部分)

```
uint32_t NN_Digits(NN_DIGIT *a, uint32_t digits);

uint32_t NN_Bits(NN_DIGIT *a, uint32_t digits);

/* Encode BIG NUMBER b into octet string a */
void NN_Encode(uint8_t *a, uint32_t len, NN_DIGIT *b, uint32_t digits);

/* Decode octet string a to BIG NUMBER b */
void NN_Decode(NN_DIGIT *a, uint32_t digits, uint8_t *b, uint32_t len);

void NN_Assign(NN_DIGIT *a, NN_DIGIT *b, uint32_t digits);

void NN_AssignZero(NN_DIGIT *a, uint32_t digits);

void NN_Assign2Exp(NN_DIGIT *a, uint32_t b, uint32_t digits);

#define NN_ASSIGN_DIGIT(a, b, digits)    {NN_AssignZero (a, digits); a[0] = b;}

#define NN_EQUAL(a, b, digits)           (! NN_Cmp (a, b, digits))

#define NN_EVEN(a, digits)               (((digits) == 0) || ! (a[0] & 1))

int NN_Cmp(NN_DIGIT *a, NN_DIGIT *b, uint32_t digits);

int NN_Is_Zero(NN_DIGIT *a, uint32_t digits);

NN_DIGIT NN_Add(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits);

NN_DIGIT NN_Sub(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits);

void NN_Mult(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits);

NN_DIGIT NN_LShift(NN_DIGIT *a, NN_DIGIT *b, uint32_t c, uint32_t digits);

NN_DIGIT NN_RShift(NN_DIGIT *a, NN_DIGIT *b, uint32_t c, uint32_t digits);
```

图 2. 大数头文件 bn.h (第 2 部分)

```
void NN_Div(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t cDigits,
     NN_DIGIT *d, uint32_t dDigits);

void NN_Mod(NN_DIGIT *a, NN_DIGIT *b, uint32_t bDigits, NN_DIGIT *c,
     uint32_t cDigits);

void NN_ModMult(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, NN_DIGIT *d,
     uint32_t digits);

void NN_ModExp(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t cDigits,
     NN_DIGIT *d, uint32_t dDigits);

void NN_ModInv(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits);

void NN_Gcd(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits);

#endif // _BN_H
```

图 3. 大数头文件 bn.h (第 3 部分)

3. BN 算法

3.1. 单宽度整数乘法

```
static void NN_DigitMult(NN_DIGIT a[2], NN_DIGIT b, NN_DIGIT c)
{
	NN_DIGIT t, u;
	NN_HALF_DIGIT bHigh, bLow, cHigh, cLow;

	bHigh = (NN_HALF_DIGIT)HIGH_HALF (b);
	bLow = (NN_HALF_DIGIT)LOW_HALF (b);
	cHigh = (NN_HALF_DIGIT)HIGH_HALF (c);
	cLow = (NN_HALF_DIGIT)LOW_HALF (c);
	a[0] = (NN_DIGIT)bLow * (NN_DIGIT)cLow;
	t = (NN_DIGIT)bLow * (NN_DIGIT)cHigh;
	u = (NN_DIGIT)bHigh * (NN_DIGIT)cLow;
	a[1] = (NN_DIGIT)bHigh * (NN_DIGIT)cHigh;
	if ((t += u) < u)
		a[1] += TO_HIGH_HALF (1);
	u = TO_HIGH_HALF (t);
	if ((a[0] += u) < u)
		a[1] ++;
	a[1] += HIGH_HALF (t);
}
```

图 4. 单宽度整数乘法

两个 32Bit 的整数相乘,结果为 64Bit 的整数。使用汇编语言编写时，如果乘法结果为两个 32Bit 寄存器保存，不考虑其他的因数，则可以简化为一条乘法指令。如此则效率会提高很多。

3.2. 单宽度整数除法

```
static void NN_DigitDiv(NN_DIGIT *a, NN_DIGIT b[2], NN_DIGIT c)
{
    NN_DIGIT t[2], u, v;
    NN_HALF_DIGIT aHigh, aLow, cHigh, cLow;

    cHigh = (NN_HALF_DIGIT)HIGH_HALF (c);
    cLow = (NN_HALF_DIGIT)LOW_HALF (c);
    t[0] = b[0];
    t[1] = b[1];
    if (cHigh == MAX_NN_HALF_DIGIT)
        aHigh = (NN_HALF_DIGIT)HIGH_HALF (t[1]);
    else
        aHigh = (NN_HALF_DIGIT)(t[1] / (cHigh + 1));
    u = (NN_DIGIT)aHigh * (NN_DIGIT)cLow;
    v = (NN_DIGIT)aHigh * (NN_DIGIT)cHigh;
    if ((t[0] -= TO_HIGH_HALF (u)) > (MAX_NN_DIGIT - TO_HIGH_HALF (u)))
        t[1]--;
    t[1] -= HIGH_HALF (u);
    t[1] -= v;
    while ((t[1] > (NN_DIGIT)cHigh) ||
        ((t[1] == (NN_DIGIT)cHigh) && (t[0] >= TO_HIGH_HALF (cLow)))) {
        if ((t[0] -= TO_HIGH_HALF (cLow)) > MAX_NN_DIGIT - TO_HIGH_HALF (cLow))
            t[1] --;
        t[1] -= cHigh;
        aHigh ++;
    }
    if (cHigh == MAX_NN_HALF_DIGIT)
        aLow = (NN_HALF_DIGIT)LOW_HALF (t[1]);
    else
        aLow =
            (NN_HALF_DIGIT)((TO_HIGH_HALF (t[1]) + HIGH_HALF (t[0])) / (cHigh + 1));
    u = (NN_DIGIT)aLow * (NN_DIGIT)cLow;
    v = (NN_DIGIT)aLow * (NN_DIGIT)cHigh;
    if ((t[0] -= u) > (MAX_NN_DIGIT - u))    t[1]--;
    if ((t[0] -= TO_HIGH_HALF (v)) > (MAX_NN_DIGIT - TO_HIGH_HALF (v)))
        t[1]--;
    t[1] -= HIGH_HALF (v);
```

图 5. 单宽度整数除法（第 1 部分）

```
    while ((t[1] > 0) || ((t[1] == 0) && t[0] >= c)) {
        if ((t[0] -= c) > (MAX_NN_DIGIT - c))
            t[1]--;
        aLow++;
    }
    *a = TO_HIGH_HALF (aHigh) + aLow;
}
```

图 6. 单宽度整数除法（第 2 部分）

一个 64Bit 的整数除以一个 32Bit 的整数，结果为 32Bit 的整数。使用汇编语言编写时，如果被除数可以表示为两个 32Bit 寄存器保存，不考虑其他的，则可以简化为一条除法指令。

3.3. 中间计算结果函数

```
static NN_DIGIT NN_AddDigitMult(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT c,
    NN_DIGIT *d, uint32_t digits)
{
    NN_DIGIT carry, t[2];
    uint32_t i;

    if (c == 0)
        return (0);
    carry = 0;
    for (i = 0; i < digits; i ++) {
        NN_DigitMult (t, c, d[i]);
        if ((a[i] = b[i] + carry) < carry)
            carry = 1;
        else
            carry = 0;
        if ((a[i] += t[0]) < t[0])
            carry ++;
        carry += t[1];
    }
    return (carry);
}
```

图 7. 中间计算结果函数 1

```
static NN_DIGIT NN_SubDigitMult(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT c,
    NN_DIGIT *d, uint32_t digits)
{
    NN_DIGIT borrow, t[2];
    uint32_t i;

    if (c == 0)
        return (0);
    borrow = 0;
    for (i = 0; i < digits; i ++) {
        NN_DigitMult (t, c, d[i]);
        if ((a[i] = b[i] - borrow) > (MAX_NN_DIGIT - borrow))
            borrow = 1;
        else
            borrow = 0;
        if ((a[i] -= t[0]) > (MAX_NN_DIGIT - t[0]))
            borrow++;
        borrow += t[1];
    }
    return (borrow);
}
```

图 8. 中间计算结果函数 2

3.4. 计算大数位数的函数

```
static uint32_t NN_DigitBits(NN_DIGIT a)
{
	uint32_t i;

	for (i = 0; i < NN_DIGIT_BITS; i ++, a >>= 1)
		if (a == 0)
			break;
	return (i);
}

uint32_t NN_Digits(NN_DIGIT *a, uint32_t digits)
{
	int32_t i;

	for (i = digits - 1; i >= 0; i --)
		if (a[i])
			break;
	return (i + 1);
}

uint32_t NN_Bits(NN_DIGIT *a, uint32_t digits)
{
	if ((digits = NN_Digits (a, digits)) == 0)
		return (0);
	return ((digits - 1) * NN_DIGIT_BITS + NN_DigitBits (a[digits - 1]));
}
```

图 9. 计算大数位数的函数

3.5. 大数转换函数

```
void NN_Encode(uint8_t *a, uint32_t len, NN_DIGIT *b, uint32_t digits)
{
	NN_DIGIT t;
	int32_t j;
	uint32_t i, u;

	for (i = 0, j = len - 1; i < digits && j >= 0; i ++) {
		t = b[i];
		for (u = 0; j >= 0 && u < NN_DIGIT_BITS; j --, u += 8)
			a[j] = (uint8_t)(t >> u);
	}
	for (; j >= 0; j --)
		a[j] = 0;
}

void NN_Decode(NN_DIGIT *a, uint32_t digits, uint8_t *b, uint32_t len)
{
	NN_DIGIT t;
	int32_t j;
	uint32_t i, u;

	for (i = 0, j = len - 1; i < digits && j >= 0; i++) {
		t = 0;
		for (u = 0; j >= 0 && u < NN_DIGIT_BITS; j--, u += 8)
			t |= ((NN_DIGIT)b[j]) << u;
		a[i] = t;
	}
	for (; i < digits; i ++)
		a[i] = 0;
}
```

图 10. 大数转换函数

3.6. 大数赋值函数

```
void NN_Assign(NN_DIGIT *a, NN_DIGIT *b, uint32_t digits)
{
	uint32_t i;

	for (i = 0; i < digits; i++)
		a[i] = b[i];
}

void NN_AssignZero(NN_DIGIT *a, uint32_t digits)
{
	uint32_t i;

	for (i = 0; i < digits; i++)
		a[i] = 0;
}

void NN_Assign2Exp(NN_DIGIT *a, uint32_t b, uint32_t digits)
{
	NN_AssignZero (a, digits);

	if (b >= digits * NN_DIGIT_BITS)
		return;
	a[b / NN_DIGIT_BITS] = (NN_DIGIT)1 << (b % NN_DIGIT_BITS);
}
```

图 11. 大数赋值函数

3.7. 大数比较函数

```
int NN_Cmp(NN_DIGIT *a, NN_DIGIT *b, uint32_t digits)
{
	int32_t i;

	for (i = digits - 1; i >= 0; i --) {
		if (a[i] > b[i])
			return 1;
		if (a[i] < b[i])
			return -1;
	}
	return 0;
}

int NN_Is_Zero(NN_DIGIT *a, uint32_t digits)
{
	uint32_t i;

	for (i = 0; i < digits; i++)
		if (a[i])
			return 0;
	return 1;
}
```

图 12. 大数比较函数

3.8. 大数移位函数

```
NN_DIGIT NN_LShift(NN_DIGIT *a, NN_DIGIT *b, uint32_t c, uint32_t digits)
{
	NN_DIGIT bi, carry;
	uint32_t i, t;

	if (c >= NN_DIGIT_BITS)
		return (0);
	t = NN_DIGIT_BITS - c;
	carry = 0;
	for (i = 0; i < digits; i++) {
		bi = b[i];
		a[i] = (bi << c) | carry;
		carry = c ? (bi >> t) : 0;
	}
	return (carry);
}

NN_DIGIT NN_RShift(NN_DIGIT *a, NN_DIGIT *b, uint32_t c, uint32_t digits)
{
	NN_DIGIT bi, carry;
	int32_t i;
	uint32_t t;

	if (c >= NN_DIGIT_BITS)
		return (0);
	t = NN_DIGIT_BITS - c;
	carry = 0;
	for (i = digits - 1; i >= 0; i--) {
		bi = b[i];
		a[i] = (bi >> c) | carry;
		carry = c ? (bi << t) : 0;
	}
	return (carry);
}
```

图 13. 大数移位函数

3.9. 大数加减函数

```
NN_DIGIT NN_Add(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits)
{
	NN_DIGIT ai, carry;
	uint32_t i;

	carry = 0;
	for (i = 0; i < digits; i ++) {
		if ((ai = b[i] + carry) < carry)
			ai = c[i];
		else if ((ai += c[i]) < c[i])
			carry = 1;
		else
			carry = 0;
		a[i] = ai;
	}
	return (carry);
}

NN_DIGIT NN_Sub(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits)
{
	NN_DIGIT ai, borrow;
	uint32_t i;

	borrow = 0;
	for (i = 0; i < digits; i ++) {
		if ((ai = b[i] - borrow) > (MAX_NN_DIGIT - borrow))
			ai = MAX_NN_DIGIT - c[i];
		else if ((ai -= c[i]) > (MAX_NN_DIGIT - c[i]))
			borrow = 1;
		else
			borrow = 0;
		a[i] = ai;
	}
	return (borrow);
}
```

图 14. 大数加减函数

3.10. 大数乘法函数

```
void NN_Mult(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits)
{
    NN_DIGIT t[2 * MAX_NN_DIGITS];
    uint32_t bDigits, cDigits, i;

    NN_AssignZero (t, 2 * digits);
    bDigits = NN_Digits (b, digits);
    cDigits = NN_Digits (c, digits);
    for (i = 0; i < bDigits; i ++)
        t[i + cDigits] += NN_AddDigitMult (&t[i], &t[i], b[i], c, cDigits);
    NN_Assign (a, t, 2 * digits);
//  memset (t, 0, sizeof (t));
}
```

图 15. 大数乘法函数

3.11. 大数除法函数

```
void NN_Div(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t cDigits,
      NN_DIGIT *d, uint32_t dDigits)
{
      NN_DIGIT ai, cc[2 * MAX_NN_DIGITS + 1], dd[MAX_NN_DIGITS], t;
      int32_t i;
      uint32_t ddDigits, shift;

      ddDigits = NN_Digits (d, dDigits);
      if (ddDigits == 0)
            return;
      shift = NN_DIGIT_BITS - NN_DigitBits (d[ddDigits-1]);
      NN_AssignZero (cc, ddDigits);
      cc[cDigits] = NN_LShift (cc, c, shift, cDigits);
      NN_LShift (dd, d, shift, ddDigits);
      t = dd[ddDigits - 1];
      NN_AssignZero (a, cDigits);
      for (i = cDigits - ddDigits; i >= 0; i--) {
            if (t == MAX_NN_DIGIT)
                  ai = cc[i+ddDigits];
            else
                  NN_DigitDiv (&ai, &cc[i + ddDigits - 1], t + 1);
            cc[i+ddDigits] -= NN_SubDigitMult (&cc[i], &cc[i], ai, dd, ddDigits);
            while (cc[i + ddDigits] || (NN_Cmp (&cc[i], dd, ddDigits) >= 0)) {
                  ai ++;
                  cc[i + ddDigits] -= NN_Sub (&cc[i], &cc[i], dd, ddDigits);
            }
            a[i] = ai;
      }
      NN_AssignZero (b, dDigits);
      NN_RShift (b, cc, shift, ddDigits);
//    memset (cc, 0, sizeof (cc));
//    memset (dd, 0, sizeof (dd));
}
```

图 16. 大数除法函数

3.12. 大数取模函数

```
void NN_Mod(NN_DIGIT *a, NN_DIGIT *b, uint32_t bDigits, NN_DIGIT *c, uint32_t cDigits)
{
      NN_DIGIT t[2 * MAX_NN_DIGITS];

      NN_Div (t, a, b, bDigits, c, cDigits);
//    memset (t, 0, sizeof (t));
}
```

图 17. 大数取模函数

3.13. 大数乘法取模函数

```
void NN_ModMult(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, NN_DIGIT *d, uint32_t digits)
{
      NN_DIGIT t[2 * MAX_NN_DIGITS];

      NN_Mult (t, b, c, digits);
      NN_Mod (a, t, 2 * digits, d, digits);
//    memset (t, 0, sizeof (t));
}
```

图 18. 大数乘法取模函数

3.14. 大数求倒数取模函数

```
void NN_ModInv(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits)
{
    NN_DIGIT q[MAX_NN_DIGITS], t1[MAX_NN_DIGITS], t3[MAX_NN_DIGITS],
        u1[MAX_NN_DIGITS], u3[MAX_NN_DIGITS], v1[MAX_NN_DIGITS],
        v3[MAX_NN_DIGITS], w[2 * MAX_NN_DIGITS];
    int32_t u1Sign;

    NN_ASSIGN_DIGIT (u1, 1, digits);
    NN_AssignZero (v1, digits);
    NN_Assign (u3, b, digits);
    NN_Assign (v3, c, digits);
    u1Sign = 1;
    while (! NN_Is_Zero (v3, digits)) {
        NN_Div (q, t3, u3, digits, v3, digits);
        NN_Mult (w, q, v1, digits);
        NN_Add (t1, u1, w, digits);
        NN_Assign (u1, v1, digits);
        NN_Assign (v1, t1, digits);
        NN_Assign (u3, v3, digits);
        NN_Assign (v3, t3, digits);
        u1Sign = -u1Sign;
    }
    if (u1Sign < 0)
        NN_Sub (a, c, u1, digits);
    else
        NN_Assign (a, u1, digits);
//  memset (q, 0, sizeof (q));
//  memset (t1, 0, sizeof (t1));
//  memset (t3, 0, sizeof (t3));
//  memset (u1, 0, sizeof (u1));
//  memset (u3, 0, sizeof (u3));
//  memset (v1, 0, sizeof (v1));
//  memset (v3, 0, sizeof (v3));
//  memset (w, 0, sizeof (w));
}
```

图 19. 大数求倒数取模函数

3.15. 大数其他函数

```
void NN_ModExp(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t cDigits,
     NN_DIGIT *d, uint32_t dDigits)
{
     NN_DIGIT bPower[3][MAX_NN_DIGITS], ci, t[MAX_NN_DIGITS];
     int32_t i;
     uint32_t ciBits, j, s;

     NN_Assign (bPower[0], b, dDigits);
     NN_ModMult (bPower[1], bPower[0], b, d, dDigits);
     NN_ModMult (bPower[2], bPower[1], b, d, dDigits);
     NN_ASSIGN_DIGIT (t, 1, dDigits);
     cDigits = NN_Digits (c, cDigits);
     for (i = cDigits - 1; i >= 0; i --) {
          ci = c[i];
          ciBits = NN_DIGIT_BITS;
          if (i == (int)(cDigits - 1)) {
               while (! DIGIT_2MSB (ci)) {
                    ci <<= 2;
                    ciBits -= 2;
               }
          }
          for (j = 0; j < ciBits; j += 2, ci <<= 2) {
               NN_ModMult (t, t, t, d, dDigits);
               NN_ModMult (t, t, t, d, dDigits);
               if ((s = DIGIT_2MSB (ci)) != 0)
                    NN_ModMult (t, t, bPower[s - 1], d, dDigits);
          }
     }
     NN_Assign (a, t, dDigits);
//   memset (bPower, 0, sizeof (bPower));
//   memset (t, 0, sizeof (t));
}
```

图 20. 大数其他函数（第 1 部分）

```
void NN_Gcd(NN_DIGIT *a, NN_DIGIT *b, NN_DIGIT *c, uint32_t digits)
{
    NN_DIGIT t[MAX_NN_DIGITS], u[MAX_NN_DIGITS], v[MAX_NN_DIGITS];

    NN_Assign (u, b, digits);
    NN_Assign (v, c, digits);
    while (! NN_Is_Zero (v, digits)) {
        NN_Mod (t, u, digits, v, digits);
        NN_Assign (u, v, digits);
        NN_Assign (v, t, digits);
    }
    NN_Assign (a, u, digits);
//  memset (t, 0, sizeof (t));
//  memset (u, 0, sizeof (u));
//  memset (v, 0, sizeof (v));
}
```

图 21. 大数其他函数（第 2 部分）

求指数 NN_ModExp 以及最大公倍数 NN_Gcd 的函数暂时没有用到。

4. HASH 函数

在签名算法 ECDSA 中要对数据进行 HASH 运算得到该数据的摘要，一般可以使用 SHA-1，在此处采用 Zigbee 中使用的 AES_MMO 算法。如果没有 AES 算法源码，又想要验证 ECC 算法，可以改写 aes_encrypt 函数为原样返回，即不进行任何加密。

```
#ifndef _AES_MMO_H
#define _AES_MMO_H

#include "bn.h"

#define AES_BLOCK_SIZE  16

/* key_len: 16, 24, or 32. */
void aes_encrypt (uint8_t *ct, const uint8_t *pt, const uint8_t *key, int key_len);

void aes_decrypt (uint8_t *pt, const uint8_t *ct, const uint8_t *key, int key_len);

void aes_mmo_hash(uint8_t hash[AES_BLOCK_SIZE], const uint8_t *buf, int len);

#endif // _AES_MMO_H
```

图 22. AES_MMO 算法头文件 (aes_mmo.h)

```
#include <string.h>
#include "aes_mmo.h"

static uint16_t crc16(uint16_t crc, const uint8_t *buf, int len)
{
	#define CRC_OK	0x470f
	#define POLY 0x8408
	uint8_t i, data;

	crc ^= 0xffff;
	while (len --) {
		data = *buf ++;
		for (i = 0; i < 8; i ++) {
			if ((crc & 0x0001) ^ (data & 0x0001))
				crc = (crc >> 1) ^ POLY;
			else
				crc >>= 1;
			data >>= 1;
		}
	}
	crc ^= 0xffff;
	return crc;
}

static void aes_mmo(uint8_t *hash, const uint8_t *buf)
{
	int i;
	uint8_t temp[AES_BLOCK_SIZE];

	aes_encrypt(temp, buf, hash, AES_BLOCK_SIZE);
	for (i = 0; i < AES_BLOCK_SIZE; i ++)
		hash[i] = temp[i] ^ buf[i];
}
```

图 23.　AES_MMO 算法（第 1 部分）

```
void aes_mmo_hash(uint8_t hash[AES_BLOCK_SIZE], const uint8_t *buf, int len)
{
	uint16_t crc, bit_len;
	uint8_t tmp[AES_BLOCK_SIZE];

	bit_len = (len + 2);
	crc = 0;

	memset(hash, 0, sizeof(hash));
	while (len >= AES_BLOCK_SIZE) {
		crc = crc16(crc, buf, len);
		aes_mmo(hash, buf);
		buf += AES_BLOCK_SIZE;
		len -= AES_BLOCK_SIZE;
	}
	memset(tmp, 0, AES_BLOCK_SIZE);
	memcpy(tmp, buf, len);
	crc = crc16(crc, buf, len);
	if (len <= AES_BLOCK_SIZE - 5) {
		tmp[len] = crc & 0xff;
		tmp[len + 1] = crc >> 8;
		tmp[len + 2] = 0x80;
	}
	else if (len <= AES_BLOCK_SIZE - 3) {
		tmp[len] = crc & 0xff;
		tmp[len + 1] = crc >> 8;
		tmp[len + 2] = 0x80;
		aes_mmo(hash, tmp);
		memset(tmp, 0, AES_BLOCK_SIZE);
	}
	else if (len == AES_BLOCK_SIZE - 2) {
		tmp[len] = crc & 0xff;
		tmp[len + 1] = crc >> 8;
		aes_mmo(hash, tmp);
		memset(tmp, 0, AES_BLOCK_SIZE);
		tmp[0] = 0x80;
	}
```

图 24. AES_MMO 算法（第 2 部分）

```
    else if (len == AES_BLOCK_SIZE - 1) {
        tmp[len] = crc & 0xff;
        aes_mmo(hash, tmp);
        memset(tmp, 0, AES_BLOCK_SIZE);
        tmp[0] = crc >> 8;
        tmp[1] = 0x80;
    }
    tmp[AES_BLOCK_SIZE - 2] = bit_len >> 5;
    tmp[AES_BLOCK_SIZE - 1] = bit_len << 3;
    aes_mmo(hash, tmp);
}
```

图 25. AES_MMO 算法（第 3 部分）

5. ECC 算法

5.1. ECC 算法头文件

```
#ifndef _ECC_H
#define _ECC_H

#include "bn.h"

#define DEGREE 163                          /* the degree of the field polynomial */
#define MARGIN 3                                                    /* don't touch this */
#define NUMWORDS        ((DEGREE + MARGIN + 31) / 32)

typedef uint32_t bitstr_t[NUMWORDS];
typedef bitstr_t elem_t;   /* this type will represent field elements */
typedef bitstr_t exp_t;

extern const elem_t ecc_poly;
extern const elem_t coeff_a;
extern const elem_t coeff_b;
extern const elem_t base_x;
extern const elem_t base_y;
extern const exp_t base_order;

/* This function must be implemented in another file */
void generate_random(char *buf, int len);
void bitstr_import(bitstr_t x, const uint8_t *s); // octet string to BN
void bitstr_export(uint8_t *s, const bitstr_t x); // BN to octet string
void ecc_generate_key(elem_t Px, elem_t Py, exp_t d);

// r and s are stored in sign using bitstr_export function
void ecc_sign(char *sign, const elem_t e, const elem_t pri_key);

// return 1 vhen the sign is ok, else return 0
int ecc_verify(const char *sign, const elem_t e, const elem_t Px, const elem_t Py);

#endif // _ECC_H
```

图 26. ECC 算法头文件 (ecc.h)

5.2. ECC 算法实现

```
#include <string.h>
#include "ecc.h"

#define SECT163_K1

const elem_t ecc_poly      = {0x000000c9, 0x00000000, 0x00000000, 0x00000000,
      0x00000000, 0x00000008};

#ifdef SECT163_K1
const elem_t coeff_a       = {0x00000001, 0x00000000, 0x00000000, 0x00000000,
      0x00000000, 0x00000000};
const elem_t coeff_b    = {0x00000001, 0x00000000, 0x00000000, 0x00000000,
      0x00000000, 0x00000000};
const elem_t base_x    = {0x5C94EEE8, 0xDE4E6D5E, 0xAA07D793, 0x7BBC11AC,
      0xFE13C053, 0x00000002};
const elem_t base_y = {0xCCDAA3D9, 0x0536D538, 0x321F2E80, 0x5D38FF58,
      0x89070FB0, 0x00000002};
const exp_t base_order    = {0x99F8A5EF, 0xA2E0CC0D, 0x00020108, 0x00000000,
      0x00000000, 0x00000004};
#endif

#ifdef SECT163_R1
const elem_t coeff_a       = {0xD2782AE2, 0xBD88E246, 0x54FF8428, 0xEFA84F95,
      0xB6882CAA, 0x00000007};
const elem_t coeff_b    = {0xF958AFD9, 0xCA91F73A, 0x946BDA29, 0xDCB40AAB,
      0x13612DCD, 0x00000007};
const elem_t base_x = {0x7876A654, 0x567F787A, 0x89566789, 0xAB438977,
      0x69979697, 0x00000003};
const elem_t base_y = {0xF41FF883, 0xE3C80988, 0x9D51FEFC, 0xEFAFB298,
      0x435EDB42, 0x00000000};
const exp_t base_order    = {0xA710279B, 0xB689C29C, 0xFFFF48AA, 0xFFFFFFFF,
      0xFFFFFFFF, 0x00000003};
#endif
```

图 27. ECC 算法 (ecc.c, 第 1 部分)

```
#ifdef SECT163_R2
const elem_t coeff_a    = {0x00000001, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000};
const elem_t coeff_b     = {0x4a3205fd, 0x512f7874, 0x1481eb10, 0xb8c953ca,
    0x0a601907, 0x00000002};
const elem_t base_x       = {0xe8343e36, 0xd4994637, 0xa0991168, 0x86a2d57e,
    0xf0eba162, 0x00000003};
const elem_t base_y    = {0x797324f1, 0xb11c5c0c, 0xa2cdd545, 0x71a0094f,
    0xd51fbc6c, 0x00000000};
const exp_t base_order    = {0xa4234c33, 0x77e70c12, 0x000292fe, 0x00000000,
    0x00000000, 0x00000004};
#endif

void field_add(elem_t c, const elem_t a, const elem_t b, const elem_t poly);

void field_sub(elem_t c, const elem_t a, const elem_t b, const elem_t poly);
```

图 28. ECC 算法 (ecc.c, 第 2 部分)

这一部分代码主要定义了 3 条二元域椭圆曲线的各种参数，分别是”sect163k1”,”sect163r1”和”sect163r2”。

可以通过修改 ecc.h 中的蓝色部分以及 bn.h 中的蓝色部分，再定义新的椭圆参数来支持其他曲线。比如“sect193r1”,“sect193r2”,甚至可以支持到“sect571k1”和“sect571r1”。当使用 sect571 时，需要加大 MAX_NN_DIGIT_BITS 的定义。

```
#define MAX_NN_DIGIT_BITS        1152
```

```
int bitstr_getbit(const bitstr_t a, int idx)
{
	return (a[idx / 32] >> (idx % 32)) & 1;
}

void bitstr_setbit(bitstr_t a, int idx)
{
	a[idx / 32] |= 1 << (idx % 32);
}

void bitstr_clrbit(bitstr_t a, int idx)
{
	a[idx / 32] &= ~(1 << (idx % 32));
}

int bitstr_is_clear(const bitstr_t x)
{
	int i;

	for (i = 0; i < NUMWORDS && ! *x; i ++)
		x ++;
	return i == NUMWORDS;
}

int bitstr_is_equal(const bitstr_t a, const bitstr_t b)
{
	return !memcmp(a, b, sizeof(bitstr_t));
}

void bitstr_clear(bitstr_t a)
{
	memset(a, 0, sizeof(bitstr_t));
}

void bitstr_copy(bitstr_t a, const bitstr_t b)
{
	memcpy(a, b, sizeof(bitstr_t));
}
```

图 29. ECC 算法 (ecc.c, 第 3 部分)

```
void bitstr_swap(bitstr_t a, bitstr_t b)
{
	bitstr_t h;

	bitstr_copy(h, a);
	bitstr_copy(a, b);
	bitstr_copy(b, h);
}

int bitstr_sizeinbits(const bitstr_t x)
{
	int i;
	uint32_t mask;

	for (x += NUMWORDS, i = 32 * NUMWORDS; i > 0 && ! * --x; i -= 32)
		;
	if (i) {
		for(mask = ((uint32_t) 1) << 31; ! (*x & mask); mask >>= 1, i --)
			;
	}
	return i;
}

void bitstr_lshift(bitstr_t a, const bitstr_t b, int count)
{
	int i, offs = 4 * (count / 32);

	memmove((uint8_t*)a + offs, b, sizeof(bitstr_t) - offs);
	memset(a, 0, offs);
	if (count %= 32) {
		for (i = NUMWORDS - 1; i > 0; i --)
			a[i] = (a[i] << count) | (a[i - 1] >> (32 - count));
		a[0] <<= count;
	}
}
```

图 30. ECC 算法 (ecc.c, 第 4 部分)

```
void field_set1(elem_t a)
{
	memset(a + 1, 0, sizeof(elem_t) - 4);
	a[0] = 1;
}

int field_is1(const elem_t a)
{
	int i;

	if (*a ++ != 1)
		return 0;
	for (i = 1; i < NUMWORDS && ! *a ++; i ++)
		;
	return i == NUMWORDS;
}

int field_cmp(const elem_t a, const elem_t b)
{
	int i;

	for (i = NUMWORDS - 1; i >= 0; i --) {
		if (a[i] < b[i])
			return -1;
		else if (a[i] > b[i])
			return 1;
	}
	return 0;
}
```

图 31. ECC 算法 (ecc.c, 第 5 部分)

这一部分代码功能相对简单，目的明确，很容易理解。当然有一些函数根本就没被调用。

```
void field_mod(elem_t c, const elem_t a, const elem_t poly)
{
	int i, flag;

	bitstr_copy(c, a);
	while (field_cmp(c, poly) > 0) {
		flag = 0;
		for (i = 0; i < NUMWORDS; i ++) {
			if (c[i] < poly[i] + flag) {
				c[i] = c[i] - (poly[i] + flag);
				flag = 1;
			}
			else {
				c[i] = c[i] - (poly[i] + flag);
				flag = 0;
			}
		}
	}
}

void field_add(elem_t c, const elem_t a, const elem_t b, const elem_t poly)
{
	int i, flag;
	elem_t d;

	flag = 0;
	for (i = 0; i < NUMWORDS; i ++) {
		d[i] = a[i] + b[i] + flag;
		if (d[i] < a[i] || d[i] < b[i])
			flag = 1;
		else
			flag = 0;
	}
	if (flag)
		field_sub(c, d, poly, poly);
	else
		bitstr_copy(c, d);
}
```

图 32. ECC 算法 (ecc.c, 第 6 部分)

```
void field_sub(elem_t c, const elem_t a, const elem_t b, const elem_t poly)
{
	int i, flag;
	elem_t d;

	flag = 0;
	for (i = 0; i < NUMWORDS; i ++) {
		d[i] = a[i] - b[i] - flag;
		if (d[i] > a[i] || d[i] > b[i])
			flag = 1;
		else
			flag = 0;
	}
	if (flag)
		field_add(c, d, poly, poly);
	else
		bitstr_copy(c, d);
}

void field_mul(elem_t c, const elem_t a, const elem_t b, const elem_t poly)
{
	NN_ModMult(c, (NN_DIGIT *)a, (NN_DIGIT *)b, (NN_DIGIT *)poly, NUMWORDS);
}

void field_inv(elem_t b, const elem_t a, const elem_t poly)
{
	NN_ModInv(b, (NN_DIGIT *)a, (NN_DIGIT *)poly, NUMWORDS);
}
```

图 33. ECC 算法 (ecc.c, 第 7 部分)

以上代码可以都可以通过调用 BN 中对应的函数来实现。针对椭圆曲线的点乘法来说，系数的计算, 使用 `field_mod`, `field_add` 和 `field_sub` 比直接调用 BN 的函数会快一些。

```
void field_add1_F2m(elem_t a)
{
	a[0] ^= 1;
}

void field_add_F2m(elem_t c, const elem_t a, const elem_t b)
{
	int i;
	for (i = 0; i < NUMWORDS; i ++)
		c[i] = a[i] ^ b[i];
}

void field_mul_F2m(elem_t c, const elem_t a, const elem_t b, const elem_t poly)
{
	elem_t x, y;
	int i, j, degree;
	bitstr_copy(x, b);
	if (bitstr_getbit(a, 0))
		bitstr_copy(y, b);
	else
		bitstr_clear(y);
	degree = bitstr_sizeinbits(poly) - 1;
	for (i = 1; i < degree; i ++) {
		for (j = NUMWORDS - 1; j > 0; j --)
			x[j] = (x[j] << 1) | (x[j - 1] >> 31);
		x[0] <<= 1;
		if (bitstr_getbit(x, degree))
			field_add_F2m(x, x, poly);
		if (bitstr_getbit(a, i))
			field_add_F2m(y, y, x);
	}
	bitstr_copy(c, y);
}

void field_sqr_F2m(elem_t c, const elem_t a, const elem_t poly)
{
	field_mul_F2m(c, a, a, poly);
}
```

图 34. ECC 算法 (ecc.c, 第 8 部分)

```
void field_inv_F2m(elem_t b, const elem_t a, const elem_t poly)
{
	elem_t c, d, u, v, x;
	int j;

	field_set1(x);
	bitstr_clear(c);
	bitstr_copy(u, a);
	bitstr_copy(v, poly);
	while (!field_is1(u)) {
		j = bitstr_sizeinbits(u) - bitstr_sizeinbits(v);
		if (j < 0) {
			bitstr_swap(u, v);
			bitstr_swap(x, c);
			j = -j;
		}
		bitstr_lshift(d, v, j);
		field_add_F2m(u, u, d);
		bitstr_lshift(d, c, j);
		field_add_F2m(x, x, d);
	}
	bitstr_copy(b, x);
}
```

图 35. ECC 算法 (ecc.c, 第 9 部分)

对于 field_sqr_F2m 有更快速的算法，而不必借助 field_mul_F2m.

函数 field_mul_F2m 请参考“参考资料 4”中的 Algorithm 1.

函数 field_inv_F2m 请参考“参考资料 4”中的 Algorithm 8.

```
int point_is_zero(const elem_t x, const elem_t y)
{
	return bitstr_is_clear(x) && bitstr_is_clear(y);
}

void point_set_zero(elem_t x, elem_t y)
{
	bitstr_clear(x);
	bitstr_clear(y);
}

void point_copy(elem_t x1, elem_t y1, const elem_t x2, const elem_t y2)
{
	bitstr_copy(x1, x2);
	bitstr_copy(y1, y2);
}
```

图 36. ECC 算法 (ecc.c, 第 10 部分)

这是一些常用的赋值和判断函数。

```
/* check if y^2 + x*y = x^3 + a*x^2 + b holds */
int is_point_on_curve(const elem_t x, const elem_t y, const elem_t poly)
{
	elem_t a, b, c;

	if (point_is_zero(x, y))
		return 1;

	// compute right
	field_sqr_F2m(a, x, poly);
	field_add_F2m(b, x, coeff_a);
	field_mul_F2m(c, a, b, poly);
	field_add_F2m(a, c, coeff_b);

	// comute left
	field_add_F2m(b, x, y);
	field_mul_F2m(b, b, y, poly);

	return bitstr_is_equal(a, b);
}
```

图 37. ECC 算法 (ecc.c, 第 11 部分)

该函数判断某点是否在椭圆曲线上。参考注释可以很容易理解。

```
/* double the point (x, y) */
void point_double(elem_t x, elem_t y, const elem_t poly)
{
	elem_t s;

	if (! bitstr_is_clear(x)) {
		field_inv_F2m(s, x, poly);    // s = 1 / x
		field_mul_F2m(s, s, y, poly); // s = y / x
		field_add_F2m(s, s, x);    // s = x + y / x

		field_sqr_F2m(y, x, poly); // y = x * x

		field_sqr_F2m(x, s, poly); // x = s * s
		field_add_F2m(x, x, s);    // x = s * s + s
		field_add_F2m(x, x, coeff_a); // x = s * s + s + a

		field_add1_F2m(s); // s = s + 1
		field_mul_F2m(s, s, x, poly); // s = (s + 1) * x2
		field_add_F2m(y, y, s);    // y = x * x + (s + 1) * x2
	}
	else {
		bitstr_clear(y);
	}
}
```

图 38 ECC 算法 (ecc.c, 第 12 部分)

```
/* add two points together (x1, y1) := (x1, y1) + (x2, y2) */
void point_add(elem_t x1, elem_t y1, const elem_t x2, const elem_t y2,
    const elem_t poly)
{
    elem_t a, b, c, s;

    if (! point_is_zero(x2, y2)) {
        if (point_is_zero(x1, y1)) {
            point_copy(x1, y1, x2, y2);
        }
        else {
            if (bitstr_is_equal(x1, x2)) {
                if (bitstr_is_equal(y1, y2)) {
                    point_double(x1, y1, poly);
                }
                else {
                    point_set_zero(x1, y1);
                }
            }
            else {
                field_add_F2m(a, y1, y2);     // a = y1 + y2
                field_add_F2m(b, x1, x2);     // b = x1 + x2
                field_inv_F2m(c, b, poly);          // c = 1 /(x1 + x2)
                field_mul_F2m(s, c, a, poly); // s = (y1 + y2) / (x1 + x2)
                bitstr_copy(a, x1);         // a = x1, save x1
                field_add_F2m(c, b, coeff_a);// c = x1 + x2 + a
                field_add_F2m(c, s, c);         // c = s + x1 + x2 + a
                field_sqr_F2m(x1, s, poly);         // x1 = s * s
                field_add_F2m(x1, x1, c);     // x1 = s * s + s + x1 + x2 + a

                field_add_F2m(y1, x1, y1);    // y1 = x3 + y1
                field_add_F2m(b, a, x1);// b = x1 + x3
                field_mul_F2m(c, s, b, poly);       // c = s * (x1 + x3)
                field_add_F2m(y1, c, y1);     // y1 = s * (x1 + x3) + x3 + y1
            }
        }
    }
}
```

图 39 ECC 算法 (ecc.c, 第 13 部分)

```
/* point multiplication via double-and-add algorithm */
void point_mul(elem_t x, elem_t y, const exp_t exp, const elem_t poly)
{
	int i;
	elem_t X, Y;

	point_set_zero(X, Y);
	for (i = bitstr_sizeinbits(exp) - 1; i >= 0; i --) {
		point_double(X, Y, poly);
		if (bitstr_getbit(exp, i))
			point_add(X, Y, x, y, poly);
	}
	point_copy(x, y, X, Y);
}
```

图 40 ECC 算法 (ecc.c, 第 14 部分)

该函数就是 ECC 算法中经常用到的点乘操作。在使用 ECC 算法时，大部分运算都是通过 point_mul 函数以及 point_add 函数完成。

请参考“参考资料3”中的“EC on Binary field F2M”

```
void bitstr_import(bitstr_t x, const uint8_t *s)
{
	int i;

	for (i = NUMWORDS - 1; i >= 0; i --) {
		x[i] = (s[0] << 24) | (s[1] << 16) | (s[2] << 8) | s[3];
		s += 4;
	}
}

void bitstr_export(uint8_t *s, const bitstr_t x)
{
	uint32_t r;
	int i;

	for (i = NUMWORDS - 1; i >= 0; i --) {
		r = x[i];
		*s ++ = (r >> 24);
		*s ++ = (r >> 16);
		*s ++ = (r >> 8);
		*s ++ = r;
	}
}
```

图 41 ECC 算法 (ecc.c, 第 15 部分)

```
static void get_random_exponent(exp_t exp, const elem_t poly)
{
	int r;
	unsigned char buf[4 * NUMWORDS];

	do {
		generate_random(buf, 4 * NUMWORDS);
		bitstr_import(exp, buf);
		for (r = bitstr_sizeinbits(poly) - 1; r < NUMWORDS * 32; r ++)
			bitstr_clrbit(exp, r);
	} while (bitstr_is_clear(exp));
}

/* check that a given elem_t-pair is a valid point on the curve != 'o' */
int ecc_check_public_key(const elem_t x, const elem_t y, const elem_t poly)
{
	int degree;

	degree = bitstr_sizeinbits(poly) - 1;
	return (bitstr_sizeinbits(x) > degree)
		|| (bitstr_sizeinbits(y) > degree)
		|| point_is_zero(x, y)
		|| !is_point_on_curve(x, y, poly) ? 0 : 1;
}

/* same thing, but check also that (Px,Py) generates a group of order n */
int ecc_full_check_public_key(const elem_t x, const elem_t y, const elem_t poly)
{
	elem_t X, Y;

	if (ecc_check_public_key(x, y, poly) == 0)
		return 0;
	bitstr_copy(X, x);
	bitstr_copy(Y, y);
	point_mul(X, Y, base_order, poly);
	return point_is_zero(X, Y);
}
```

图 42ECC 算法 (ecc.c, 第 16 部分)

注意，需要在其他程序文件中提供 generate_random 函数，该函数产生 len 字节长的随机字节。随机数产生函数相当重要，如果攻击者能猜出随机数的话，则加密就没有任何意义了。

```
void ecc_generate_key(elem_t Px, elem_t Py, exp_t d)
{
	while (1) {
		get_random_exponent(d, base_order);
		point_copy(Px, Py, base_x, base_y);
		point_mul(Px, Py, d, ecc_poly);
		if (!bitstr_is_clear(Px) && !bitstr_is_clear(Py)
			&& ecc_full_check_public_key(Px, Py, ecc_poly))
			break;
	}
}
```

图 43 ECC 算法 (ecc.c, 第 17 部分)

```
void ecc_sign(char *msg, const elem_t e, const elem_t d)
{
	elem_t r, s, k;

	while (1) {
		ecc_generate_key(r, s, k);
		field_inv(k, k, base_order);
		field_mod(r, r, base_order);
		field_mul(s, d, r, base_order);
		field_add(s, e, s, base_order);
		field_mul(s, k, s, base_order);
		if (!bitstr_is_clear(s))
			break;
	}
	bitstr_export(msg, r);
	bitstr_export(msg + sizeof(uint32_t) * NUMWORDS, s);
}
```

图 44 ECC 算法 (ecc.c, 第 18 部分)

```
int ecc_verify(const char *msg, const elem_t e, const elem_t Px, const elem_t Py)
{
	elem_t r, s, u1, u2, x1, y1;

	bitstr_import(r, msg);
	bitstr_import(s, msg + sizeof(uint32_t) * NUMWORDS);
	if (field_cmp(r, base_order) >= 0 || field_cmp(s, base_order) >= 0)
		return 0;
	field_inv(s, s, base_order);
	field_mul(u1, e, s, base_order);
	field_mul(u2, r, s, base_order);

	point_copy(x1, y1, Px, Py);
	point_mul(x1, y1, u2, ecc_poly);
	point_copy(s, u2, base_x, base_y);
	point_mul(s, u2, u1, ecc_poly);
	point_add(x1, y1, s, u2, ecc_poly);

	field_mod(x1, x1, base_order);
	return !field_cmp(x1, r);
}
```

图 45 ECC 算法 (ecc.c, 第 19 部分)

请参考“参考资料3”中的“Signature Generation” 和 “Signature Verification”，请注意此处的运算不要使用后缀为_F2m 的函数，而应该使用 BN 中的函数。

```
void ecc_kdf(int a, char *secret, ...)
{
	// TODO
}

void ecc_dh(char *secret, const elem_t d, const elem_t Qx, const elem_t Qy)
{
	elem_t x1, y1;

	point_copy(x1, y1, Qx, Qy);
	point_mul(x1, y1, d, ecc_poly);
	field_mod(x1, x1, base_order);
	bitstr_export(secret, x1);
	ecc_kdf(0, secret);
}
```

图 46ECC 算法 (ecc.c, 第 20 部分, 完毕)

由于存在很多的 KDF 算法，ecc_dh 也有很多算法，因此只做了一个最简单的。谢谢！
存在的问题是没有进行 DER 的编解码，不支持公钥的压缩，性能也很差。

6. 附录

以下是测试代码，可以进行初步验证。想要进行进一步的测试，可以考虑与 openssl 的函数相互进行签名与验签。

```
#include <stdio.h>
#include <string.h>
#include "ecc.h"
#include "aes_mmo.h"

#define PRINTF                  printf
void PRINTB(char *prompt, char *buf, int len)
{
	int i;
	printf("%s", prompt);
	for (i = 0; i < len; i ++)
		printf("%02x", (unsigned char)buf[i]);
	printf("\n");
}
void PRINT_NN(const char *prompt, const elem_t a)
{
	char buf[1024], *ptr;
	const char *str = "0123456789ABCDEF";
	int i, j, k;
	uint32_t r;
	for (i = NUMWORDS - 1; i > 0; i --) {
		if (a[i])	break;
	}
	ptr = buf;
	do {
		r = a[i --];
		k = 28;
		for (j = 0; j < 8; j ++) {
			*ptr ++ = str[(r >> k) & 0x0f];
			k -= 4;
		}
	} while (i >= 0);
	*ptr = 0;
	printf("%s%s\n", prompt, buf);
}
```

图 47 ECC 算法验证 (ecc_test.c, 第 1 部分)

```
void generate_random(char *buf, int len)
{     // too simple, only for test. You should use SHA-1 or another to generate randoms
      static int xrandm = 100000000;
      static int xrandm1 = 10000;
      static int xrandb1 = 517236213;
      static int a = 1234567891;
      int p1, p0, q1, q0, val, p, q;

      while (len --) {
            p = a;
            q = xrandb1;
            p1 = p / xrandm1; p0 = p % xrandm1;
            q1 = q / xrandm1; q0 = q % xrandm1;
            val = (((p0 * q1 + p1 * q0) % xrandm1) * xrandm1 + p0 * q0) % xrandm;
            a = (val + 1) % xrandm;
            *buf ++ = a & 0xff;
      }
}

void hash(elem_t e, const char *text, int len)
{
      uint8_t hash[4 * NUMWORDS];

      memset(hash, 0, sizeof(hash));
      aes_mmo_hash(hash + sizeof(hash) - AES_BLOCK_SIZE, text, len);
      bitstr_import(e, hash);
}
```

图 48 ECC 算法验证（ecc_test.c，第 2 部分）

Hash 函数主要目的是对 text 进行摘要运行，以保证发送的数据没有被改动，最好采用 SHA-1 计算。在这儿为了简单，使用了在 Zigbee 中常用的 AES_MMO 算法。

```
int main(void)
{
	elem_t Px, Py, d, e;
	const char *text = "This is an test\n";
	int i, ret;
	char msg[8 * NUMWORDS];

	hash(e, text, strlen(text));
	PRINT_NN("e:\t", e);
	ecc_generate_key(Px, Py, d);
	PRINT_NN("d:\t", d);
	PRINT_NN("Px:\t", Px);
	PRINT_NN("Py:\t", Py);
	for (i = 0; i < 3; i ++) {
		PRINTF("\nloop: %d\n", i);
		ecc_sign(msg, e, d);
		PRINTB("r:\t", msg, 4 * NUMWORDS);
		PRINTB("s:\t", msg + 4 * NUMWORDS, 4 * NUMWORDS);
		ret = ecc_verify(msg, e, Px, Py);
		PRINTF("verify %s\n", ret ? "ok" : "fail");
	}
	return 0;
}
```

图 49 ECC 算法验证 (ecc_test.c, 第 3 部分)

7. 参考资料

【1】 SEC 1: Elliptic Curve Cryptography

【2】 SEC 2: Recommended Elliptic Curve Domain Parameters

【3】 Elliptic Curve Cryptography - An Implementation Tutorial

【4】 Darrel Hankerson, Julio Lopez Hernandez, and Alfred Menezes, Software Implementation of Elliptic Curve Cryptography over Binary Fields

【5】 Darrel Hankerson，Alfred Menezes，Scott Vanstone, Guide to Elliptic Curve Cryptography