Apéndice A

Código de la aritmética de puntos

A.1. Definición de clase

```
#include <NTL/GF2XFactoring.h>
#include <NTL/GF2E.h>
#include <NTL/ZZ.h>

// Elliptic curve parameters

GF2E ec_A();

GF2E ec_B();

void ec_init(const GF2E& A, const GF2E& B);

void ec_init(GF2X Phi, long a, long b);


// Elliptic curve point
class ec_point {
public:

      unsigned infinity;
      GF2E x,y;

      ec_point() { infinity=0; x=y=0L;}

      ~ec_point() {}
```

```
      ec_point& operator=(const ec_point& P)
            { infinity=P.infinity; x=P.x; y=P.y; return *this; }

      static void init(GF2X Phi, long a, long b)
            { ec_init(Phi,a,b); }

      static void init(GF2E A, GF2E B)
            { ec_init(A,B); }

};


// Comparison

inline unsigned IsOnEC(const ec_point& P)
{
      return (P.infinity||
                 (P.y*(P.y+P.x))==(P.x*P.x*(P.x+ec_A())+ec_B()));
}

inline unsigned IsInfinity(const ec_point& P)
{
      return (P.infinity);
};

inline unsigned operator==(const ec_point& P, const ec_point& Q)
{
      return ((P.infinity==Q.infinity)||(P.x==Q.x && P.y==Q.y));
};

//Arithmetic operations for elliptic curve points

void negate(ec_point& P, const ec_point& Q);

void add(ec_point& P, const ec_point& Q, const ec_point& R);

void duplicate(ec_point& P, const ec_point& Q);

void multiply(ec_point& Q, const ZZ& k, const ec_point& P);

inline ec_point operator+(const ec_point& P, const ec_point& Q)
{
      ec_point T; add(T,P,Q); return T;
};
```

```
inline ec_point dup(const ec_point& P)
{
      ec_point T; duplicate(T,P); return T;
};

inline ec_point operator-(const ec_point& P)
{
      ec_point T; negate(T,P); return T;
};

inline ec_point operator*(const ZZ& k, const ec_point& P)
{
      ec_point T; multiply(T,k,P); return T;
};
```

A.2. Implementación de clase

```
#include <ecc.h>

//Arithmetic operations for elliptic curve points

GF2E _ec_AB(unsigned T, const GF2E& A, const GF2E& B)
{
      static GF2E __ec_A,__ec_B;

      switch (T){
            case 0: __ec_A=A;
                  __ec_B=B;
                  return __ec_A;
                  break;
            case 1: return __ec_A;
            case 2:
            default: return __ec_B;
      }
};

GF2E ec_A()
{
      return _ec_AB(1,GF2E::zero(),GF2E::zero());
};

GF2E ec_B()
{
```

```
      return _ec_AB(2,GF2E::zero(),GF2E::zero());
};

void ec_init(const GF2E& A,const GF2E& B)
{
      _ec_AB(0,A,B);
};

void ec_init(GF2X Phi, long a, long b)
{
      GF2E::init(Phi);
      GF2E A,B;
      conv(A,a); conv(B,b);
      ec_init(A,B);
};


void negate(ec_point& P, const ec_point& Q) // P= -Q
{
      P.infinity=Q.infinity;
      P.x=Q.x;
      P.y=Q.x+Q.y;
};

void add(ec_point& P, const ec_point& Q, const ec_point& R) // P=Q+R
{
      GF2E lambda, mu;

      if(Q.infinity) P=R;
      else if (R.infinity) P=Q;
            else {
                  if(Q.x==R.x){
                        if(Q.y==(R.x+R.y)){ //then Q==-R
                              P.infinity=1;
                              P.x=P.y=0L;
                              return;
                        };
                        lambda=(Q.x*Q.x+Q.y)/Q.x; // Q==R
                        mu=Q.x*Q.x;
                  }
                  else {
                        lambda=(R.y+Q.y)/(R.x+Q.x);
                        mu=(Q.y*R.x+R.y*Q.x)/(R.x+Q.x);
                  };
                  P.infinity=0;
```

```
                  P.x=lambda*lambda+lambda+ec_A()+Q.x+R.x;
                  P.y=(lambda+1L)*P.x+mu;
            }

};

void duplicate(ec_point& P, const ec_point& Q) // P=2Q
{
      GF2E lambda, mu;

      if(Q.infinity) { P.infinity=1; P.x=P.y=0L; }
      else {
            mu=Q.x*Q.x;
            lambda=(mu+Q.y)/Q.x;
            P.infinity=0;
            P.x=lambda*lambda+lambda+ec_A();
            P.y=(lambda+1L)*P.x+mu;
      }
}

void multiply(ec_point& Q, const ZZ& k, const ec_point& P) // Q=kP
{
      long j;

      Q.infinity=1;
      Q.x=Q.y=0L;

      for(j=NumBits(k)-1;j>=0;j--){
            duplicate(Q,Q);
            if(bit(k,j))add(Q,Q,P);
      };
}
```

A.3. Programa de prueba

```
#include <iostream.h>
#include <ECC/ecc.h>


GF2X Phi; // Irreducible polynomial

void print_point(const ec_point& P)
{
```

```
    if(IsInfinity(P))
         cout << "Infinito\n";
    else
         cout << P.x << "\n" << P.y << "\n";

    if (!IsOnEC(P))
         cout << "El punto NO satisface la ecuacin de la curva\n";
}

main()
{

      long i;

      cout << "Polinomio irreducible:\nPhi(t)=";

      do {
            cin >> i;
            SetCoeff(Phi,i);
            if(i) cout << "t^" << i << "+"; else cout << "+1\n";
      } while(i);

      if(!IterIrredTest(Phi))
            cout << "ADVERTENCIA: Polinomio reducible!!\n";

      GF2X::HexOutput=1; // Change to hex I/O format

      GF2E::init(Phi); // Initialize underlying field

      GF2E a,b; // Coefficients of elliptic curve: y^2+xy=x^3+ax^2+b

      cin >> a;
      cin >> b;

      ec_point::init(a,b);

      ec_point G; // Generator point


      G.infinity=0;
      cin >> G.x;
      cin >> G.y;

      cout << "\nEl punto generador G es:\n";
      print_point(G);
```

```
ZZ n;
cin >> n;

cout << "\nEl orden de G es:\nn="<< n;

ec_point P,Q,R;

multiply(Q,n,G);

cout << "\nEl resultado de nG es:\n";
print_point(Q);

//duplicate(P,G); // P=2G
P=dup(G);
cout << "\nEl resultado de P=2G es:\n";
print_point(P);

//add(Q,P,G);  // Q=3G
Q=P+G;
cout << "\nEl resultado de Q=P+G es:\n";
print_point(Q);

//negate(P,G); // P=-G
P=-G;
cout << "\nEl resultado de P=-G es:\n";
print_point(P);

//add(R,Q,P); //R=2G
R=Q+P;
cout << "\nEl resultado de R=Q+P es:\n";
print_point(R);

//add(P,R,Q); //P=5G
P=R+Q;
cout << "\nEl resultado de P=R+Q es:\n";
print_point(P);

ZZ k1,k2;

// Initialize random number generator
SetSeed(to_ZZ((long)time(NULL)));

RandomBits(k1,deg(Phi));
cout << "\nEntero aleatorio:\nk1=" << k1 << "\n";
```

```
    RandomBits(k2,deg(Phi));
    cout << "\nEntero aleatorio:\nk2=" << k2 << "\n";

    multiply(R,k1,P); // R=k1P
    multiply(Q,k2,R);
    cout << "\nEl resultado de k2*(k1*P) es:\n";
    print_point(Q);

    multiply(R,k2,P);
    multiply(Q,k1,R);
    cout << "\nEl resultado de k1*(k2*P) es:\n";
    print_point(Q);

}
```

A.4. Salida de una ejecución

```
Polinomio irreducible:
Phi(t)=t^163+t^7+t^6+t^3++1

El punto generador G es:
0x8eee49c5e5d6e4ed397d70aaca11cbb7350c31ef2
0x9d3aadcc835d635008e2f12385ff83d50bf070982

El orden de G es:
n=5846006549323611672814741753598448348329118574063
El resultado de nG es:
Infinito

El resultado de P=2G es:
0xbe14c5a8d828b77a24b00bfcaa003ef8372ac5bc
0xb6afefe515466696a3af5d3dca09f58ba9e97c922

El resultado de Q=P+G es:
0x02596f02bd330028f4208282f3e8fa2a9ccfcfca2
0x4efdceff9014085e41fd71c4b7cdab519f74c9275

El resultado de P=-G es:
0x8eee49c5e5d6e4ed397d70aaca11cbb7350c31ef2
0x13d4e409668b87bd319f81894fee48623efc4177

El resultado de R=Q+P es:
0xbe14c5a8d828b77a24b00bfcaa003ef8372ac5bc
```

```
0xb6afefe515466696a3af5d3dca09f58ba9e97c922

El resultado de P=R+Q es:
0xc6b6b701e40a4888828e49206ffde3249e22f9973
0x8dae174d03b4cc4ac77d7e65ec215439f7919c286

Entero aleatorio:
k1=3728977874060251105598286897733878381535118234258

Entero aleatorio:
k2=2211754861024375336142134678017244734940075421768

El resultado de k2*(k1*P) es:
0xd9db11c518586ff98171617aa4530bf09af9ef304
0xde61eb1ea4a53fd536e4925fc62cd71094f163124

El resultado de k1*(k2*P) es:
0xd9db11c518586ff98171617aa4530bf09af9ef304
0xde61eb1ea4a53fd536e4925fc62cd71094f163124
```

Apéndice B

Código del esquema de cifrado ECES

B.1. Generación de claves

```
//*********************************************************
// keypairgen.c -- Key pair generation for the
//                ECDSA digital signature algorithm
//
// Author: Juan Manuel Garcia (jmgarcia@sekureit.com)
//
// Compile as: g++ -o keypairgen keypairgen.c -lntl -lecc
//
// Intended with the only purpose of testing the ECC library
//
// Last modification date: February 12, 2003
//
//**********************************************************
#include <iostream.h>
#include <time.h>
#include <ECC/ecc.h>

GF2X Phi; // Irreducible polynomial
ZZ d; // Private key
ZZ n; // Generator point order


main()
{
```

```
    long i;


    do { // Read the irreducible polynomial coefficients
          cin >> i;
          SetCoeff(Phi,i);
    } while(i);


    GF2X::HexOutput=1;  // Change to hex I/O format

    GF2E::init(Phi); // Initialize Galois field

    GF2E a,b; // Coefficients of elliptic curve: y^2+xy=x^3+ax^2+b

    cin >> a; // Read coeffs
    cin >> b;

    ec_point::init(a,b); // Initialize elliptic curve

    ec_point G; // Generator point
    ec_point Q; // Public key


    G.infinity=0;
    cin >> G.x;
    cin >> G.y;

    cin >> n; // Read order of G

    // Initialize random number generator
    SetSeed(to_ZZ((long)time(NULL)));

    // Select a random d, 0< d < n
    RandomBnd(d,n);

    // Compute Q=dG
    multiply(Q,d,G);

    // Printout keys
    cout << "Private key:\n" << d << "\n";
    cout << "Public key:\n";
    cout << Q.x <<"\n";
    cout << Q.y <<"\n";
}
```

B.2. Cifrado

```
//**********************************************************
// crypt.c -- Elliptic Curve Encryption Scheme (ECES)
//                      encryption program
//
// Author: Juan Manuel Garcia (jmgarcia@sekureit.com)
//
// Compile as: g++ -o crypt crypt.c -lntl -lecc
//
// Intended with the only purpose of testing the ECC library
//
// Last modification date: February 12, 2003
//
//**********************************************************

#include <iostream.h>
#include <ECC/ecc.h>

GF2X Phi; // Irreducible polynomial
ZZ n; // Generator point order
ZZ data; // Data to be encrypted

void conv(ZZ& a, const GF2E& x)
{
      unsigned char *p; long numb;

      numb=NumBytes(x._GF2E__rep);
      p=(unsigned char *)malloc(numb);
      BytesFromGF2X(p,x._GF2E__rep,numb);
      ZZFromBytes(a,p,numb);
      free(p);
};

main()
{

      long i;


      do { // Read the irreducible polynomial coefficients
            cin >> i;
            SetCoeff(Phi,i);
      } while(i);
```

```
GF2X::HexOutput=1; // Change to hex I/O format

GF2E::init(Phi); //Initialize Galois field

GF2E a,b; // Coefficients of elliptic curve: y^2+xy=x^3+ax^2+b

cin >> a; // Read coeffs
cin >> b;

ec_point::init(a,b); // Initialize elliptic curve

ec_point G; // Generator point

G.infinity=0;
cin >> G.x;
cin >> G.y;

cin >> n; // Read order of G

ec_point Q; // Public key

cin >> Q.x; // Read public key
cin >> Q.y;

//cin >> data; // Read data to be signed

ZZ k; GF2E m,c; ec_point P1,P2;


//conv(m,data); // Convert data to a field's element
cin >> m;

// Initialize random number generator
SetSeed(to_ZZ((long)time(NULL)));

do{
      // Select a random k, 0< k < n
      RandomBnd(k,n);

      // Compute P1=kG
      multiply(P1,k,G);
      // Compute P2=kQ
      multiply(P2,k,Q);
} while(IsZero(P2.x));
```

```
    c=m*P2.x;


    // Printout ciphertext
    cout << P1.x <<"\n";
    cout << P1.y <<"\n";
    cout << c <<"\n";


}
```

B.3. Descifrado

```
//************************************************************
// decrypt.c -- Elliptic Curve Encryption Scheme (ECES)
//                  decryption algorithm
//
// Author: Juan Manuel Garcia (jmgarcia@sekureit.com)
//
// Compile as: g++ -o decrypt decrypt.c -lntl -lecc
//
// Intended with the only purpose of testing the ECC library
//
// Last modification date: January 30, 2003
//
//************************************************************

#include <iostream.h>
#include <ECC/ecc.h>

GF2X Phi;   // Irreducible polynomial
ZZ n;       // Generator point order

void conv(ZZ& a, const GF2E& x)
{
    unsigned char *p; long numb;

    numb=NumBytes(x._GF2E__rep);
    p=(unsigned char *)malloc(numb);
    BytesFromGF2X(p,x._GF2E__rep,numb);
    ZZFromBytes(a,p,numb);
    free(p);
};
```

```
main()
{
      long i;

      // Read the irreducible polynomial coefficients
      do {
            cin >> i;
            SetCoeff(Phi,i);
      } while(i);

      GF2X::HexOutput=1; // Change to hex I/O format
      GF2E::init(Phi); //Initialize Galois field

      GF2E a,b; // Coefficients of elliptic curve: y^2+xy=x^3+ax^2+b

      cin >> a; // Read coeffs
      cin >> b;

      ec_point::init(a,b); // Initialize elliptic curve

      ec_point G; // Generator point

      G.infinity=0;
      cin >> G.x;
      cin >> G.y;

      cin >> n; // Read order of G

      ZZ d; // Private key
      cin >> d; // Read private key

      GF2E m,c; ec_point P1,P2;

      cin >> P1.x;
      cin >> P1.y;

      cin >> c; // Read ciphered text

      P2=d*P1;
      m=c*inv(P2.x);

      // Printout plain text
      cout << m <<"\n";
}
```

Apéndice C

Código de firma digital ECDSA

C.1. Generación de claves

```
//**********************************************************
// keypairgen.c -- Key pair generation for the
//                ECDSA digital signature algorithm
//
// Author: Juan Manuel Garcia (jmgarcia@sekureit.com)
//
// Compile as: g++ -o keypairgen keypairgen.c -lntl -lecc
//
// Intended with the only purpose of testing the ECC library
//
// Last modification date: February 12, 2003
//
//**********************************************************
#include <iostream.h>
#include <time.h>
#include <ECC/ecc.h>

GF2X Phi; // Irreducible polynomial
ZZ d; // Private key
ZZ n; // Generator point order


main()
{
```

```
    long i;

    do { // Read the irreducible polynomial coefficients
          cin >> i;
          SetCoeff(Phi,i);
    } while(i);


    GF2X::HexOutput=1;  // Change to hex I/O format

    GF2E::init(Phi); // Initialize Galois field

    GF2E a,b; // Coefficients of elliptic curve: y^2+xy=x^3+ax^2+b

    cin >> a; // Read coeffs
    cin >> b;

    ec_point::init(a,b); // Initialize elliptic curve

    ec_point G; // Generator point
    ec_point Q; // Public key


    G.infinity=0;
    cin >> G.x;
    cin >> G.y;

    cin >> n; // Read order of G

    // Initialize random number generator
    SetSeed(to_ZZ((long)time(NULL)));

    // Select a random d, 0< d < n
    RandomBnd(d,n);

    // Compute Q=dG
    multiply(Q,d,G);

    // Printout keys
    cout << "Private key:\n" << d << "\n";
    cout << "Public key:\n";
    cout << Q.x <<"\n";
    cout << Q.y <<"\n";
}
```

C.2. Generación de firma

```
//*******************************************************
// siggen.c -- Signature generation for the ECDSA
//                digital signature algorithm
//
// Author: Juan Manuel Garcia (jmgarcia@sekureit.com)
//
// Compile as: g++ -o siggen siggen.c -lntl -lecc
//
// Intended with the only purpose of testing the ECC library
//
// Last modification date: February 12, 2003
//
//********************************************************

#include <iostream.h>
#include <ECC/ecc.h>

GF2X Phi; // Irreducible polynomial
ZZ d; // Private key
ZZ n; // Generator point order
ZZ data; // Data to be signed

void conv(ZZ& a, const GF2E& x)
{
      unsigned char *p; long numb;

      numb=NumBytes(x._GF2E__rep);
      p=(unsigned char *)malloc(numb);
      BytesFromGF2X(p,x._GF2E__rep,numb);
      ZZFromBytes(a,p,numb);
      free(p);
};

main()
{
      long i;

      // Read the irreducible polynomial coefficients
      do {
            cin >> i;
            SetCoeff(Phi,i);
      } while(i);
```

```
    GF2X::HexOutput=1; // Change to hex I/O format
    GF2E::init(Phi); //Initialize Galois field

    GF2E a,b; // Coefficients of elliptic curve: y^2+xy=x^3+ax^2+b

    cin >> a; // Read coeffs
    cin >> b;

    ec_point::init(a,b); // Initialize elliptic curve

    ec_point G; // Generator point

    G.infinity=0;
    cin >> G.x;
    cin >> G.y;

    cin >> n; // Read order of G
    cin >> d; // Read private key

    cin >> data; // Read data to be signed

    ZZ k,kinv,r,s; ec_point Q;

    // Initialize random number generator
    SetSeed(to_ZZ((long)time(NULL)));

    do {
          do{
                // Select a random k, 0< k < n
                RandomBnd(k,n);

                // Compute Q=kG
                multiply(Q,k,G);
                //r=Q.x;
                conv(r,Q.x);
          } while(IsZero(r));
          InvMod(kinv,k,n);
          s=kinv*(data+(d*r))%n;
    } while(IsZero(s));

    // Printout signature
    cout << r <<"\n";
    cout << s <<"\n";
}
```

C.3. Verificación de firmas

```
//*******************************************************
// verify.c -- Signature verification for the ECDSA
//               digital signature algorithm
//
// Author: Juan Manuel Garcia (jmgarcia@sekureit.com)
//
// Compile as: g++ -o verify verify.c -lntl -lecc
//
// Intended with the only purpose of testing the ECC library
//
// Last modification date: January 21, 2003
//
//*******************************************************
#include <iostream.h>
#include <ECC/ecc.h>

GF2X Phi; // Irreducible polynomial
ZZ n; // Generator point order
ZZ data; // Data to be verified
ZZ r,s; // Signature

void conv(ZZ& a, const GF2E& x)
{
      unsigned char *p; long numb;

      numb=NumBytes(x._GF2E__rep);
      p=(unsigned char *)malloc(numb);
      BytesFromGF2X(p,x._GF2E__rep,numb);
      ZZFromBytes(a,p,numb);
      free(p);
};

main()
{
      long i;

      // Read the irreducible polynomial coefficients
      do {
            cin >> i;
            SetCoeff(Phi,i);
      } while(i);
```

```
    GF2E::init(Phi);

    GF2E a,b;

    cin >> a;
    cin >> b;

    ec_point::init(a,b); // Initialize elliptic curve

    ec_point G; // Generator point
    ec_point Q; // Public key

    G.infinity=0;
    cin >> G.x;
    cin >> G.y;

    cin >> n; // Read order of G

    Q.infinity=0; // Read public key
    cin >> Q.x;
    cin >> Q.y;

    cin >> data; // Read data
    cin >> r; // Read signature
    cin >> s;

    ZZ w,u1,u2,v; ec_point X;

    InvMod(w,s,n);
    u1=data*w%n;
    u2=r*w%n;
    X=u1*G+u2*Q;

    conv(v,X.x);
    v%=n;

    if(v==r) cout << "Signature OK\n";
    else cout << "Wrong signature\n";
}
```

Apéndice D

Código del algoritmo paralelo

D.1. Algoritmo binario de orden p

```
//************************************************************
//  ec_par.c - Implementation of the binary p-order algorithm
//
//  Author: Juan Manuel Garcia (jmgarcia@sekureit.com)
//
//  Compile as mpiCC -o ec_par ec_par.c -lntl -lecc
//
//  Last modification date: March 10, 2003
//************************************************************
#include <NTL/GF2E.h>
#include <NTL/ZZ.h>
#include <ECC/ecc.h>
#include <time.h>
#include <mpi++.h>

#define MAX_PROC 10

GF2X Phi; // Irreducible polynomial
int rank, size;

void print_point(const ec_point& P)
{
//    cout <<"El punto:"<<endl;
      if(IsInfinity(P))
          cout <<"Infinito"<<endl;
      else
          cout <<P.x<<endl<<P.y<<endl;
      if (!IsOnEC(P))
```

```
          cout <<"El punto NO pertenece a la curva"<<endl;
}

#define send_uint(ui,pr) MPI::COMM_WORLD.Send(&ui,1,MPI::INT,pr,1);
#define receive_uint(ui,pr) MPI::COMM_WORLD.Recv(&ui,1,MPI::INT,pr,1);

void send_ZZ(ZZ n, long pr)
{
      unsigned long num_bytes; char *s;

//    cout<<"Proceso "<< pr << " enviando "<< n << endl;
      num_bytes=NumBytes(n);
      MPI::COMM_WORLD.Send(&num_bytes,1,MPI::LONG,pr,1);
      s=(char*)malloc(sizeof(char)*num_bytes);
      BytesFromZZ((unsigned char*)s,n,num_bytes);
      MPI::COMM_WORLD.Send(s,num_bytes,MPI::CHAR,pr,1);
      free(s);
}

ZZ receive_ZZ(long pr)
{
      ZZ n; char *s; unsigned long num_bytes;

//    cout <<"Proceso "<< pr << "recibiendo ";
      MPI::COMM_WORLD.Recv(&num_bytes,1,MPI::LONG,pr,1);
      s=(char*)malloc(sizeof(char)*num_bytes);
      MPI::COMM_WORLD.Recv(s,num_bytes,MPI::CHAR,pr,1);
      n=ZZFromBytes((const unsigned char*)s,num_bytes);
      free(s);
//    cout << n << endl;
      return n;
}

GF2X receive_GF2X(long pr)
{
      GF2X pol; char *s; unsigned long num_bytes;

      MPI::COMM_WORLD.Recv(&num_bytes,1,MPI::LONG,pr,1);
      s=(char*)malloc(sizeof(char)*num_bytes);
      MPI::COMM_WORLD.Recv(s,num_bytes,MPI::CHAR,pr,1);
      pol=GF2XFromBytes((const unsigned char*)s,num_bytes);
      free(s);
      return pol;
}
```

```
void send_GF2X(GF2X p, long pr)
{
      unsigned long num_bytes; char *s;

      num_bytes=NumBytes(p);
      MPI::COMM_WORLD.Send(&num_bytes,1,MPI::LONG,pr,1);
      s=(char*)malloc(sizeof(char)*num_bytes);
      BytesFromGF2X((unsigned char*)s,p,num_bytes);
      MPI::COMM_WORLD.Send(s,num_bytes,MPI::CHAR,pr,1);
      free(s);
}

void send_GF2E(GF2E p, long pr)
{
//    cout << "Proceso "<< pr <<" enviando "<< p << endl;
      send_GF2X(p._GF2E__rep,pr);
}

GF2E receive_GF2E(long pr)
{
      GF2X paux; GF2E pol;

//    cout << "Proceso "<< pr <<" recibiendo ";
      paux=receive_GF2X(pr);
      pol=to_GF2E(paux);
//    cout << pol << endl;
      return (pol);
}

void send_ec_point(ec_point P,long pr)
{
      send_uint(P.infinity,pr);
      send_GF2E(P.x,pr);
      send_GF2E(P.y,pr);
}

ec_point receive_ec_point(long pr)
{
      ec_point P; unsigned ui;

      receive_uint(ui,pr);
      P.infinity=ui;
      P.x=receive_GF2E(pr);
      P.y=receive_GF2E(pr);
      return P;
```

```
}

//****************************************
// Slave process
//****************************************

void slave()
{
      ZZ k;   // Subexponent
      unsigned level; // Tree level on the associative fan

      k=receive_ZZ(0L);

      Phi=receive_GF2X(0L);
//    cout <<Phi<<endl;
      if(!IterIrredTest(Phi))
      {
          cout <<"Error en la transmision, terminando..."<<endl;
          MPI::Finalize();
          exit(1);
      }
      GF2E::init(Phi);
      cout <<"OK"<<endl;

      GF2E a,b;

      GF2X::HexOutput=1;

      //Receiving elliptic curve parameters
      a=receive_GF2E(0L);
      b=receive_GF2E(0L);

      ec_point::init(a,b);

      ec_point Q,R,T;

      //Receiving Q from master process
      Q=receive_ec_point(0L);

      multiply(R,k,Q);

      // Associative fan
      for(level=1;level!=(size-1);level*=2) {
            if(rank%(2*level)){
                  send_ec_point(R,rank+level);
```

```
                return;
            }
            else {
                T=receive_ec_point(rank-level);
                R=T+R;
            }
        }

        //Sending R to master process
        send_ec_point(R,0L);
}

//**********************************************
// Master process
//**********************************************

void master()
{

        ZZ n_spread[MAX_PROC]; //array for divided scalar
        long i; //loop counter
        unsigned long len_ZZ;

        //set up irreductible polynomial coefficients
        do
        {
            cin >>i;
            SetCoeff(Phi,i);
        }while(i);

        if(!IterIrredTest(Phi))
        {
            cout <<"Polinomio Reducible"<<endl;
            MPI::Finalize();
            exit(1);
        }

        GF2X::HexOutput=1;
        GF2E::init(Phi);

//      cout<<Phi<<endl;
        GF2E a,b; //a,b constants for the eliptic curve

        cin >>a;
        cin >>b;
```

```
//    cout <<a<<endl;
//    cout <<b<<endl;

      ec_point::init(a,b);
      ec_point G,P,aux; //Points for the eliptic curvearithmetics

      G.infinity=0;
      cin >>G.x;
      cin >>G.y;

//    cout <<G.x<<endl;
//    cout <<G.y<<endl;

      ZZ n,k;
      cin >>n;
      RandomBits(k,deg(Phi));

      P.infinity=1;
      P.x=0x0;
      P.y=0x0;

      for(i=0;i<size;i++)n_spread[i]=0;

      len_ZZ=NumBits(k);
      for(i=0;i<len_ZZ;i++)
          if(bit(k,i))
            SetBit(n_spread[i%(size-1)],i);

      // Bits dispersion

      for(i=1;i<size;i++)
      {
          cout <<"Proceso Maestro: Enviando datos a "<<i<<endl;
          //sending k[i]
          send_ZZ(n_spread[i-1],i);

          //sending Phi
          send_GF2X(Phi,i);

          //sending elliptic curve coefficients
          send_GF2E(a,i);
          send_GF2E(b,i);

          //sending G
```

```
            send_ec_point(G,i);
        }


        // Associative fan

        P=receive_ec_point(size-1);
}

int main(int argc, char* argv[])
{

    MPI::Init(argc,argv);

    size=MPI::COMM_WORLD.Get_size();
    rank=MPI::COMM_WORLD.Get_rank();

    if(rank) slave();
       else master();

    MPI::Finalize();

    return 0;
}
```

Apéndice E

Especificación de curvas elípticas

Para cada curva se indica el grado de extensión m del campo subyacente $GF(2^m)$, el polinomio irreductible $\phi(x)$, los coeficientes a y b de la ecuación de la curva elíptica $y^2 + xy = x^3 + ax^2 + b$, los valores n y h tales que $\#E_{a,b}(GF(2^m)) = h \cdot n$ y las coordenadas G_x y G_y del punto $G = (G_x, G_y)$ de orden n.

E.1. Curva B-163

$$
\begin{aligned}
m &= 163 \\
\phi(x) &= x^{163} + x^7 + x^6 + x^3 + 1 \\
a &= \text{0x1} \\
b &= \text{0xd f502 3a44 787f 2150 1be1 841a c359 c8b7 0910 6a02} \\
G_x &= \text{0x6 3e34 38e7 3649 94d8 6119 90ae 75d2 a682 61ab e0f3} \\
G_y &= \text{0x1 f423 797c 0c5c 11b5 45dd c2af 4900 a17c 6cbf 15d0} \\
h &= 2 \\
n &= 5846006549323611672814742442876390689256843201587
\end{aligned}
$$

E.2. Curva B-233

$$
\begin{aligned}
m &= 233 \\
\phi(x) &= x^{233} + x^{74} + 1 \\
a &= \mathrm{0x1} \\
b &= \mathrm{0xda0\ 9f8d\ 7f51\ 1ef1\ 824e\ c9e0\ 2b33\ 3b31\ 285b\ b329\ 0c8f\ 7c23} \\
&\quad \mathrm{3c6e\ de74\ 6660} \\
G_x &= \mathrm{0xb85\ 5df1\ 737b\ e8f8\ f63b\ 8f19\ 3cb5\ 6fef\ 557b\ b1f9\ 312b\ b313} \\
&\quad \mathrm{8cab\ cfd9\ caf0} \\
G_y &= \mathrm{0x250\ 18f1\ 0e7f\ 6176\ 3ac7\ a768\ ffeb\ 0a8f\ beb8\ 2585\ e876\ 0533} \\
&\quad \mathrm{0914\ a80a\ 6001} \\
h &= 2 \\
n &= 69017463467905637874347558622770255558398127373450135553 \\
&\quad 79383634485463
\end{aligned}
$$

E.3. Curva B-283

$$
\begin{aligned}
m &= 283 \\
\phi(x) &= x^{283} + x^{12} + x^{7} + x^{5} + 1 \\
a &= \mathrm{0x1} \\
b &= \mathrm{0x5f2\ a97b\ 313e\ 3626\ fa58\ 4185\ a2af\ 9035\ 467d\ f79a\ cf30\ 30a9} \\
&\quad \mathrm{1a8f\ a4a5\ ad69\ 58b8\ ca08\ 6b72} \\
G_x &= \mathrm{0x350\ 21b6\ 8dce\ bdc8\ f891\ e2e0\ 8c9c\ ae75\ 58b5\ 2dee\ 2cef\ d0b0} \\
&\quad \mathrm{7c8f\ 4391\ e09d\ d7bd\ 8529\ 39f5} \\
G_y &= \mathrm{0x4f2\ 118e\ b54f\ d0f3\ 18c9\ 7762\ 80bd\ de05\ 3207\ ff61\ 54b2\ 0d02} \\
&\quad \mathrm{b4d6\ ef89\ bc14\ 142e\ f458\ 6763} \\
h &= 2 \\
n &= 77706755689029162836778476272940756265696259243769048 8 \\
&\quad 91091965267700442777873786928 71
\end{aligned}
$$

E.4. Curva B-409

$$
\begin{aligned}
m &= 409 \\
\phi(x) &= x^{409} + x^{87} + 1 \\
a &= \text{0x1} \\
b &= \text{0xf54 531b 713e a05f 4aa5 5a75 dc6f 2282 72b7 91a9 a8c7 2ca6} \\
&\quad \text{d99a f167 476d d3f1 fe22 46df 7b67 4b7b 357a 9b4c 5bef 9ee8} \\
&\quad \text{c2c5 a120} \\
G_x &= \text{0x7a6 997b b45e 4970 6bae a306 5150 811a 868a 552c d307 95e4} \\
&\quad \text{3b5e ff10 bd4d 1771 fa4e dc14 4062 6574 606c 0b69 43bd d880} \\
&\quad \text{d068 4d51} \\
G_y &= \text{0x607 c372 0ab4 63c1 863b 1812 d04f 4b4f df1f 4158 3f80 d884} \\
&\quad \text{5f4a a851 0d89 1db7 a5c9 b636 7a60 1de4 2387 afbb 23f5 eb6b} \\
&\quad \text{afc1 b160} \\
h &= 2 \\
n &= 661055968790248598951915308032771039828404682964281219 2 \\
&\quad 846487983041577748273748052081437237621791109659798672 8 \\
&\quad 8366567526771
\end{aligned}
$$

E.5. Curva B-571

$$
\begin{aligned}
m &= 571 \\
\phi(x) &= x^{571} + x^{10} + x^{5} + x^{2} + 1 \\
a &= \text{0x1} \\
b &= \text{0xa72 7559 2f7f feff 7c0a cab9 37ed 4e02 5aa2 1ff8 7a58 1dfa} \\
&\quad \text{492e 66a6 576d a7eb 2339 5afe 8dbb aff4 8da8 1a9a 4ec8 ab6d} \\
&\quad \text{c1ff ec8b cff7 9a6c 5f26 d3f7 b711 792e d592 f122 2e7e 04f2} \\
G_x &= \text{0x91d 2cee 8c96 77e1 e729 d058 c4b3 afba 4931 f416 8300 6ea9} \\
&\quad \text{941b f76b 53a1 17dd c392 d0c4 f059 35ed bdba 2b7b d8cf 04f5} \\
&\quad \text{aa08 af55 92d1 d39a 0577 dc3d 04d0 c61c 6926 58b4 3d10 0303}
\end{aligned}
$$

$$
\begin{aligned}
G_y &= \text{0xb51 ca8b 1fa7 284a 1c3d d32e 6151 f2e6 1b91 c584 0f2d 1353} \\
&\quad \text{b8a2 bb16 4f8f a192 675a 80ba b34e 3244 86a8 e129 3358 f089} \\
&\quad \text{1acb bc90 06a7 2c6c 87d9 6d37 beff fccd 6b93 6ad2 4372 fb73} \\
h &= 2 \\
n &= 38645375230172583446953518909319873442989273297064349 98 \\
&\quad 65723525145151914228956042453614399938941577308313388 1 \\
&\quad 12192694448624687246281681307023452828830333241139319 1 \\
&\quad 105285703
\end{aligned}
$$

E.6. Curva K-163

$$
\begin{aligned}
m &= 163 \\
\phi(x) &= x^{163} + x^7 + x^6 + x^3 + 1 \\
a &= \text{0x1} \\
b &= \text{0x1} \\
G_x &= \text{0x8 eee4 9c5e 5d6e 4ed3 97d7 0aac a11c bb73 50c3 1ef2} \\
G_y &= \text{0x9 d3aa dcc8 35d6 3500 8e2f 1238 5ff8 3d50 bf07 0982} \\
h &= 2 \\
n &= 5846006549323611672814741753598448348329118574063
\end{aligned}
$$

E.7. Curva K-233

$$
\begin{aligned}
m &= 233 \\
\phi &= x^{233} + x^{74} + 1 \\
a &= \text{0x0} \\
b &= \text{0x1} \\
G_x &= \text{0x621 6daf ee6d 9c4a 05fb 62c9 14a3 6594 14ff 22f9 21fa 137e} \\
&\quad \text{7a35 8ab2 3271} \\
G_y &= \text{0x3a6 eaf6 5011 c0e6 5b9b ea81 fb9d c8a7 24c7 6a55 5f07 f7b9}
\end{aligned}
$$

$$
\begin{aligned}
& && \text{18ec ed73 5bd1} \\
h &=& & 4 \\
n &=& & 3450873173395281893717377931138512760570940988862252126 \\
& && 328087024741343
\end{aligned}
$$

E.8. Curva K-283

$$
\begin{aligned}
m &= 283 \\
\phi(x) &= x^{283} + x^{12} + x^7 + x^5 + 1 \\
a &= \text{0x0} \\
b &= \text{0x1} \\
G_x &= \text{0x638 2948 542c a2c0 b319 6786 1a76 51c3 2f56 2dc3 55e8 81f2} \\
& \quad \text{618b 3a1f 3884 4ac8 7f31 2305} \\
G_y &= \text{0x952 2dd7 7161 143e 4632 6954 e896 4818 e0c5 4e78 ef62 45e7} \\
& \quad \text{0d59 f09d 813e 9c1f 083a dcc1} \\
h &= 4 \\
n &= 3885337784451458141838923813647037813284811733793061324 \\
& \quad 295874997529815829704422603873
\end{aligned}
$$

E.9. Curva K-409

$$
\begin{aligned}
m &= 409 \\
\phi(x) &= x^{409} + x^{87} + 1 \\
a &= \text{0x0} \\
b &= \text{0x1} \\
G_x &= \text{0x647 3209 efc0 4553 b1be 222e e26a aaa5 be98 1064 c2cc 76f9} \\
& \quad \text{f8bf cca7 2c48 c703 e789 0dfe 0124 817f 0981 ba3d a1c9 4f85} \\
& \quad \text{6f50 f060} \\
G_y &= \text{0xb68 20e8 d84c e368 5a72 ac9a a512 55c9 e24c 6f5a d3e0 1ae9}
\end{aligned}
$$

$$
\begin{aligned}
& \text{e561 5236 e724 ae81 9f28 7064 3c99 240f bcad 1abc a24e 4c7b} \\
& \text{0509 63e1} \\
h &= 4 \\
n &= 3305279843951242994759576540163855199142023414821406096 \\
& \quad 42324395022880711289249191050673258457777458014096366590 \\
& \quad 617731358671
\end{aligned}
$$

E.10. Curva K-571

$$
\begin{aligned}
m &= 571 \\
\phi(x) &= x^{571} + x^{10} + x^5 + x^2 + 1 \\
a &= \text{0x0} \\
b &= \text{0x1} \\
G_x &= \text{0x279 8c10 a382 5492 e7c8 8acd 4717 4b88 9bf6 7749 493a b1db} \\
& \quad \text{bc80 bec4 bd40 3ad7 46e5 02b3 9485 9073 44ac 1481 0840 8420} \\
& \quad \text{64d5 d210 0792 ac9c a4ef 3018 f136 9812 8cbf 3299 58a7 be62} \\
G_y &= \text{0x3a7 c1fe 341c 4dc1 06f4 8919 58c0 3402 3b1f a7ab 7a10 b026} \\
& \quad \text{bcde a277 f9bb bebf 47ae a44c a0c9 794d 9c2a 8d60 0cfe 16cf} \\
& \quad \text{f45a 703f 9cec 85dd 4135 9acb 3eda ea4f 473f bf4f 708c d943} \\
h &= 4 \\
n &= 19322687615086291723476759454659936721494636648532174993 \\
& \quad 28617625725759571144780212268133978522706711834706712800 \\
& \quad 82535146127367497406661731192968242161709250355573368 \\
& \quad 5276673
\end{aligned}
$$