borZoi Manual

Dragongate Technologies Ltd.

September 21, 2003

Contents

Chapter 1

Introduction

borZoi is an Elliptic Curve Cryptography Library which implements the following algorithms using elliptic curves defined over finite fields of characteristic 2 ($GF2^m$).

- Elliptic Curve Diffie-Hellman Key Agreement Scheme as specified in ANSI X9.63 and IEEE P1363.
- ECDSA (Elliptic Curve Digital Signature Algorithm) as specified in ANSI X9.62, FIPS 186-2 and IEEE P1363.
- ECIES (Elliptic Curve Integrated Encryption Scheme) as specified in ANSI X9.63 and the IEEE P1363a Draft.

The AES symmetric encryption scheme (NIST AES draft), SHA-1 hash algorithm (FIPS 180-1), KDF2 key derivation function (P1363a), HMAC message authentication code (P1363) and DER encoding functions are also included.

Chapter 2

Preliminaries

2.1 Header Files

Two header files must be included to use borZoi:

- `borzoi.h` provides the borZoi routines
- `nist_curves.h` provides the standard elliptic curves recommended by NIST in FIPS-186-2

2.2 Type Definitions

Two useful type definitions are provided by the borzoi.h header file:

- `typedef unsigned char OCTET;`

  This represents an unsigned 8 bit type. It is called octet rather than byte in order to be consistent with the various standards.

- `typedef std::vector<OCTET> OCTETSTR;`

  This is a vector of octets and represents an octet string as described in various standards.

2.3 Environmental Variables

- `USE_MPI`

  This must be defined in the preprocessor.

2.4 Including borZoi in Your Project

- Microsoft Visual C++

  Add the "borZoi.dsp" project file to your project's workspace and add the header file directory to your project's include files path.

- Gnu Developer Tools (gcc, libtool, make)
  - Installation

    ```
    ./configure
    make
    make install
    ```

  - Usage

    ```
    c++ -DSTDC_HEADERS=1 -I/usr/local/include -g -O2 -DUSE_MPI -c myprog.cpp
    c++ -g -O2 -DUSE_MPI -o myprog myprog.o -lborzoi
    ```

Chapter 3

Low Level Classes

3.1 Large Integer and Finite Field Classes

3.1.1 BigInt

This is a multi-precision large integer class. All of the usual integer operations are provided (see `borzoi_math.h` for more details).

3.1.2 F2X

This is a multi-precision polynomial class that represents polynomials of the form: $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$, where the coefficients a_i are modulo 2. All of the usual polynomial operations including addition, subtraction, multiplication and division are supported (see `borzoi_math.h` for more details).

3.1.3 F2M

This is a multi-precision binary finite field class ($GF2^m$) that represents finite field elements in a polynomial basis of the form: $b_{m-1}x^{m-1}+b_{m-2}x^{m-2}+\cdots + b_0$, where are all operations are modulo an irreducible polynomial $a_m x^m + a_{m-1}x^{m-1} + \cdots + a_0$ (see `borzoi_math.h` for more details).

3.2 Point and Elliptic Curve Classes

3.2.1 Point

This is an elliptic curve point class consisting of two finite field elements x and y. See `borzoi_util.h` for more details.

3.2.2 Curve

This is an elliptic curve class ($E : y^2+xy = x^3+ax^2+b$). See `borzoi_util.h` for more details.

3.3 Conversion Functions

- `OCTETSTR BS2OSP (F2M b);`

  Convert binary string b (represented by a finite field element) to octet string form using the algorithm in the IEEE P1363 standard.

- `OCTETSTR FE2OSP (F2M x);`

  Convert finite field element x to octet string form using the algorithm in the IEEE P1363 standard.

- `BigInt OS2IP (OCTETSTR o);`

  Convert octet string o to large integer form using the algorithm in the IEEE P1363 standard.

- `OCTETSTR I2OSP (BigInt i);`

  Convert large integer i to octet string form using the algorithm in the IEEE P1363 standard.

- `BigInt FE2IP (F2M j);`

  Convert finite field element j to large integer form using the algorithm in the IEEE P1363 standard.

- `F2M I2FEP (BigInt z);`

  Convert large integer z to finite field element form.

Chapter 4

EC Domain Parameters

The elliptic curve domain parameters specify the elliptic curve used. These are described in more detail in section 7.1.2 of the IEEE P1363 standard. The parameters consist:

4.1 The finite field ($GF2^m$)

borZoi uses a binary field of the form 2^m over a polynomial basis. This is specified by `m`, an unsigned long, `basis`, an int, which can be 1 (Gaussian Basis: not supported in borZoi), 2 (Trinomial Basis: $x^m + x^k + 1$) or 3 (Pentanomial Basis: $x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$), `trinomial_k`, an unsigned long, representing the k power of the trinomial polynomial, `pentanomial_k3`, an unsigned long, representing the k_3 power of the pentanomial polynomial, `pentanomial_k2`, an unsigned long, representing the k_2 power of the pentanomial polynomial and `pentanomial_k1`, an unsigned long, representing the k_1 power of the pentanomial polynomial.

4.2 The elliptic curve ($E : y^2 + xy = x^3 + ax^2 + b$)

The elliptic curve is specified by `a`, a finite field element (`F2M`), `b`, a finite field element (`F2M`), `r`, a large positive prime integer (`BigInt`) which divides the number of points on the curve, `G`, a point on the curve (`Point`, see `borzoi_util.h`) which is the generator of a subgroup (of the points on the curve) of order r and `k`, a positive prime integer (`BigInt`) called the cofactor which is equal to the number of points on the curve divided by r.

4.3 Constructors

- `EC_Domain_Parameters ();`
- `EC_Domain_Parameters (m, basis, trinomial_k, c, r, G, k);`

Bit Size	Curve Name
163	NIST_B_163
233	NIST_B_233
283	NIST_B_283
409	NIST_B_409
571	NIST_B_571

Table 4.1: NIST Recommended Curves

- `EC_Domain_Parameters (m, basis, pentanomial_k3, pentanomial_k2, pentanomial_k1,`

  Note - `c` is of type Curve.

4.4 Methods

- `bool valid ();`

  This checks if the EC domain parameters are valid using a subset (steps 6.4 to 7) of the method in section A.16.8 of the IEEE P1363 standard:

  - Check that $b \neq 0$ in $GF2^m$.
  - Check that $G \neq O$.
  - Check that G_x and G_y are elements of $GF2^m$.
  - Check that $y^2 + xy = x^3 + ax^2 + b$ in $GF2^m$.
  - Check that $rG = O$.
  - Check that the curve is not subject to the MOV reduction attack.

- `EC_Domain_Parameters& operator= (const EC_Domain_Parameters& dp);`

  Assignment

- `std::ostream& put (std::ostream&s) const;`

  Output

4.5 NIST Recommended Curves

These are listed in table 4.1 taken from the standard curves recommended by NIST in FIPS-186-2 and are defined in `nist_curves.h`. To use one of these curves, you must first initialize the irreducible polynomial and then set the EC domain parameters:

Curve `NIST_B_163` Example

```
use_NIST_B_163 (); // initialize the irreducible polynomial
EC_Domain_Parameters dp = NIST_B_163; // set the domain parameters
```

Curve `NIST_B_233` Example

```
use_NIST_B_233 (); // initialize the irreducible polynomial
EC_Domain_Parameters dp = NIST_B_233; // set the domain parameters
```

Curve `NIST_B_283` Example

```
use_NIST_B_283 (); // initialize the irreducible polynomial
EC_Domain_Parameters dp = NIST_B_283; // set the domain parameters
```

Curve `NIST_B_409` Example

```
use_NIST_B_409 (); // initialize the irreducible polynomial
EC_Domain_Parameters dp = NIST_B_409; // set the domain parameters
```

Curve `NIST_B_571` Example

```
use_NIST_B_571 (); // initialize the irreducible polynomial
EC_Domain_Parameters dp = NIST_B_571; // set the domain parameters
```

4.6 User Defined Curves

If you want to use a different curve to those defined in `nist_curves.h`, you should refer to the `nist_curves.h` file for details of how to initialize the irreducible polynomial and construct the EC domain parameters. One method is to define the `use_MyCurve()` function and the `MyCurve EC_Domain_Parameters` object for your curve and then use them as below:

```
use_MyCurve (); // initialize the irreducible polynomial
EC_Domain_Parameters dp = MyCurve; // set the domain parameters
```

Note - You should ensure that the curve is not a *"weak* curve that could lessen security (see IEEE P1363 for more details of how to determine this).

Chapter 5

EC Keys

5.1 EC Private Keys

Elliptic Curve Private Keys have two member variables: `dp`, the EC domain parameters and `s`, the private key which is a large integer (`BigInt`) and must be kept secret.

5.1.1 Constructors

- `ECPrivKey (dp);`

  Generate an EC private key object with EC domain parameters dp and a random private key.

- `ECPrivKey (dp, s);`

  Generate an EC private key object with EC domain parameters dp and a private key s.

5.1.2 Methods

- `=`

  Assignment

5.2 EC Public Keys

Elliptic Curve Public keys have two member variables: dp, the EC domain parameters and W, the public key which is a point on the curve (Point).

5.2.1 Constructors

- `ECPubKey ();`

  Create an empty EC public key object.

- `ECPubKey (sk);`

  Calculate an EC public key object from an EC private key sk.

- `ECPubKey (dp, W);`

  Create an EC public key object with EC domain parameters dp and public key W.

5.2.2 Methods

- `bool valid ();`

  Checks if the EC public key object is valid

- `=`

  Assignment

5.3 EC Keys Example

Generate a random EC private key using the `NIST_B_163` curve and calculate the EC public key:

```
use_NIST_B_163 ();
EC_Domain_Parameters dp = NIST_B_163;
ECPrivKey sk (dp); // generate random EC private key
ECPubKey pk (sk); // calculate EC public key
```

Chapter 6

ECKAS-DH1

In ECKAS-DH1 (the Elliptic Curve Key Agreement Scheme, Diffie-Hellman 1), each party combines its own private key with the other party's public key to calculate a shared secret key which can then be used as the key for a symmetric encryption algorithm such as AES. Other (public or private) information known to both parties may be used as key derivation parameters to ensure that a different secret key is generated every session. This key agreement scheme is described in more detail in section 9.2 of the IEEE P1363 standard.

6.1 Functions

- `OCTETSTR ECKAS_DH1 (dp, s, Wi);`

  Calculates a 128 bit secret key from EC domain parameters `dp`, private key `s` and public key `Wi`. `s` belongs to one party, `Wi` belongs to the other and `dp` is common to both of them.

- `OCTETSTR ECKAS_DH1 (dp, s, Wi, P);`

  Calculates a 128 bit secret key from EC domain parameters `dp`, private key `s`, public key `Wi` and key derivation parameter `P` (an octet string). `s` belongs to one party, `Wi` belongs to the other and `dp` and `P` are common to both of them.

6.2 Example

In this example (Figure 6.1) party A and party B have previously decided to use the `NIST_B_163` curve.

User A	User B
`use_NIST_B_163 ();`	`use_NIST_B_163 ();`
`EC_Domain_Parameters dp = NIST_B_163;`	`EC_Domain_Parameters dp = NIST_B_163;`
`ECPrivKey skA (dp); // generate private key`	`ECPrivKey skB (dp); // generate private key`
`ECPubKey pkA (skA); // calculate public key`	`ECPubKey pkB (skB); // calculate public key`
`// Send pkA to B and obtain pkB`	`// Send pkB to A and obtain pkA`
`OCTETSTR K = ECKAS_DH1 (dp, skA.s, pkB.W));`	`OCTETSTR K = ECKAS_DH1 (dp, skB.s, pkA.W));`
`// Use K with a symmetric`	`// Use K with a symmetric`
`// encryption algorithm`	`// encryption algorithm`

Figure 6.1: ECKAS_DH1 Example

Chapter 7

Hash Algorithms

These functions implement the SHA-1 hash algorithm as specified in FIPS 180-1.

7.1 Functions

- `OCTETSTR SHA1 (x);`

  Calculate the SHA-1 hash of octet string x.

- `OCTETSTR SHA1 (x);`

  Calculate the SHA-1 hash of string x.

- `OCTETSTR SHA1 (x);`

  Calculate the SHA-1 hash of BigInt x.

7.2 Example

```
std::string M ("Message to be hashed");
OCTETSTR hash = SHA1(M);
```

Chapter 8

ECDSA

ECDSA (the Elliptic Curve Digital Signature Algorithm) is used to generate a digital signature of a message digest or hash. The signature consists of `c` and `d` which are two large integers (`BigInt`). This signature algorithm is described in more detail in sections 7.2.7 and 7.2.8 of the IEEE P1363 standard.

8.1 Constructors

- `ECDSA ();`

  Create an empty ECDSA signature object.

- `ECDSA (c, d);`

  Create an ECDSA signature object from `c` (`BigInt`) and `d` (`BigInt`).

- `ECDSA (sk, f);`

  Generate an ECDSA signature object from message digest `f` (`BigInt`) and private key `sk` (`ECPrivKey`).

8.2 Methods

- `verify (pk, f);`

  Verify the signature with message digest `f` (`BigInt`) and public key `pk` (`ECPubKey`).

- `Put`

  Output

- `=`

  Assignment

8.3 Example (generating a signature)

In this example, the user has previously generated a private key `sk`:

```
std::string M ("message");
ECDSA sig (sk, OS2IP (SHA1 (M)));
```

8.4 Example (verifying a signature)

In this example, the user tries to verify the signature generated in the previous example:

```
if (sig.verify (pk, OS2IP (SHA1 (M)))) {
  // Valid Signature
} else {
  // Invalid Signature
}
```

Chapter 9

ECIES

ECIES (the Elliptic Curve Integrated Encryption Scheme) combines elliptic curve asymmetric encryption and the AES symmetric encryption algorithm with the SHA-1 hash algorithm to provide an easy to use encryption scheme with message authentication support. An `ECIES` ciphertext object `(V, C, T)` consisting of EC public key `V`, encrypted message `C` and authentication tag `T` is generated from a message `M` and the recipient's EC public key `Wi`. The recipient decrypts this ciphertext with their EC private key and an exception is thrown if the authentication tag is invalid. This encryption scheme is described in more detail in section 11.3 of the IEEE P1363a draft standard.

9.1 Constructors

- `ECIES ();`

  Create an empty ECIES ciphertext object.

- `ECIES (V, C, T);`

  Create an `ECIES` ciphertext object with EC public key `V`, encrypted message `C (OCTETSTR)` and authentication tag `T (OCTETSTR)`.

- `ECIES (M, Wi);`

  Encrypt message `M` with EC public key `Wi` to create an `ECIES` ciphertext object.

9.2 Methods

- `OCTETSTR decrypt (sk);`

  Decrypt a ciphertext with EC private key `sk`. This method throws `borzoiException` if the authentication tag is invalid.

- `Put`

  Output

9.3 Example (encrypt a message)

In this example, the sender encrypts the message `M` using the recipient's EC public key `pk`:

```
ECIES CT (M, pk);
```

9.4 Example (decrypting an encrypted message)

In this example, the receiver decrypts the ciphertext `CT` using their EC private key `sk`:

```
try { // try to catch any exceptions if the tag is invalid
  M = CT.decrypt(sk); // decrypt using the private key
} catch (borzoiException e) {
  // Authentication Tag Invalid
}
```

Chapter 10

Symmetric Encryption (AES)

Symmetric encryption support is provided by the Advanced Encryption Symmetric (AES) encryption algorithm in Cipher Block Chaining (CBC) mode with a null initialization vector. This is described in more detail in the IEEE P1363a draft standard.

10.1 Functions

- `OCTETSTR AES_CBC_IV0_Encrypt (OCTETSTR KB, OCTETSTR M, int k=128);`

  Encrypt an octet string `M` with key `KB` of length `k` (default: 128 bits, can also be set to 192 or 256 bits).

- `OCTETSTR AES_CBC_IV0_Decrypt (OCTETSTR KB, OCTETSTR C, int k=128);`

  Decrypt an octet string `C` with key `KB` of length `k` (default: 128 bits, can also be set to 192 or 256 bits). This function throws `borzoiException` if the length of `C` or the padding is invalid.

10.2 Example (encrypt a message)

```
int keysize = 128; // 128 bit symmetric encryption key
OCTETSTR P; // The Key Derivation Parameter P is NULL
OCTETSTR key (keysize/8);

// generate keysize/8 octets of random data for the key
for (int i=0; i<(keysize/8); i++) {
  key[i] = (OCTET)gen_random ();
}
```

```
// encrypt M to produce ciphertext C
OCTETSTR C = AES_CBC_IV0_Encrypt (key, M, keysize);
```

10.3 Example (decrypting an encrypted message)

```
try { // try to catch any exceptions
  M = AES_CBC_IV0_Decrypt (key, C, keysize);
} catch (borzoiException e) { // print the error message and exit
  // C Invalid
}
```

Chapter 11

DER Encoding

DER (Distinguished Encoding Rules for ASN.1) is a standard way of encoding data for transmission over networks or communicating with other programs. The objects in this library are encoded according to the ASN.1 syntax in the ANSI X9.62, ANSI X9.62 and SEC 1 Elliptic Curve Cryptography standards.

11.1 Constructors

- `DER (OCTETSTR vin);`

  DER encode an octet string.

- `DER (ECPubKey pk);`

  DER encode an EC public key. This throws `borzoiException` if the EC Domain Parameter basis is invalid.

- `DER (ECPrivKey sk);`

  DER encode an EC private key. This throws `borzoiException` if the EC Domain Parameter basis is invalid.

- `DER (ECDSA sig);`

  DER encode an ECDSA signature

- `DER (ECIES ct);`

  DER encode an ECIES ciphertext. This throws `borzoiException` if the EC Domain Parameter basis is invalid.

  Currently, there is no standard `ASN.1` module syntax for ECIES ciphertexts so borZoi uses the following custom syntax:

  ```
  ECIES-Ciphertext-Value ::= SEQUENCE {
      V ECPubKey,
  ```

```
        C OCTET STRING,
        T OCTET STRING
    }
```

11.2 Methods

- `ECPubKey toECPubKey ();`

  Convert a DER string to an EC public key

- `ECPrivKey toECPrivKey ();`

  Convert a DER string to an EC private key

- `ECDSA toECDSA ();`

  Convert a DER string to an ECDSA signature

- `ECIES toECIES ();`

  Convert a DER string to an ECIES ciphertext

- `=`

  Assignment

- `Put`

  Output

11.3 Example (Decoding a DER encoded ECDSA signature)

```
ECDSA sig;
try { // try to catch any DER parsing errors
  sig = der_str.toECDSA (); // decode the DER string
} catch (borzoiException e) { // print the error message and exit
  e.debug_print ();
  return;
}
```

Chapter 12

HEX Encoding

This is a utility class for outputting octet strings and DER objects in hexadecimal format.

12.1 Constructors

- `HexEncoder (OCTETSTR vin);`

  Encode an octet string.

- `HexEncoder (DER der);`

  Encode a DER object.

12.2 Methods

- `=`

  Assignment

- `Put`

  Output

12.3 Example (Encoding a DER object for output to a stream)

```
HexEncoder h(der);                              // der, a DER object
std::cout << "DER data: " << h << std::endl; // output to cout
```

Chapter 13

Miscellaneous Functions

- `unsigned long gen_random();`

  Generate a random `unsigned long` using Microsoft's Crypto API on Windows systems and /dev/random on Linux systems.

- `OCTETSTR KDF2 (OCTETSTR Z, OCTETSTR::size\_type oLen, OCTETSTR P);`

  This is Key Derivation Function 2 (KDF2) from the IEEE P1363a draft standard. It generates a secret key of length `oLen` bytes from shared secret `Z` and key derivation parameter `P`. Exception `borzoiException` is thrown for invalid output lengths.

- `OCTETSTR MAC1 (OCTETSTR KB, OCTETSTR M);`

  This function implements MAC1 as described in the IEEE P1363 standard. It computes a HMAC message authentication code tag from secret key `KB` and message `M`.