CompactECC – Elliptic Curve Cryptography

C++ class template library suitable for embedded systems

Reference Manual

Revision 1.4

18th July 2010

ubisys

1 Introduction

Elliptic curve cryptography is a public-key cryptographic system based on the algebraic structure of elliptic curves over finite fields. Public-key cryptography is based on the complexity of certain mathematical problems, making them unsolvable in practice.

1.1 Elliptic curves

An elliptic curve is a plane curve, consisting of points satisfying the equation

$y^2 = x^3 + ax + b$, including the point ∞.

Koblitz curves are a special case. These are binary anomalous curves with coefficients $a, b \in \{0, 1\}$. This allows for certain optimizations.

1.2 Elliptic curve cryptography

The following cryptographic algorithms are using elliptic curves:

- Elliptic Curve Diffie-Hellman (ECDH), a key agreement scheme based on the Diffie-Hellman scheme
- Elliptic Curve MQV, a key agreement scheme based on MQV
- Elliptic Curve Digital Signature Algorithm (ECDSA)

Currently, CompactECC implements the ECDSA and ECDH algorithms.

1.3 Elliptic curve domain parameters

The parameters fully describing an elliptic curve are called the domain parameters and they must be agreed upon by all participants in an elliptic curve cryptographic system.

a, b	Coefficients specifying the elliptic curve $y^2 = x^3 + ax + b$
p	Modulus defining the finite field
G	Base point on the curve
n	Order of G

1.4 Security

Elliptic curve cryptographic systems are based on point multiplication on elliptic curves over finite fields. Their security is based on the assumption that the Elliptic Curve Discrete Logarithm Problem (ECDLP) is practically unsolvable for complex curves. While it is easy to calculate $R = dQ$, it is intractable to calculate the inverse, i.e. find d for a known R and Q [5].

1.5 Elliptic Curve Digital Signature Algorithm (ECDSA)

1.5.1 Advantages

Compared to the ordinary, RSA-based Digital Signature Algorithm (DSA), the main advantage of the ECDSA algorithm is the smaller key size at a comparable security level. Thus, less storage space is required for the keys and signatures as well as a smaller amount of RAM and computational power.

1.5.2 Key generation

The private key d is generated by randomly selecting an integer in the range $[1, p-1]$. The public key Q is derived from the private key through multiplication on the curve: $Q = dG$.

1.5.3 Signature generation

L_n is the bit length of the group order n, i.e. the bit length of the signature components as well as the bit length of the private key.

1. Calculate the hash over the message to be signed by using a cryptographic hash function. Set z to the L_n leftmost bits of the hash.
2. Select a random integer k in the range $[1, p-1]$.
3. Calculate $(x1, y1) = kG$ and set $r = x1 \ (mod\ p)$.
4. If $r = 0$, go back to step 2.
5. Calculate $s = k^{-1}(z + rd) \ (mod\ p)$.
6. If $s = 0$, go back to step 2.
7. The signature consists of the pair (r, s).

Note that the comparisons in step 4 and 6 are inherent to the algorithm and cannot be avoided. Depending on the randomly generated value k, it might be necessary to re-calculate a part of the signature or even the whole signature in very rare cases.

1.5.4 Signature verification

1. Verify that r and s are in the range $[1, p-1]$.
2. Calculate the hash over the message by using the same hash function used in the signature generation. Set z to the L_n leftmost bits of the hash.
3. Calculate $w = s^{-1} \ (mod\ p)$.
4. Calculate $u_1 = zw \ (mod\ p)$.
5. Calculate $u_2 = rw \ (mod\ p)$.
6. Calculate $(x_1, y_1) = u_1 G + u_2 Q$.
7. Verify $r = x1 \ (mod\ p)$.

2 Implementation overview

2.1 Namespace

All CompactECC types and functions are defined in the ‘cecc’ namespace.

2.2 Handling of big unsigned integer numbers

Asymmetric cryptographic algorithms such as ECC are based on big integer numbers. ECC, compared to RSA, uses relatively small numbers, but still consisting of several hundred bits. Common processors cannot handle them natively and thus they are not available to most programming languages, such as C/C++. Thus, a dedicated implementation is provided, which implements standard arithmetic functions and reduces them to several invocations of the available operations on ordinary integers. The mentioned functions are implemented in the C++ class template CBigUnsigned. Two parameters are available to parameterise the class template to adapt it to the used architecture.

2.3 Point representation

Points on the elliptic curve are stored in two-dimensional Cartesian coordinates. The class CPoint represents a simple container, using two CBigUnsigned members, representing the x and y coordinates. The point class is defined as a class template with two parameters, specifying the types to use for the CBigUnsigned class.

2.4 Finite fields

Elliptic curves operate on so-called finite fields. Finite field operations are implemented in the class CFiniteField. A specialized and optimized version for Koblitz curves operating on binary finite fields is implemented in the derived class CFiniteFieldBinary.

Further optimizations are possible for prime fields, by replacing the generic modulo-operation with an optimized implementation by exploiting features of the so-called generalized Mersenne numbers. For now, this optimization is only implemented for the secp192r1 curve. The corresponding finite field class is called CFiniteField192r1.

The finite field classes are defined as class templates with two parameters, specifying the types to use for the CBigUnsigned class. The finite field classes are used internally and do not require any user-interaction or parameterisation.

2.5 Elliptic curves

A generic elliptic curve representation is implemented in the class CEllipticCurve. For Koblitz curves, the class CKoblitzCurve provides an optimized implementation, exploiting certain features inherent to Koblitz curves.

Both classes are defined as class templates with three parameters. The first two parameters specify the types to use for the CBigUnsigned class. The third

parameter specifies the finite field to use, defaulting either to the generic CFiniteField or the optimized CFiniteFieldBinary for Koblitz curves.

Point multiplication on elliptic curves is optimized by using a sliding-window approach. This involves pre-computation of a set of base points (16 by default) during curve initialization, allowing processing several bits in a single step (4 bits for 16 base points).

The recommended curves defined in the "Standards for Efficient Cryptography (SEC)" [3] are predefined with their parameters. Hence, it is usually not required to deal with the CEllipticCurve class itself, but using the derived class definitions instead.

2.6 Elliptic Curve Digital Signature Algorithm (DSA)

The Elliptic Curve Digital Signature Algorithm is implemented in the classes CEllipticCurveDSASignOnly and CEllipticCurveDSA, which provide functions for creating and verifying signatures.

Both classes are defined as class templates with three parameters. The first two parameters define the types to parameterise CBigUnsigned. The third parameter defines the type of finite field used by the elliptic curve.

The "Sign" function to create signatures receives another template parameter, namely the type of the random number generator to use to generate the random number k during creating of the signature.

2.7 Generation of Random Numbers

The security of the Elliptic Curve Digital Signature Algorithm depends on the availability of proper random numbers. The random number generator available in the C/C++ runtime library is usually based on a simple linear feedback shift register (LFSR) and does not meet the requirements. This is especially a problem on embedded systems, as there is usually no stochastic entropy available to seed such a random number generator.

Thus, an interface to a user-specified random number generator is provided in the class CEllipticCurveDSA. Refer to section 6.10 for details.

A reference implementation is available using the Microsoft Windows Cryptographic API on a Windows Computer. A testing-only implementation is provided as well, which uses the unsafe random number generator of the C/C++ standard library.

An implementation on an embedded system should provide a way to sample some true random data, possibly from some attached peripherals (e.g. noise, network traffic, user interaction etc.).

3 Predefined curves

CompactECC includes the following predefined curves, recommended in "Standards for Efficient Cryptography (SEC), SEC2: Recommended Elliptic Curve Domain Parameters" [3].

Elliptic Curve	Elliptic Curve Class	Finite Field class
secp160k1	CEllipticCurve160k1	CBinaryField
secp160r1	CEllipticCurve160r1	CFiniteField
secp192k1	CEllipticCurve192k1	CBinaryField
secp192r1	CEllipticCurve192r1	CFiniteField192r1
secp224k1	CEllipticCurve224k1	CBinaryField
secp224r1	CEllipticCurve224r1	CFiniteField
secp256k1	CEllipticCurve256k1	CBinaryField
secp256r1	CEllipticCurve256r1	CFiniteField
secp384r1	CEllipticCurve384r1	CFiniteField
secp521r1	CEllipticCurve521r1	CFiniteField

Note that the same curve might have different names in different standards than those used in the SEC. For example, the ANSI X9.62 prime192v1 curve is equivalent to the secp192r1 curve.

If the stack-based implementation of CBigUnsigned is used, properly adapt the value of the enum CBigUnsigned::capacity for the curve with the greatest bit-length used in the particular application. Refer to section 6.1.2 for details.

4 Usage

A short usage example for creating and verifying signatures is provided based on the secp192r1 curve.

4.1 Signature generation

- Instantiate the curve and a random number generator

```
// Instantiate the curve
CElliptic Curve192r1<unsigned int, unsigned long long> curve;

// Instantiate a random number generator (c.f. section 2.7 and 6.10)
CRandomNumberGenerator rng;
```

- Define the private key (pre-generated)

  Usually, the private key is stored somewhere in flash and a CBigUnsigned instance must be created:

```
// Create the private key from the unsigned int array init_d[]
// Digits (of unsigned int type) must be stored with the
// least-significant digit first

const unsigned int anPrivateKey[] = {
    0x628f6ca5,
    0x1e3a45db,
    0xf4b12d89,
    0x594470af,
    0x70791d12,
    0xdeb8634e
};

CBigUnsigned<T, T2> privateKey(anPrivateKey, curve.m_nKeyDigits);
```

- Generate the private key on-the-fly (testing only)

  Alternatively, for testing, the private key may be generated on the fly, by creating a random number and verifying that it meets the requirements (i.e. it must be in the range $[1, p-1]$)

```
CBigUnsigned<T, T2> privateKey(curve.m_nKeyDigits);

do
{
    rng(privateKey);
    // ensure that privateKey is < m_p
    privateKey = privateKey % curve.m_field.m_p;
} while (privateKey == 0);
```

- Derive the public key from the private key (if required)

  The public key can be derived from the private key by multiplication on the curve:

```
CPoint<unsigned int, unsigned long long> publicKey;

// Either use curve.Multiply() or curve.WindowMultiply().
// The latter uses a sliding window approach and pre-computed points.
```

```
curve.WindowMultiply(publicKey, privateKey);
```

- Calculate the hash of the message

  The following fragment is only provided as an example for a message stored in the array abMessage. The user should calculate a cryptographic digest, e.g. SHA-1 or SHA-256. Note the SHA-1 is considered insecure nowadays.

```
CBigUnsigned<unsigned int, unsigned long long> digest;

// Calculate the digest - to be be implemented by the user
CalculateDigest(abMessage, sizeof(abMessage), digest,
                curve.m_nDigits);
```

- Instantiate the ECDSA class (or the ECDSA sign-only class)

```
// ECDSA class: sign and verify (requires the public key)
CEllipticCurveDSA<unsigned int, unsigned long long,
CFiniteField192r1<unsigned int, unsigned long long> >
    ecdsa(curve, publicKey);

// ECDSA sign-only class (public key not required)
CEllipticCurveDSASignOnly<unsigned int, unsigned long long,
    CFiniteField192r1<unsigned int, unsigned long long> > ecdsa(curve);
```

- Sign the hash of the message

```
CBigUnsigned<unsigned int, unsigned long long> r, s;

// Pass r, s to receive the signature
// the private key d, the message digest and the random number
// generator to generate k
ecdsa.Sign(r, s, d, digest, rng);
```

- (r, s) contain the signature of the message

4.2 Signature verification

- Instantiate the curve

```
CEllipticCurve192r1<unsigned int, unsigned long long> curve;
```

- Define the public key

  The private key is not known to the party which verifies the signature (except in test cases). Thus, the public key must be stored somewhere in memory.

```
const unsigned int anPublicKeyX[] = {
    0xe980fe4e,
    0xc642ac56,
    0xc4934fe9,
    0xab16793a,
    0x9c7ecc10,
    0x64400ec8
};
```

```
const unsigned int anPublicKeyY[] = {
    0x725f30f5,
    0x9587914a,
    0x73b7e74a,
    0x2a1d415d,
    0xc8debdcf,
    0xcbb2c125
};


// Create the public key Q (point on the curve) from the init data:
CPoint<unsigned int, unsigned long long>
    publicKey(anPublicKeyX, anPublicKeyY, curve.m_nKeyDigits);
```

- Instantiate the ECDSA class

```
CEllipticCurveDSA<unsigned int, unsigned long long,
    CFiniteField192r1<unsigned int, unsigned long long> >
    ecdsa(curve, publicKey);
```

- Calculate the hash of the message

  Use the same hash function as used by the signing party.

```
CBigUnsigned<unsigned int, unsigned long long> digest;

// Calculate the digest - to be be implemented by the user
CalculateDigest(abMessage, sizeof(abMessage), digest,
                curve.m_nDigits);
```

- Verify the signature

```
bool bPass = ecdsa.Verify(r, s, digest);
```

4.3 Notes on class instantiation

The instantiation of the CEllipticCurve and CEllipticCurveDSA classes might take a fair amount of time. This is due to the computation of base points for faster multiplication which are based on the point G (a curve parameter) for the CEllipticCurve class and based on the public key Q for the CEllipticCurveDSA class.

Note that the instantiation of the CEllipticCurve and classes is only necessary to be done once, not every time a signature needs to be created or verified.

5 Performance

5.1 ARM7TDMI / AT91SAM7S

The following values for memory consumption and runtime were measured at a clock speed of 48 MHz by using the Thumb instruction set and setting the compiler optimizations to “High: Speed”. All variables were instantiated on the stack. Memory consumption includes a small test program as well as required parts of the C/C++ runtime library.

5.1.1 Code size and RAM requirements

	Sign only		Sign and Verify	
Curve	Code	Stack	Code	Stack
secp160k1	20 kB	5.4 kB	22 kB	7.5 kB
secp160r1	19 kB	5.4 kB	21 kB	7.5 kB
secp192k1	21 kB	6 kB	22 kB	8.4 kB
secp192r1	20 kB	6 kB	22 kB	8.4 kB
secp224k1	21 kB	6.5 kB	22 kB	9.1 kB
secp224r1	20 kB	6.6 kB	22 kB	9.1 kB
secp256k1	21 kB	7.2 kB	22 kB	10 kB
secp256r1	21 kB	7.2 kB	22 kB	10 kB
secp384r1	19 kB	9.3 kB	20 kB	12.8 kB
secp521r1	20 kB	12.5 kB	21 kB	17.5 kB

Table 1 Code and RAM requirements for different curves on the ARM7 microprocessor

5.1.2 Runtimes

Curve	Sign	Verify
secp160k1	430 ms	865 ms
secp160r1	480 ms	960 ms
secp192k1	680 ms	1.38 s
secp192r1	410 ms	825 ms
secp224k1	980 ms	1.98 s
secp224r1	1.1 s	2.2 s
secp256k1	1.3 s	2.7 s
secp256r1	1.8 s	3.6 s
secp384r1	4.4 s	8.8 s
secp521r1	10 s	20 s

Table 2 Runtimes for signature creation and verification on an ARM7 microprocessor at 48 MHz

Remarks:

The secp192r1 curve uses a fast-reduction algorithm for the modulo-p operation.

An optimized implementation for signature creation on the secp192r1 curve on ARM7TDMI microprocessors is available.

5.2 ARM Cortex-M3 / ATSAM3S

CompactECC is also available for Cortex-M3 controllers and has been successfully tested on Atmel's ATSAM3S device running at 64MHz. In general, performance is superior to that of the SAM7S.

5.2.1 Code size and RAM requirements

TBD

5.2.2 Runtimes

TBD

6 Class Reference

6.1 CBigUnsigned

6.1.1 Declaration

The class CBigUnsigned is defined as a generic class template, parameterised by two type paremeters:

```
template<class T, class T2>
class CBigUnsigned;
```

The type parameter T specifies the internal storage type for each digit. T2 defines the result type of a multiplication of two variables of type T.

Most 32-bit compilers use `unsigned int` (32 bits) as their native type and provide an `unsigned long long` type of 64 bits[1].

In this case, the parameterised type to use would be `CBigUnsigned<unsigned int, unsigned long long>`.

6.1.2 Implementations

Currently, two distinct implementations are available: the default implementation allocates storage on the heap as required and allows to dynamically adjust the required capacity.

The alternative implementation provides a fixed amount of storage space inside the class instance, i.e. usually on the stack. Definition of the preprocessor symbol `__COMPACT_CRYPTO_STACK__` enables the alternative implementation. The amount of reserved space is defined by the value of the enum `CBigUnsigned::capacity`. The value is predefined to the mentioned preprocessor symbol, i.e. the defined value of the pre-processor symbol selects the amount of reserved space. Refer to table Table 3 for the required capacity for the different curve types on a 32 bit platform.

The heap-based implementation is usually slower than the stack-based implementation, as it requires memory allocation and deallocation.

Curves	Key size	Required capacity		
		8 bit	16 bit	32 bit
secp160k1, secp160r1	160	40	24	14
secp192k1, secp192r1	192	52	28	16
secp224k1, secp224r1	224	60	32	18
secp256k1, secp256r1	256	68	36	20
secp384r1	384	100	52	28
secp521r1	521	136	70	38

Table 3 Required storage capacity for the stack-based implementation on different platforms

[1] Although the type unsigned long long was only standardized in C99 and not yet in C++, it is available as a language extension in most compilers and will be included in the upcoming C++ standard (informally known as C++0x).

6.1.3 Constructors

```
// Construct from array of digits (which may be stored in ROM)
CBigUnsigned(const T *pData, const unsigned int nDigits);

// Copy constructor
CBigUnsigned(const CBigUnsigned &storage);

// This special copy-like constructor does not increment the
// reference counter. It returns a copy of the same object,
// probably with different (i.e. non-zero) offset
explicit CBigUnsigned(const CBigUnsigned &storage,
    const unsigned int nOffset);

// Construct instance with a given number of digits
// Does NOT clear the contents. Use Clear() for this purpose.
explicit CBigUnsigned(const unsigned int nCount = 0,
    const bool bAllocate = true);

// Constructor-like static function: creates an instance from an encoded
// byte stream, stored with the most-significant byte first
// (mainly for testing)
static CBigUnsigned FromByteArray(const unsigned char *pData,
                                  const unsigned int cbData);
```

6.1.4 Operators

6.1.4.1 Assignment operators

```
// Assign value of another CBigUnsigned instance
CBigUnsigned &operator=(const CBigUnsigned &);

// Assign a single digit of type T (all other digits are cleared)
CBigUnsigned &operator=(const T);
```

6.1.4.2 Comparison operators

```
// Compare with a single digit of type T
bool operator==(const T digit) const;
bool operator!=(const T digit) const;

// Compare with another CBigUnsigned
bool operator==(const CBigUnsigned &a) const;
bool operator!=(const CBigUnsigned &a) const;
bool operator>=(const CBigUnsigned &a) const;
bool operator<=(const CBigUnsigned &a) const;
bool operator>(const CBigUnsigned &a) const;
bool operator<(const CBigUnsigned &a) const;
```

6.1.4.3 Arithmetic operators

```
CBigUnsigned operator+(const CBigUnsigned &a) const;
CBigUnsigned operator+(const T a) const;

CBigUnsigned operator-(const CBigUnsigned &a) const;

CBigUnsigned operator*(const CBigUnsigned &a) const;
CBigUnsigned operator*(const T a) const;

CBigUnsigned operator/(const CBigUnsigned &a) const;
CBigUnsigned operator%(const CBigUnsigned &a) const;

CBigUnsigned operator<<(const unsigned int nValue) const;
CBigUnsigned operator>>(const unsigned int nValue) const;
```

6.1.4.4 Non-standard arithmetic

```
// Computes a = b * 2^c and returns carry
static T ShiftLeftEx(CBigUnsigned &a, const CBigUnsigned &b, const T c);

// Computes a = b / 2^c and returns carry
static T ShiftRightEx(CBigUnsigned &a, const CBigUnsigned &b, const T c);

// Calculates a = b + c and returns carry
static T AddEx(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &c);

// Calculates a = b - c and returns borrow
static T SubtractEx(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &c);

// Calculates a = b + c * d (where c is a digit) and returns carry
static T AddMultiplyEx(CBigUnsigned &a, const CBigUnsigned &b,
    const T c, const CBigUnsigned &d);

// Calculates a = b - c * d (where c is a digit) and returns borrow
static T SubtractMultiplyEx(CBigUnsigned &a, const CBigUnsigned &b,
    const T c, const CBigUnsigned &d);

// Calculates a = c div d and b = c mod d
static void DivideEx(CBigUnsigned &a, CBigUnsigned &b,
    const CBigUnsigned &c, const CBigUnsigned &d);

// Calculates a = gcd(b, c)
static void CalculateGCD(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &c);

// Calculates a = b^c mod d;
static void PowerModulo(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &c, const CBigUnsigned &d);

// Calculates a = 1/b mod c;
static void InverseModulo(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &c);

// Calculates a = (b + c) mod d
static void AddModulo(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &c, const CBigUnsigned &d);

// Calculates a = (b - c) mod d
static void SubtractModulo(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &c, const CBigUnsigned &d);

// Calculates a = b * c mod d, where d is generalized Mersenne prime,
// d = 2^key_bits - omega
static void MultiplyModuloOptimized(CBigUnsigned &a,
    const CBigUnsigned &b, const CBigUnsigned &c,
    const CBigUnsigned &d, const CBigUnsigned &omega);

// Calculates a = b^2 according to the Standard Squaring Algorithm in
// "High-Speed RSA Implementation"
static void Square(CBigUnsigned &a, const CBigUnsigned &b);

// Calculates a = b^2 mod d according to the Standard Squaring
// Algorithm in "High-Speed RSA Implementation"
static void SquareModulo(CBigUnsigned &a, const CBigUnsigned &b,
    const CBigUnsigned &d);

// Calculates a = b^2 mod d according to the Standard Squaring
// Algorithm in "High-Speed RSA Implementation". Optimized variant,
// where d is generalized Mersenne prime
static void SquareModuloOptimized(CBigUnsigned &a, const CBigUnsigned &b,
     const CBigUnsigned &d, const CBigUnsigned &omega);
```

6.2 CPoint

6.2.1 Declaration

```
template<class T, class T2>
class CPoint;
```

6.2.2 Constructors

```
CPoint();

// Create a CPoint with nDigits, initialize X and Y coordinates
// from the given init values (nDigits each)
CPoint(const T *const pnX, const T *const pnY,
    const unsigned int nDigits);

// Assignment constructor
CPoint(const CPoint &point);
```

6.2.3 Attributes

```
// X and Y coordinates
CBigUnsigned<T, T2> m_x;
CBigUnsigned<T, T2> m_y;
```

6.2.4 Operations

```
// Clear X and Y coordinate, i.e. set to zero
void Clear();

// Compare point to (0, 0)
bool IsZero() const;

// Compare point with another point
bool operator==(const CPoint &point) const;
```

6.3 CFiniteField

6.3.1 Declaration

```
template<class T, class T2>
class CFiniteField;
```

6.3.2 Constructors

```
// Construct the finite field
// The initializer of the prime modulus p and the number of digits
// must be given
CFiniteField(const T *pnP, const unsigned int nDigits);
```

6.3.3 Attributes

```
// the prime modulus p
const CBigUnsigned<T, T2> m_p;
```

6.3.4 Operations

```
// Calculates a = b * c mod m_p
void Multiply(CBigUnsigned<T, T2> &a, const CBigUnsigned<T, T2> &b,
    const CBigUnsigned<T, T2> &c) const;

// Calculates a = b^2 mod m_p
void Square(CBigUnsigned<T, T2> &a,
    const CBigUnsigned<T, T2> &b) const;

// Calculates a = (b - c) mod m_p
void Subtract(CBigUnsigned<T, T2> &a, const CBigUnsigned<T, T2> &b,
     const CBigUnsigned<T, T2> &c) const;

// Calculates a = (b + c) mod m_p
void Add(CBigUnsigned<T, T2> &a, const CBigUnsigned<T, T2> &b,
     const CBigUnsigned<T, T2> &c) const;

// Calculates a = 1/b mod m_p;
void Inverse(CBigUnsigned<T, T2> &a,
     const CBigUnsigned<T, T2> &b) const;
```

6.4 CFiniteFieldBinary

6.4.1 Declaration

```
template<class T, class T2>
class CFiniteFieldBinary : public CFiniteField<T, T2>;
```

6.4.2 Constructors

```
// Construct binary field.
// Initialize p and Omega with the specified initializers
// and number of digits
CFiniteFieldBinary(const T *pnP, const T *pnOmega,
                   const unsigned int nDigits);
```

6.4.3 Attributes

```
// Omega, p = 2^m - omega
const CBigUnsigned<T, T2> m_omega;
```

6.4.4 Overloaded method

```
// Calculates a = b * c mod m_p
void Multiply(CBigUnsigned<T, T2> &a, const CBigUnsigned<T, T2> &b,
              const CBigUnsigned<T, T2> &c) const;

// Calculates a = b^2 mod m_p
void Square(CBigUnsigned<T, T2> &a, const CBigUnsigned<T, T2> &b) const;
```

6.5 CFiniteField192r1

6.5.1 Declaration

```
template<class T, class T2>
class CFiniteField192r1 : public CFiniteField<T, T2>;
```

6.5.2 Constructors

```
// Construct the finite field with the given initialize and number
// of digits for the prime modulus p
CFiniteField192r1(const T *pnP, const unsigned int nDigits);
```

6.5.3 Overloaded methods

```
// Calculates a = b * c mod m_p using an optimized modulo implementation
void Multiply(CBigUnsigned<T, T2> &a, const CBigUnsigned<T, T2> &b,
              const CBigUnsigned<T, T2> &c) const;

// Calculates a = b^2 mod m_p using an optimized modulo implementation
void Square(CBigUnsigned<T, T2> &a,
            const CBigUnsigned<T, T2> &b) const;
```

6.6 CEllipticCurve

6.6.1 Declaration

```
template<class T, class T2, class F = CFiniteField<T, T2> >
class CEllipticCurve;
```

6.6.2 Constructors

```
// Assumption p, a, b, gx, gy have key_bits / digit_bits digits, r
// has got one more digit
CEllipticCurve(const unsigned int nKeyBits, const F &field,
     const T *pnA, const T *pnB, const T *pnGx, const T *pnGy, const T *pnR);
```

6.6.3 Attributes

```
// Number of key digits
const unsigned int m_nKeyDigits;

// Finite field (prime, Koblitz)
const F &m_field;

// curve's coefficients, a
const CBigUnsigned<T, T2> m_a;

// curve's coefficients, b
const CBigUnsigned<T, T2> m_b;

// base point, a point on e of order r
const CPoint<T, T2> m_g;

// a positive, prime integer dividing the number of points on e
const CBigUnsigned<T, T2> m_r;
```

6.6.4 Operations

```
// P0 = P1 + P2
void Add(CPoint<T, T2> &p0, const CPoint<T, T2> &p1,
     const CPoint<T, T2> &p2) const;

// (P0,Z0) = (P1,Z1) + (P2,Z2) in Jacobian projective coordinate space
void AddProjective(CPoint<T, T2> &p0, CBigUnsigned<T, T2> &z0,
     const CPoint<T, T2> &p1, const CBigUnsigned<T, T2> &z1,
     const CPoint<T, T2> &p2, const CBigUnsigned<T, T2> &z2) const;

// (P0,Z0) = 2*(P1,Z1)
void DoubleProjective(CPoint<T, T2> &p0,
     CBigUnsigned<T, T2> &z0, const CPoint<T, T2> &p1,
     const CBigUnsigned<T, T2> &z1) const;

// P0 = n * P1 (scalar point multiplication)
void Multiply(CPoint<T, T2> &p0, const CPoint<T, T2> &p1,
     const CBigUnsigned<T, T2> &n) const;

// Precompute array of base points for sliding window
void Precompute(const CPoint<T, T2> &p,
     CPoint<T, T2> *pBasePoints);

// P0 = n * basepoint
// (scalar point multiplication, optimized sliding window algorithm)
void WindowMultiply(CPoint<T, T2> &p0,
     const CBigUnsigned<T, T2> &n) const;

// P0 = n * basepoint
// (scalar point multiplication, optimized sliding window algorithm)
void WindowMultiply(CPoint<T, T2> &p0, const CBigUnsigned<T, T2> &n,
     const CPoint<T, T2> *pBasePoints) const;
```

6.7 CKoblitzCurve

Defines a Koblitz curve, operating on a binary finite field (class CFiniteFieldBinary). Distinguished behaviour different from the CEllipticCurve behaviour is implemented in the class CFiniteFieldBinary, except for the constructor.

6.7.1 Declaration

```
template<class T, class T2, class F = CFiniteFieldBinary<T, T2> >
class CKoblitzCurve : public CEllipticCurve<T, T2, F>;
```

6.7.2 Constructors

```
// Assumption p, omega, a, b, gx, gy have key_bits / digit_bits digits,
// r  has got one more digit
CKoblitzCurve(const unsigned int nKeyBits, const F &field,
              const unsigned int *pnA, const unsigned int *pnB,
              const unsigned int *pnGx, const unsigned int *pnGy,
              const unsigned int *pnR);
```

6.8 CEllipticCurveDSASignOnly

6.8.1 Declaration

```
template<class T, class T2, class F >
class CEllipticCurveDSASignOnly;
```

6.8.2 Constructor

```
CEllipticCurveDSASignOnly (CEllipticCurve<T, T2, F> &ec);
```

6.8.3 Attributes

```
CEllipticCurve<T, T2, F> &m_ec;
```

6.8.4 Operations

```
// Sign the message. (r, s) is the ECC signature, d is the
// private key, digest is an appropriate hash value (MD5, SHA), and
// GenerateRandom is a function object of class type R providing a
// cryptographically strong random number.

template<class R>
void Sign(CBigUnsigned<T, T2> &r, CBigUnsigned<T, T2> &s,
    const CBigUnsigned<T, T2> &d,
    const CBigUnsigned<T, T2> &digest,
    R &GenerateRandom) const;
```

6.9 CEllipticCurveDSA

6.9.1 Declaration

```
template<class T, class T2, class F>
class CEllipticCurveDSA : public CEllipticCurveDSASignOnly;
```

6.9.2 Constructor

```
CEllipticCurveDSA(CEllipticCurve<T, T2, F> &ec,
    const CPoint<T, T2> &key);
```

6.9.3 Attributes

```
CPoint<T, T2> m_base[CEllipticCurve<T, T2>::points];
```

6.9.4 Operations

```
// Verify the message. (r, s) is the ECC signature, digest is an
// appropriate hash value, calculated with the same algorithm that
// the signer used. The stored public key is used for verification
bool Verify(const CBigUnsigned<T, T2> &r,
    const CBigUnsigned<T, T2> &s,
    const CBigUnsigned<T, T2> &digest) const;
```

6.10 Random number generator

The random number generator provided to the Sign() method of the CEllipticCurveDSA and CEllipticCurveDSASignOnly classes must be a so-called “function object” or “functor”, i.e. it must be of class type and provide the “function call operator” operator() to generate a random number:

```
template<class T, class T2>
class CRandomNumberGenerator
{
    // Operations
    public:
        void operator()(CBigUnsigned<T, T2> &random);
};
```

In a specific project, parameters T and T2 to CBigUnsigned are determined, thus a simpler non-template definition can be used as well:

```
class CRandomNumberGenerator
{
    // Operations
    public:
        void operator()
            (CBigUnsigned<unsigned int, unsigned long long> &random);
};
```

7 Evaluation

Evaluation releases for Atmel’s SAM3S-EK, SAM7S-EK, and SAM7X-EK evaluation boards are available upon request. These images can be flashed on the boards with SAM-BA. The debug unit can be connected to a standard terminal application via serial port (115200, 8, N, 1, no handshake) to observe the output. There is also an output pin available for timing measurements on a scope or logic analyzer, for example. Other platforms supported upon request. Please contact ubisys support (support@ubisys.de) if you are interested in evaluating any of the cryptographic libraries.

8 References

[1] FIPS 186-2, Digital Signature Standard. Federal Information Processing Standards Publication 186-2, 2000.

[2] Certicom Research, Standards for Efficient Cryptograhpy, SEC 1: Elliptic Curve Cryptography, 2000, Working Draft, Certicom Corp.

[3] Certicom Research, Standards for Efficient Cryptograhpy, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0, 2000

[4] Certicom Research, Guidelines for Efficient Cryptography, GEC 2: Test Vectors for SEC 1, Working Draft, 1999, Certicom Corp.

[5] Certicom Research, The Elliptic Curve Digital Signature Algorithm (ECDSA), 2001, Certicom Corp.

[6] D. Johnson, A. Menezes, S. Vanstone, The Elliptic Curve Digital Signature Algorithm (ECDSA), 2001, Certicom Corp.

[7] N. Sklavos and X. Zhang, Wireless Security and Cryptography: Specifications and Implementations, 2007, CRC Press

[8] American National Standard X9.62: “Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECSDA)”

9 Revision History

Revision	Date	Changes
1.0	26th October 2005	Initial Version
1.1	10th July 2007	Separation of signing and verifying, improved random number generator interface
1.2	28th August 2009	Finite field abstraction and optimization for the secp192r1 curve. Separate namespace 'cecc' defined. Performance data included.
1.3	15th September 2009	Introduce a static, constructor-like function FromByteArray() in CBigUnsigned. Include required CBigUnsigned capacity for 8- and 16-bit systems.
1.4	18th July 2010	Updated for Cortex-M3/ATSAM3S Notice on availability of ECDH

10 License

11 Contact

UBISYS TECHNOLOGIES GMBH
HARDWARE AND SOFTWARE DESIGN
ENGINEERING AND CONSULTING
AM WEHRHAHN 45
40211 DÜSSELDORF
GERMANY
T: +49. 211. 54 21 55 00
F: +49. 211. 54 21 55 99
www.ubisys.de
info@ubisys.de