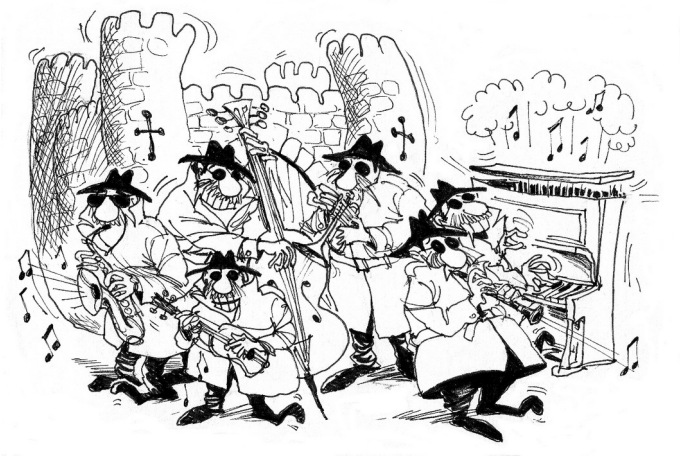
The BC-CSharp
(Bouncy Castle .NET API)
User Guide (Draft)

Version: 1.0.0 Date: 07/17/2023

Oreste Panaia



Legion of the Bouncy Castle Inc.
(ABN 84 166 338 567)
https://www.bouncycastle.org

Copyright and Trademark Notice

Sponsored By

https://www.keyfactor.com

https://www.cryptoworkshop.com

Acknowledgements

Crypto Workshop would like to acknowledge that its contribution has largely been made possible through its clients purchasing Bouncy Castle support agreements.

To clients and our anonymous donors, we are grateful. Thanks.

For further information about this distribution, or to help support this work further, please contact us at office@bouncycastle.org.

Table of Contents

1 Introduction

This document is a guide to the use of the Legion of the Bouncy Castle (BC) C-Sharp (C#) Application Programming Interface (API); BC C# API for short, and how it presents the cryptographic functions/algorithms. It does not directly provide details on the cryptographic algorithms used, unless required in the examples which follow. The document is meant to provide details and examples on how to use the .NET module.

The reader is assumed to have an understanding of C# and also to have had some exposure to the principals of cryptography. Having an existing understanding of the System.Security.Cryptography (both the Microsoft Windows .NET Framework API and the cross-platform .NET 5+ versions) is useful in following the examples below – however, the **BouncyCastle.Cryptography.dll** assembly is standalone.

The examples are not meant to be definitive, but they should provide you with a good overview of what can be done with the BC C# API. For simplicity the examples in this document do not include the using directive statement, but you can find the full source for them as well as a user guide at https://www.bouncycastle.org/csharp.

2 Getting Started

This chapter provides a quick look at the set up and configuration of the Bouncy Castle C# cryptographic assembly **BouncyCastle.Cryptography.dll**. In what follows we refer to this cryptographic assembly as the "Provider" or "Bouncy Castle Provider". The Bouncy Castle provider can either be installed via the Global Assembly Cache (GAC) or used directly during execution. You will need to find out the process involved for the **BouncyCastle.Cryptography.dll** assembly if you intend to install in the GAC or other known assembly path.

You can access the assembly directly at compile time and make this available to your application. Under Linux ensure you have the correct version of ***dotnet*** installed. The current assembly is complied against .NET 6.0 and .NET 4.6.1 – both of these will build the assembly directly using either the ***Linux dotnet*** or ***MS Windows dotnet*** executable. Once you have the source code, navigate to the Linux folder: **`/home/../bc-csharp/crypto/src`** (or to the folder **`c:\..\bc-csharp\crypto\src`** if you are using MS Windows). Then it is a simple matter to execute the command: **`dotnet clean`** followed by **`dotnet build`**.

Your dotnet framework should come with the ***msc*** (Mono C# compiler). Create your application and then use msc to compile the application to use with the assembly. Under MS Windows, the setup is similar except you will use the ***csc.exe*** compiler instead. The compiler will produce an executable which can access the assembly during execution of your application. Alternatively as a quick start, create a MS project file for your application (or project). The example below provides a very simple MS project file which will allow you to use the BC C# assembly from your application:

Example 1 – Basic .Net Project Using BC C# Assembly

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <RootNamespace>my_dotnet</RootNamespace>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <StartupObject>TestExamples</StartupObject>
  </PropertyGroup>

 <ItemGroup>
    <Reference Include="BouncyCastle.Cryptography.dll">
      <HintPath>./BouncyCastle.Cryptography.dll</HintPath>
    </Reference>
  </ItemGroup>

</Project>
```

Once you have the project file and the assembly, the command:

```
dotnet run --project my-dotnet.csproj
```

will execute your main method in the specified class (`TestExamples` in the above example).

2.1 Brief Introduction to Random Numbers

In the area of cryptography, random numbers are essential. There are many areas in which random numbers are used, for example in generating keys for symmetric ciphers, helping in generating large numbers (generally primes) for asymmetric algorithms and for introducing ***nonces***[1] into a cryptographic algorithm to stop (or reduce) so called ***replay***[2] attacks.

There are at least three meanings when one mentions random numbers. These are:

1. ***TRNG***, True Random Number Generator – these are systems which generate numbers (or bit strings if you prefer) which come from a physical random source such as noise from heat/voltage dissipation, noise from radioactive decay or from atmospheric noise (see www.random.org) and many more random physical processes. If your hardware supports a TRNG, then this can accessed from your C# code.
2. ***PRNG***, Pseudo-Random Number Generator (also known as ***DRBG***, Deterministic Random Bit Generator) – these are mathematical algorithms which generate numbers from a function usually initialised using a integer termed a ***seed*** value. Naturally these numbers are not truly random, since there are finite number of such values which can be generated – eventually the sequence of numbers will repeat. (Contrast this to something like an atmospheric noise TRNG which can be approximated as a continuous set of values. Thus for all practical purposes, with a TRNG we can generate a infinite number of distinct values.)
3. ***CSPRNG***, Cryptographically Secure Pseudo-Random Number Generator – these again use a mathematical algorithms in which generate random bit strings. The difference between a PRNG and a CSPRNG is that the later must be resistant to cryptographic attacks.

In Bouncy Castle C# cryptographic assembly it is possible to use all three of the above. For a TRNG your code will need to access the hardware and once that is achieved, you can use the implement the interface class **`IEntropySourceProvider`**. C# directly supports a modified version of a linear congruence alogorithm which can be used as a PRNG. We use this in the examples which follow below since we wish to obtain consistent keys and cipher text during testing.

Example 2 – Default C# PRNG

```
public static byte[] exampleDefaultCSharpRandom()
{
    ExValues.fixedRandomSeed = 151;
    ExValues.fixedRandom = new Random(ExValues.fixedRandomSeed);
    byte[] bytes = new byte[32];
    ExValues.fixedRandom.NextBytes(bytes);
    return bytes;
}
```

The code in the example above will provide you with a consistent set of “random” numbers useful for testing. Executing the above code twice will result in the same array of bytes being generated. Note that C# also provides a cryptographically secure PRNG in the class **`RandomNumberGenerator`**. The BC API also provides for PRNG and CSPRNG as in the examples below.

1 A ***nonce*** is an integer – within a specified range – which is a random integer or a pseudo-random integer.

2 A ***replay*** attack, in simple terms, occurs when an adversary intercepts a cryptographic message/data and re-sends that message/data at a later time.

Example 3 – BC Default SecureRandom

```
public static byte[] exampleDefaultSecureRandom()
{
    SecureRandom defaultSecureRandom = new SecureRandom();
    byte[] bytes = new byte[32];
    defaultSecureRandom.NextBytes(bytes);
    return bytes;
}
```

The default SecureRandom is based upon the SHA256 digest algorithm. The seed value is supplied via the C# internal class **`RandomNumberGenerator`**. Executing the above code twice shows that the set of bytes are different. **`SecureRandom`** generated as in the example above is cryptographically secure and is sufficent in most cryptographic cases. However, the BC API also provides for several NIST SP.800-90A recommended methods in generating cryptograpcially secure random numbers. The NIST document recommends DRBG based on

- hash functions,
- ***HMAC***s (known as Keyed-Hash Message Authentication Codes or Hash-based Message Authentication Codes), and
- block ciphers.

The example below shows the generation of a **`SecureRandom`** object based upon a HMAC.

Example 4 – BC HMAC SecureRandom

```
public static byte[] exampleHMACSecureRandom(IDigest digestToUse, byte[] nonce)
{
    SP800SecureRandomBuilder hMacSecureRandomBuilder =
        new SP800SecureRandomBuilder(ExValues.cSharpFixedRandom, false);

    hMacSecureRandomBuilder.SetPersonalizationString(ExValues.personalizationString);
    // hMacSecureRandomBuilder.SetSecurityStrength(256); default is 256 bits
    // hMacSecureRandomBuilder.SetEntropyBitsRequired(256); default is 256 bits

    HMac hmacDigest = new HMac(digestToUse);
    SecureRandom hmacSecureRandom = hMacSecureRandomBuilder.BuildHMac(hmacDigest, nonce, false);
    byte[] bytes = new byte[32];
    hmacSecureRandom.NextBytes(bytes);
    return bytes;
}
```

3 Symmetric Cipher Algorithms

Symmetric cryptographic algorithms can be categorised into two distinct flavours:

- ***Block Ciphers*** which encrypt a fixed size ordered set of plain text bytes, known as a block, one block at at time. In this case if the plain text does not come fit into an integral number of blocks, then padding is used to fill the remaining block.
- ***Stream Ciphers*** convert one byte of plain text directly into a byte of cipher text. In contrast to Block Ciphers, no padding is required.

Things however, get more complicated than the above two categories when talking about symmetric ciphers. In addition we have the concept of "modes" of operation for block ciphers. Depending on the mode of operation, certain block ciphers can operate as stream ciphers. (Note however, that a dedicated stream cipher can not be converted, using some mode, into a block cipher.) In summary then, we have

- Block Ciphers which operate in certain modes. Certain modes allow certain block ciphers to operate as a streaming cipher.
- Stream Ciphers which operate in their own dedicated modes.

As mentioned above, block ciphers operate on a fixed block size (usually 64, 128 or 256 bits). Therefore, in certain modes, that is, modes which don't convert to streaming, where a block cipher is used, if the plain text is not a multiple of the block size, then padding needs to be added to the plain text. In contrast, stream ciphers, or block ciphers used as streams, these do not require this step of adding padding.

The BC C# API essentially consists of a series of "engines" and a series of "modes". An engine can be thought of as the underlying mathematical permutation algorithm used to generate the cipher. (More technically if you are inclined – an engine with has a k-bit key is a bijection from the set 2^k bits applied to a string of n-bits.) A mode of operation of a block cipher can be thought of as a process (or algorithm) which specifies how to apply an n-bit block cipher to achieve this.

To make use of symmetric encryption available in the BC C# API, in basic terms, a cipher engine is chosen and a compatible encryption mode is selected. Next, generate a (random) key which is compatible with the engine selected and finally, if padding is required, choose a type of padding, again, compatible with the mode and engine. Here is an example of the most basic block cipher available in the API – which provides a template for other symmetric ciphers. (Other symmetric cipher examples will follow in the sections below.)

Example 5 – Random Symmetric Key Generation

```
public static ICipherParameters keyParameterGeneration(int keySize)
{
    CipherKeyGenerator keyGen = new CipherKeyGenerator();
    keyGen.Init(new KeyGenerationParameters(ExValues.cSharpFixedRandom, 256));
    KeyParameter keyParam = keyGen.GenerateKeyParameter();
    return keyParam;
}
```

The above example will generate a random key of size 256 bits. Alternatively, if you already have a key, then Example 6 shows how to generate a given fixed key. In almost all cases, you should use Example 5 above to generate a cryptographic secure key for you rather than substituting your own set of bytes as a key.

Example 6 – Fixed Symmetric Key Generation

```
public static ICipherParameters keyParameterGeneration(byte[] myKey)
{
    ICipherParameters keyParam = new KeyParameter(myKey);
    return keyParam;
}
```

Example 7 below shows the common steps required for encryption and decryption. The mode of operation is ECB, Electronic Code Book mode and the engine chosen is AES. ECB mode operates on blocks and the plain text must be padded to align onto a block. The example also demonstrates how padding is added to the encryption/decryption process.

Example 7 – ECB Mode Symmetric Cipher

```
public static byte[] ecbPaddedEncrypt(ICipherParameters keyParam, byte[] plainTextData)
{
    // First choose the "engine", in this case AES
    IBlockCipher symmetricBlockCipher = new AesEngine();

    // Next select the mode compatible with the "engine", in this case we use the simple ECB mode
    IBlockCipherMode symmetricBlockMode = new EcbBlockCipher(symmetricBlockCipher);

    // Finally select a compatible padding, PKCS7 which is the default
    IBlockCipherPadding padding  = new Pkcs7Padding();

    PaddedBufferedBlockCipher ecbCipher =
        new PaddedBufferedBlockCipher(symmetricBlockMode, padding);

    // apply the mode and engine on the plainTextData
    ecbCipher.Init(true, keyParam);
    int blockSize = ecbCipher.GetBlockSize();
    byte[] cipherTextData = new byte[ecbCipher.GetOutputSize(plainTextData.Length)];
    int processLength =
        ecbCipher.ProcessBytes(plainTextData, 0, plainTextData.Length, cipherTextData, 0);
    int finalLength = ecbCipher.DoFinal(cipherTextData, processLength);
    byte[] finalCipherTextData = new byte[cipherTextData.Length - (blockSize - finalLength)];
    Array.Copy(cipherTextData,0,finalCipherTextData,0,finalCipherTextData.Length);

    return finalCipherTextData;
}

public static byte[] ecbPaddedDecrypt(ICipherParameters keyParam, byte[] cipherTextData)
{
    // First choose the "engine", in this case AES
    IBlockCipher symmetricBlockCipher = new AesEngine();

    // Next select the mode compatible with the "engine", in this case we use the simple ECB mode
    IBlockCipherMode symmetricBlockMode = new EcbBlockCipher(symmetricBlockCipher);

    // Finally select a compatible padding, PKCS7 which is the default
    IBlockCipherPadding padding  = new Pkcs7Padding();

    PaddedBufferedBlockCipher ecbCipher =
        new PaddedBufferedBlockCipher(symmetricBlockMode, padding);
```

```
// apply the mode and engine on the plainTextData
ecbCipher.Init(false, keyParam);
int blockSize = ecbCipher.GetBlockSize();
byte[] plainTextData = new byte[ecbCipher.GetOutputSize(cipherTextData.Length)];
int processLength  =
    ecbCipher.ProcessBytes(cipherTextData, 0, cipherTextData.Length, plainTextData, 0);
int finalLength = ecbCipher.DoFinal(plainTextData, processLength);
byte[] finalPlainTextData = new byte[plainTextData.Length - (blockSize - finalLength)];
Array.Copy(plainTextData,0,finalPlainTextData,0,finalPlainTextData.Length);

return finalPlainTextData;
}
```

3.1 Block Cipher Modes Of Operation

As mentioned above, the BC C# API consists of various cipher engines and various modes of operation. For simplicity, we can think of a block cipher mode as the process involved on how different blocks of plain text should be encrypted and decrypted. The table below provides a list of the most common block cipher modes which are available in the BC C# API.

Mode	Algorithm
CBC	Cipher Block Chaining, requires an initialisation vector[3] (IV).
CCM	Counter Cipher Block Chaining Message, requires an initialisation vector. Used for AEAD[4] algorithms.
CFB	Cipher Feed Back – can be used as a streaming cipher, requires an initialisation vector.
CTR	Counter Mode – can be used as a streaming cipher, requires an initialisation vector and a counter.
EAX	Encrypt Authenticate Translate – used for AEAD algorithms.
ECB	Electronic Code Book
FF1	Format-preserving Feistel-based Encryption Mode
GCM	Galois Counter Mode – can be used as a streaming cipher, requires an initialisation vector.
OCB	Offset codebook mode, requires an initialisation vector and a counter.
OFB	Output Feed Back – can be used as a streaming cipher, requires an initialisation vector.

The other part of the encryption/decryption equation is the actual cipher engines which can be used by the modes in the table above. The table below provides a list of the most common block cipher engines which are available on the BC C# API.

3 An initialization vector (IV) is a known random set of bytes which are fed into the encryption algorithm during encryption and decryption. The number of bytes is usually the same as the block size of the engine used.

4 Authenticated Encryption with Associated Data (AEAD) modes are algorithms which are built to provide authentication apart from the encrypted data.

Engine Algorithm	Key Size	Mode Algorithm
AES	128, 192, 256	ECB, CBC, CFB8, CFB128, OFB, CTR, CCM, GCM, FF1
ARIA	128, 192, 256	ECB, CBC, CFB8, CFB128, OFB, CTR, CCM, GCM, FF1
Blowfish	128,256 (variable)	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
Camellia	128, 192, 256	CBC, CCM, CFB8, CFB128, CTR, ECB, GCM, OCB, OFB
Cast5 (CAST-128)	128	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
Cast6 (CAST-256)	128, 160, 192, 224, 256	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
GOST	256	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
IDEA	128	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
SEED	128	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
Serpent	129,192,256	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
Skipjack		ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
Threefish	256, 512, 1024	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB
TripleDES	112, 168	OpenPGPCFB ECB, CBC, CFB8, CFB64, OFB, CTR
Twofish	129,192,256	ECB, CBC, CCM, CFB8, CFB128, CTR, GCM, OFB

In the mode of operation table above, it was mentioned that certain modes require an *Initialisation Vector* (IV). Recall that an IV is a string of bits that is used to produce a unique cipher text when the same encryption key is used. The IV should be a randomly generated set of bits and length of the IV is usually the same as that of the block size of the underling encryption engine. Note however, that the IV is not a secret and needs to be known to both parties – the sender and receiver of the encrypted data. Therefore in the BC C# API, the IV and the secret key are stored within the same object. The example below demonstrates how to generate a key, an IV and how these are stored.

Example 8 – Key and Initialisation Vector Generation

```
public static ParametersWithIV keyParameterGenerationWithIV(int keySize, byte[] IV)
{
    CipherKeyGenerator keyGen = new CipherKeyGenerator();
    keyGen.Init(new KeyGenerationParameters(ExValues.cSharpFixedRandom, keySize));
    KeyParameter keyParam = keyGen.GenerateKeyParameter();
    ParametersWithIV keyParamWithIV = new ParametersWithIV(keyParam, IV);

    return keyParamWithIV;
}
```

Example 9 – CBC Mode Encryption/Decryption

```
public static byte[] cbcPaddedEncrypt(ICipherParameters keyParamWithIV, byte[] plainTextData)
{
    IBlockCipher symmetricBlockCipher = new AesEngine();
    IBlockCipherMode symmetricBlockMode = new CbcBlockCipher(symmetricBlockCipher);
    IBlockCipherPadding padding  = new Pkcs7Padding();

    PaddedBufferedBlockCipher cbcCipher =
        new PaddedBufferedBlockCipher(symmetricBlockMode, padding);

    cbcCipher.Init(true, keyParamWithIV);
    int blockSize = cbcCipher.GetBlockSize();
    byte[] cipherTextData = new byte[cbcCipher.GetOutputSize(plainTextData.Length)];
    int processLength =
        cbcCipher.ProcessBytes(plainTextData, 0, plainTextData.Length, cipherTextData, 0);
    int finalLength = cbcCipher.DoFinal(cipherTextData, processLength);
    byte[] finalCipherTextData = new byte[cipherTextData.Length - (blockSize - finalLength)];
    Array.Copy(cipherTextData,0, finalCipherTextData,0, finalCipherTextData.Length);

    return finalCipherTextData;
}

public static byte[] cbcPaddedDecrypt(ICipherParameters keyParamWithIV, byte[] cipherTextData)
{
    IBlockCipher symmetricBlockCipher = new AesEngine();
    IBlockCipherMode symmetricBlockMode = new CbcBlockCipher(symmetricBlockCipher);
    IBlockCipherPadding padding  = new Pkcs7Padding();

    PaddedBufferedBlockCipher cbcCipher =
        new PaddedBufferedBlockCipher(symmetricBlockMode, padding);

    cbcCipher.Init(false, keyParamWithIV);
    int blockSize = cbcCipher.GetBlockSize();
    byte[]  plainTextData = new byte[cbcCipher.GetOutputSize(cipherTextData.Length)];
    int processLength =
        cbcCipher.ProcessBytes(cipherTextData, 0, cipherTextData.Length, plainTextData, 0);
    int finalLength = cbcCipher.DoFinal(plainTextData, processLength);
    byte[] finalPlainTextData = new byte[plainTextData.Length - (blockSize - finalLength)];
    Array.Copy(plainTextData,0, finalPlainTextData,0, finalPlainTextData.Length);

    return finalPlainTextData;
}
```

As Examples 7 and 9 show, ciphers which are operated in block mode require padding of some sort. BC C# API provides the following paddings:

- ***ZERO*** – simply adds a string of zero bytes to fill in block,
- ***PKCS7 (PKCS5)*** – the value of each byte is the total number of bytes that are added,
- ***ISO10126-2*** – last padding byte is the number of padding bytes used, the other bytes can be random,
- ***X9.23*** – the last byte of the padding is the number of pad bytes, all other bytes of the padding are zeros,
- ***ISO7816-4 (ISO9797-1)*** – first padding byte is *0x80*, other padding bytes are *0s*, and
- **TBC** (Trailing Bit Complement) – if the plain text ends in a *0* bit, all the padding bits will be *1s*, else all the padding bits will be *0s*.

We mentioned that certain cipher modes can also be operated as a stream where no padding is required. The API provides a couple of ways in which a block cipher can be transformed into a stream cipher. The first example simply uses the API **`StreamBlockCipher`** class.

Example 10 – CFB Stream Mode Encryption/Decryption

```
public static byte[] cfbByteStreamEncrypt(ICipherParameters keyParamWithIV, byte[] plainTextData)
{
    IBlockCipher symmetricBlockCipher = new IdeaEngine();

    // Next select the mode compatible with the "engine", in this case we
    // use CFB mode as a streaming cipher - set the block size to 1 byte
    IBlockCipherMode symmetricBlockMode = new CfbBlockCipher(symmetricBlockCipher, 8);

    StreamBlockCipher cfbCipher = new StreamBlockCipher(symmetricBlockMode);
    cfbCipher.Init(true, keyParamWithIV);
    byte[] cipherTextData = new byte[plainTextData.Length];

    // simulate stream
    for (int j = 0; j < plainTextData.Length; j++)
    {
        cipherTextData[j] = cfbCipher.ReturnByte(plainTextData[j]);
    }

    return cipherTextData;
}

public static byte[] cfbByteStreamDecrypt(ICipherParameters keyParamWithIV, byte[] cipherTextData)
{
    IBlockCipher symmetricBlockCipher = new IdeaEngine();

    // Next select the mode compatible with the "engine", in this case we
    // use CFB mode as a streaming cipher - set the block size to 1 byte
    IBlockCipherMode symmetricBlockMode = new CfbBlockCipher(symmetricBlockCipher, 8);

    StreamBlockCipher cfbCipher = new StreamBlockCipher(symmetricBlockMode);
    cfbCipher.Init(false, keyParamWithIV);
    byte[] plainTextData = new byte[cipherTextData.Length];

    // simulate stream
    for (int j = 0; j < plainTextData.Length; j++)
    {
        plainTextData[j] = cfbCipher.ReturnByte(cipherTextData[j]);
    }

    return plainTextData;
}
```

The next example uses the CTR mode. It is also different from CFB mode above, in that the generated cipher stream is made by encrypting the nonce (or IV) and counter. This means that the IV generated must leave sufficient room for a counter. In example below, observe that we have initialized the cipher with an IV less than the block size of the cipher.. The implementation of this mode operates on having the last ***m*** bytes of the IV as the counter. In the example we have allocated ***28*** bytes to the IV (which in reality should be random) and ***4*** bytes to the counter. Thus in this example the maximum length of a message we could encrypt is 2^{32} blocks (or 2^{37} bytes – since we are using the Threefish cipher engine with a block size of 256 bits). The BC C# API will throw an exception if the IV is greater than the cipher block size – note however, that if the IV is the same size as the cipher block size, then we essentially have the CFB mode again. Finally, in addition, with this example, we will use the mode without padding but still access the cipher/plain text as a buffer.

Example 11 – CTR Mode Without Padding Encryption/Decryption

```
public static byte[] ctrNoPaddingEncrypt(ICipherParameters keyParamWithIV, byte[] plainTextData)
{
    IBlockCipher symmetricBlockCipher = new ThreefishEngine(256);
    IBlockCipherMode symmetricBlockMode = new KCtrBlockCipher(symmetricBlockCipher);
    BufferedBlockCipher ctrCipher = new BufferedBlockCipher(symmetricBlockMode);

    ctrCipher.Init(true, keyParamWithIV);
    int blockSize = ctrCipher.GetBlockSize();
    byte[]  cipherTextData = new byte[ctrCipher.GetOutputSize(plainTextData.Length)];
    int processLength =
        ctrCipher.ProcessBytes(plainTextData, 0, plainTextData.Length, cipherTextData, 0);

    int finalLength = ctrCipher.DoFinal(cipherTextData, processLength);
    byte[] finalCipherTextData = new byte[cipherTextData.Length - (blockSize - finalLength)];
    Array.Copy(cipherTextData,0, finalCipherTextData,0 ,finalCipherTextData.Length);

    return cipherTextData;
}

public static byte[] ctrNoPaddingDecrypt(ICipherParameters keyParamWithIV, byte[] cipherTextData)
{
    IBlockCipher symmetricBlockCipher = new ThreefishEngine(256);
    IBlockCipherMode symmetricBlockMode = new KCtrBlockCipher(symmetricBlockCipher);
    BufferedBlockCipher ctrCipher = new BufferedBlockCipher(symmetricBlockMode);

    ctrCipher.Init(false, keyParamWithIV);
    int blockSize = ctrCipher.GetBlockSize();
    byte[]  plainTextData = new byte[ctrCipher.GetOutputSize(cipherTextData.Length)];
    int processLength =
        ctrCipher.ProcessBytes(cipherTextData, 0, cipherTextData.Length, plainTextData, 0);

    int finalLength = ctrCipher.DoFinal(plainTextData, processLength);
    byte[] finalPlainTextData = new byte[plainTextData.Length - (blockSize - finalLength)];
    Array.Copy(plainTextData,0, finalPlainTextData,0, finalPlainTextData.Length);

    return plainTextData;
    }
```

The final block cipher example we present is that of an ***Authenticated Encryption with Associated Data*** (AEAD) mode. Both the GCM (Galois/Counter Mode) and CCM (Counter with Cipher block chaining message authentication Mode or counter with CBC-MAC) can be used with various cipher engines. Essentially in this mode along with an Initialisation Vector, we add some plain text data which can be used to "authenticate" the encrypted data which has been received. The plain text data is (essentially) woven into the cipher and a tag is appended to the encrypted data which is to provide this authentication.

It should be noted that apart from CCM the GCM modes, the EAX mode can also be used for authentication. EAX mode can be used as a stream, while both CCM and GCM can only be used in block mode. Finally, it is up your requirements to choose an appropriate length of the tag size (or mac size as used in the BC C# API) – usually the size of the underlying cipher block size is desirable. (However, 4, 6, 8, 10, 12, 14 or 16 bytes can be used.)

Example 12 – CCM AEAD Mode Without Padding Encryption/Decryption

```
public static byte[] ccmAEADNoPaddingEncrypt(KeyParameter keyParam, byte[] plainTextData)
{
    IBlockCipher symmetricBlockCipher = new AesEngine();
    int macSize = 8*symmetricBlockCipher.GetBlockSize();
    byte[] nonce = new byte[12];
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
    Array.Copy(ExValues.sampleIVnonce, nonce, nonce.Length);
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize, nonce, associatedText);

    CcmBlockCipher cipherMode = new CcmBlockCipher(symmetricBlockCipher);
    cipherMode.Init(true,keyParamAead);
    cipherMode.ProcessBytes(plainTextData, 0, plainTextData.Length, null, 0);

     int outputSize = cipherMode.GetOutputSize(0);
     byte[]  cipherTextData = new byte[outputSize];
     cipherMode.DoFinal(cipherTextData, 0);

     return cipherTextData;
}

public static byte[] ccmAEADNoPaddingDecrypt(KeyParameter keyParam, byte[] cipherTextData)
{
    IBlockCipher symmetricBlockCipher = new AesEngine();
    int macSize = 8*symmetricBlockCipher.GetBlockSize();
    byte[] nonce = new byte[12];
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
    Array.Copy(ExValues.sampleIVnonce, nonce, nonce.Length);
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize, nonce, associatedText);

    CcmBlockCipher cipherMode = new CcmBlockCipher(symmetricBlockCipher);
    cipherMode.Init(false,keyParamAead);
    cipherMode.ProcessBytes(cipherTextData, 0, cipherTextData.Length, null, 0);

    int outputSize = cipherMode.GetOutputSize(0);
    byte[]  plainTextData = new byte[outputSize];
    cipherMode.DoFinal(plainTextData , 0);

    return plainTextData;
}
```

Finally before we move onto dedicated stream ciphers, lets summarise the the BC C# API for block ciphers in this section:

- Block ciphers comprise of an engine and mode.
- Some modes require padding and others can be used as a stream cipher.
- Some modes require an initialisation vector (or nonce) and some modes can have associated data for useful for authentication.

3.2 Stream Ciphers

The table below provides a list of the most common stream cipher "engines" which are available in the BC C# API.

Engine Algorithm	Key Size	Notes
ChaCha	128, 256	Certified eSTREAM[5]. Based on Salsa20 and therefore requires a 64 bit initialisation vector.
HC128	128	Certified eSTREAM. Requires a 128 bit initialisation vector.
HC256	256	Requires 256 bit initialisation vector.
Salsa20	128, 256	Certified eSTREAM. Requires a 64 bit initialisation vector.
XSalsa20	256	Requires a 192 bit initialisation vector.

As previously mentioned, there are no modes which we need to consider when we are using one of the stream ciphers in the table above. The examples below demonstrate the usage of the API for these cipher. In the example below we have used a 128 bit key, however a 256 bit key can used as the table above indicates.

Example 13 – ChaCha Stream Cipher Encryption/Decryption

```
public static byte[] chachaStreamEncrypt(ICipherParameters keyParamWithIV, byte[] plainTextData)
{
    IStreamCipher cipher = new ChaChaEngine();
    cipher.Init(true, keyParamWithIV);

    byte[] cipherTextData = new byte[plainTextData.Length];

    // simulate stream
    for (int j = 0; j < plainTextData.Length; j++)
    {
        cipherTextData[j] = cipher.ReturnByte(plainTextData[j]);
    }

    return cipherTextData;
}
```

Notice below that there is no difference between the encryption method and the decryption method – since the cipher engine is effectively using XOR to encrypt and decrypt.

5 European Union cryptographic validation certification program.

```
public static byte[] chachaStreamDecrypt(ICipherParameters keyParamWithIV, byte[] cipherTextData)
{
    IStreamCipher cipher = new ChaChaEngine();
    cipher.Init(false, keyParamWithIV);

    byte[] plainTextData = new byte[cipherTextData .Length];

    // simulate stream
    for (int j = 0; j < cipherTextData.Length; j++)
    {
        plainTextData[j] = cipher.ReturnByte(cipherTextData[j]);
    }

    return plainTextData;
}
```

Note that the Salsa20 implementation is essentially that of the ChaCha cipher – so no example is required. In the next example we will use the stream cipher HC128.

Example 14 – HC128 Stream Cipher Encryption/Decryption

```
public static byte[] h128StreamEncrypt(ICipherParameters keyParamWithIV, byte[] plainTextData)
{
    IStreamCipher cipher = new ChaChaEngine();
    cipher.Init(true, keyParamWithIV);

    byte[] cipherTextData = new byte[plainTextData.Length];

    // simulate stream
    for (int j = 0; j < plainTextData.Length; j++)
    {
        cipherTextData[j] = cipher.ReturnByte(plainTextData[j]);
    }

    return cipherTextData;
}

public static byte[] h128StreamDecrypt(ICipherParameters keyParamWithIV, byte[] cipherTextData)
{
    IStreamCipher cipher = new ChaChaEngine();
    cipher.Init(false, keyParamWithIV);

    byte[] plainTextData = new byte[cipherTextData .Length];

    // simulate stream
    for (int j = 0; j < cipherTextData.Length; j++)
    {
        plainTextData[j] = cipher.ReturnByte(cipherTextData[j]);
    }

    return plainTextData;
}
```

3.3 AEAD Ciphers

In this section we look at two ciphers which are especially designed for AEAD. Recall that an AEAD cipher is composed of two parts: ***AE*** which stands for Authenticated Encryption – this part is used to achieve protection against message data modification and message data injection during the transport of the cipher text. However, this alone does not stop the so called “reply” attacks. The

AD part stands for Associated Data – which is designed to minimise replay type attacks. Together we have a ***AEAD*** cipher which provides authentication as well as additional support for other cryptographic attacks.

One of the most common block cipher mode for AEAD is Galois Counter Mode (GCM) for AEAD. In basic terms GCM combines CTR mode for encryption and polynomial over a finite field (say 2^n a so called Galois field) for the generation of the MAC.

Example 15 – GCM AEAD Mode Encryption/Decryption

```
public static byte[] gcmAEADEncrypt(KeyParameter keyParam, byte[] plainTextData)
{
    IBlockCipher cipher = new AesEngine();
    int macSize = 8*cipher.GetBlockSize();
    byte[] nonce = ExValues.sampleIVnonce;
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize, nonce, associatedText);

    GcmBlockCipher cipherMode = new GcmBlockCipher(cipher);
    cipherMode.Init(true,keyParamAead);

    int outputSize = cipherMode.GetOutputSize(plainTextData.Length);
    byte[]  cipherTextData = new byte[outputSize];
    int result = cipherMode.ProcessBytes(plainTextData, 0, plainTextData.Length, cipherTextData, 0);
    cipherMode.DoFinal(cipherTextData, result);

    return cipherTextData;
}


public static byte[] gcmAEADDecrypt(KeyParameter keyParam, byte[] cipherTextData)
{
    IBlockCipher cipher = new AesEngine();
    int macSize = 8*cipher.GetBlockSize();
    byte[] nonce = ExValues.sampleIVnonce;
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize, nonce, associatedText);

    GcmBlockCipher cipherMode = new GcmBlockCipher(cipher);
    cipherMode.Init(false,keyParamAead);

    int outputSize = cipherMode.GetOutputSize(cipherTextData.Length);
    byte[]  plainTextData = new byte[outputSize];
    int result = cipherMode.ProcessBytes(cipherTextData, 0, cipherTextData.Length,
                                         plainTextData, 0);
    cipherMode.DoFinal(plainTextData, result);

    return plainTextData;
}
```

The BC C# API also provides dedicated ciphers for AEAD. The two we consider are ASCON and ChaCha20Poly1305. Note that ASCON has recently been selected as a standard for lightweight cryptography in the NIST Lightweight Cryptography competition. The ChaCha20Poly1305 is essentially the stream cipher we considered in the Section 3.2 above, combined with the Poly1305 family of polynomial hash functions.

The inputs to the ASCON cipher are:

- A key of maximum size 128 bits. A key of 128 bits is recommended however.

- An IV (or nonce) which is also 128 bits.
- Your Associated Data.

The MAC (or Tag) size depends on the key size and the API provides a method to determine this as shown in the example below.

Example 16 – ASCON AEAD Encryption/Decryption

```
public static byte[] asconAEADEncrypt(KeyParameter keyParam, byte[] plainTextData)
{
    AsconEngine cipher = new AsconEngine(AsconEngine.AsconParameters.ascon128a);
    int macSize = 8*cipher.GetKeyBytesSize();
    byte[] nonce = new byte[cipher.GetIVBytesSize()];
    Array.Copy(ExValues.sampleIVnonce, nonce, cipher.GetIVBytesSize());
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize,nonce, associatedText);

    cipher.Init(true,keyParamAead);
    int outputSize = cipher.GetOutputSize(plainTextData.Length);
    byte[]  cipherTextData = new byte[outputSize];
    int result = cipher.ProcessBytes(plainTextData, 0, plainTextData.Length, cipherTextData, 0);
    cipher.DoFinal(cipherTextData,result);

    return cipherTextData;
}

public static byte[] asconAEADDecrypt(KeyParameter keyParam, byte[] cipherTextData)
{
    AsconEngine cipher = new AsconEngine(AsconEngine.AsconParameters.ascon128a);
    int macSize = 8*cipher.GetKeyBytesSize();
    byte[] nonce = new byte[cipher.GetIVBytesSize()];
    Array.Copy(ExValues.sampleIVnonce, nonce, cipher.GetIVBytesSize());
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize,nonce, associatedText);

    cipher.Init(false,keyParamAead);
    int outputSize = cipher.GetOutputSize(cipherTextData.Length);
    byte[]  plainTextData = new byte[outputSize];
    int result = cipher.ProcessBytes(cipherTextData, 0, cipherTextData.Length, plainTextData, 0);
    cipher.DoFinal(plainTextData,result);

    return plainTextData;
}
```

The inputs to ChaCha20Poly1305 are:

- A 256 bit key.
- An IV (or nonce) whose size is a fixed 96 bits (the remaining 32 bits are used as a counter).
- Your Associated Data.

The MAC (or Tag) size is fixed at 128 bits.

Example 17 – ChaCha20Poly1305 AEAD Encryption/Decryption

```
public static byte[] chacha20poly1305AEADEncrypt(KeyParameter keyParam, byte[] plainTextData)
{
    ChaCha20Poly1305 cipher = new ChaCha20Poly1305();
    int macSize = 128;
    byte[] nonce = new byte[12];
    Array.Copy(ExValues.sampleIVnonce, nonce, 12);
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
```

```
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize,nonce, associatedText);

    cipher.Init(true,keyParamAead);

    int outputSize = cipher.GetOutputSize(plainTextData.Length);
    byte[]  cipherTextData = new byte[outputSize];
    int result = cipher.ProcessBytes(plainTextData, 0, plainTextData.Length, cipherTextData, 0);
    cipher.DoFinal(cipherTextData,result);

    return cipherTextData;
}

public static byte[] chacha20poly1305AEADDecrypt(KeyParameter keyParam, byte[] cipherTextData)
{
    ChaCha20Poly1305 cipher = new ChaCha20Poly1305();
    int macSize = 128;
    byte[] nonce = new byte[12];
    Array.Copy(ExValues.sampleIVnonce, nonce, 12);
    byte[] associatedText = ExValues.additionalAuthenticatedDataA;
    AeadParameters keyParamAead = new AeadParameters(keyParam, macSize,nonce, associatedText);

    cipher.Init(false,keyParamAead);

    int outputSize = cipher.GetOutputSize(cipherTextData.Length);
    byte[]  plainTextData = new byte[outputSize];
    int result = cipher.ProcessBytes(cipherTextData, 0 , cipherTextData.Length, plainTextData, 0);
    cipher.DoFinal(plainTextData,result);

    return plainTextData;
}
```

3.4 Format Preserving Encryption using AES

In this final section of symmetric ciphers we describe a family of Format Preserving Encryption (***FPE***) algorithms. FPE is an encryption method which attempts to preserve the format of the plaintext. Such encryption is used mainly for credit card numbers and other identifiable data associated with a person. FPE has three different modes of operation: FF1, FF2, and FF3. Note that FF2 was not an approved mode for FPE by NIST and as such is not part of the BC C# API. Further, FF3 was found to have a flaw on the level of security and as such FF3 has been replaced with FF3-1. Both FF1 and FF3-1 (FF3_1) are implemented and available in the API.

FPE works on an input alphabet and produces encrypted strings in the same alphabet. From the algorithm's point of view the alphabet is just a set of indexes, starting at zero and going up to the number of characters in the alphabet. The size of the alphabet is referred to as the radix. The second thing which is needed is a tweak value, which is used to improve the security of the algorithm. The tweak value does not need to be kept secret. Also note that in the example below we have used a tweak whose size is 56 bits – this is the current fixed length as recommended in the NIST SP800-38r1 document and is the only supported tweak length available. (Appendix C in the same NIST document provides a nice summary on the use of tweaks.)

As mentioned above, A FPE cipher engine requires an alphabet. In our case an alphabet is simply an array of characters. Clearly the plaintext which you wish to encrypt must consist of characters obtained from your alphabet. Note that the BC C# API will produced an exception if there are characters within your plaintext which are not in the alphabet.

Example 18 – FF3-1 FPE Mode Encryption/Decryption

```
public static char[] ff3_1FPEEncrypt(KeyParameter keyParam, char[] alphabet, char[] plainTextData)
{
    // Create a mapper from our alphabet to indexs
    IAlphabetMapper alphabetMapper = new BasicAlphabetMapper(alphabet);

    // Create FpeParameter object
    byte[] tweak = ExValues.sampleTweak;
    FpeParameters fpeKeyParam = new FpeParameters(keyParam, alphabetMapper.Radix, tweak);

    IBlockCipher cipher = new AesEngine();
    FpeFf3_1Engine cipherMode = new FpeFf3_1Engine(cipher);
    cipherMode.Init(true,fpeKeyParam);

    byte[] cipherTextData = new byte[plainTextData.Length];
    byte[] convertedPlainTextData = alphabetMapper.ConvertToIndexes(plainTextData);

    int result = cipherMode.ProcessBlock(convertedPlainTextData, 0,
                    convertedPlainTextData.Length, cipherTextData, 0);

    char[] convertedCipherTextData = alphabetMapper.ConvertToChars(cipherTextData);

    return convertedCipherTextData;
}

public static char[] ff3_1FPEDecrypt(KeyParameter keyParam, char[] alphabet, char[] cipherTextData)
{
    IAlphabetMapper alphabetMapper = new BasicAlphabetMapper(alphabet);

    byte[] tweak = ExValues.sampleTweak;
    FpeParameters fpeKeyParam = new FpeParameters(keyParam, alphabetMapper.Radix, tweak);

    IBlockCipher cipher = new AesEngine();
    FpeFf3_1Engine cipherMode = new FpeFf3_1Engine(cipher);
    cipherMode.Init(false,fpeKeyParam);

    byte[] plainTextData = new byte[cipherTextData.Length];
    byte[] convertedCipherTextData = alphabetMapper.ConvertToIndexes(cipherTextData);

    int result = cipherMode.ProcessBlock(convertedCipherTextData, 0,
                    convertedCipherTextData.Length, plainTextData, 0);

    char[] convertedPlainTextData = alphabetMapper.ConvertToChars(plainTextData);

    return convertedPlainTextData;
}
```

4 Key Agreement and Exchange Algorithms

4.1 What Happended To the Asymmetric Ciphers?

Recall that asymmetric ciphers (or public key encryption) consists of a key pair say K_{pub} and K_{pri}, where K_{pri} represents the private part of the key pair and K_{pub} represents the publicly available part. Note that, it is not given a priori which of K_{pub} or K_{pri} are to be used to encrypt a plaintext message. This is only one of the issues associated with asymmetric ciphers used for encryption of large blocks of data – there are others as we explain below. Let K_e denote the key which will be used for encryption (that is, K_e could be either K_{pub} or K_{pri}) and let K_d denote the decryption key. In very simple terms, given the plain text $\boldsymbol{m}$ and the encryption, decryption functions $E_{K\,e}$ and D_{K_d}, respectively, then we obtain $\boldsymbol{c}{=}E_{K_e}(\boldsymbol{m})$ the ciphertext and $\boldsymbol{m}{=}D_{K_d}(\boldsymbol{c})$. Here we observe another issue, that while $\boldsymbol{m} = D_{K_d}(E_{K_e}(\boldsymbol{m}))$, in general we do not have:

$$\boldsymbol{m} = E_{K_e}(D_{K_d}(\boldsymbol{m}))$$ for a plaintext message $\boldsymbol{m}$.

(This property holds for RSA asymmetric ciphers, however it need not hold for encryption based on elliptic curves.) Other issues arise due to the fact that if K_{pub} is used for encryption, then this provides privacy but not authenticity – whereas if K_{pri} is used for encryption, then authenticity is guaranteed, but not privacy (this is the basis of digital signatures). However, the biggest issues when using asymmetric ciphers for encryption of large amounts of data is that:

- The underlying mathematics in such cryptographic systems means that the plain text data must be mapped to size of the key. Elliptic curves, for example do not have an underlying encryption scheme and an additional bijective function must be added.
- Additionally, asymmetric ciphers are significantly slower than symmetric ciphers – for example, RSA encryption (which could in theory be used in ECB mode) will execute 1000s of times slower than when using AES in the same mode.

The limitations in the use of asymmetric ciphers are overcome by the use of hybrid encryption schemes. Hybrid encryption, usually, consists of using an asymmetric key pair to generate and exchange a symmetric key which can then be used to efficiently encrypt data over an unsecured channel. Normally such symmetric keys are short lived (ephemeral).

Below are some terms which will be useful in the examples which follow:

- ***Key Exchange*** – is a method, process or protocol which allows two (or sometimes more) parties to agree on a secret key, enabling these parties to send and receive cryptographically secure data.
- ***Key Agreement*** – is a *chosen* Key Exchange method to share the generated secret key. For example, one of the most common is the Diffie–Hellman key agreement.
- ***Key Encapsulation Mechanisms (KEM)*** – is a specific Key Agreement using asymmetric encryption, where (in simple terms) the public key is used to encrypt the symmetric key and the private key is used to decrypt the symmetric key for use in data exchange.
- ***Key Derivation Function (KDF)*** – is a function (or algorithm) which takes as input some not necessarily cryptographically random strings and outputs cryptographically strong secret keys.

- ***Key-encryption key (KEK)*** – generate a (symmetric) key using a KDF.
- ***Pseudorandom function (PRF)*** – according to NIST, a PRF is a function that can be used to generate output from a random seed and a data variable, such that the output is computationally indistinguishable from truly random output. Actually KDFs and PRFs are closely related in that a KDF will make use of a PRF.

Having defined the terms we can provide some examples of key agreements.

4.2 Diffie-Hellman Key Agreement

The Diffie-Hellman (DH) Key agreement algorithm requires the generation of the ***domain parameters*** set $\{p, q, g\}$, where p is a (sufficiently) large prime, q is either $p-1$ or a large prime divisor of $p-1$ and g is a generator for the cyclic subgroup of order q of the multiplicative group of the finite field $\mathbb{Z}_p^*$. The set $\{p, q, g\}$ is public information. These domain parameters for DH are very (CPU time) expensive to generate – depending on your hardware and size of p (or alternatively the size of q), this could take from several minutes to several hours!

Note that there are various restrictions on the values of p which are required for cryptographic security. Such cryptographic primes are referred to as *strong primes*. There is a subset of strong primes known as ***safe primes***. These safe primes are of the form q prime and $p = 2q + 1$. (Primes of this form are known as Sophie Germain primes.) A list of pre-defined these safe primes are given RFC7919 and RFC3526.

The example below demonstrates how the BC C# API can be used generate a pair of safe primes.

Example 19 – Generating DH Finite Field Parameters

```
public static DHParameters generateDHParameters(int size)
{
    DHParametersGenerator pGen = new DHParametersGenerator();
    pGen.Init(size, 10, ExValues.cSharpFixedRandom);
    return pGen.GenerateParameters();
}
```

The parameters passed to the method above

`pGen.Init(int size, int certainty, SecureRandom random)` are:

size – this is the bit size of the prime p, choose a small value for testing, 512 bits will suffice.
certainty – is (related to) the number of iterations of the Miller-Rabin test for testing primality. In the paper, *Rabin, Michael O. (1980), "Probabilistic algorithm for testing primality"*, it was shown that the after k iterations, the probability that it would fail the test for primality is at least $\left(1 - \frac{1}{4^k}\right)$. Thus after k iterations if the test did not fail, one could be confident that the integer n is prime.
random – this allows us to generate a random integer q and test it for primality.

Having provided the example above, the next statement is surprising: ***"Don't generate your own field parameters"***. The BC C# API provides are list of pre-defined safe primes which can be used directly. The example below provides an method of how an asymmetric key pair can be generated easily with the provided safe primes and with a security strength of approximately 275 bits. Please read the appropriate sections in the RFC7919 and RFC3526 documentation to determine the security strength you require.

Example 20 – Diffie-Hellman Key Pair Generation

```
public static AsymmetricCipherKeyPair generateDHKeyPair(DHParameters dhParams)
{
    DHParameters dhParams = DHStandardGroups.rfc7919_ffdhe3072;

    DHKeyGenerationParameters dhKeyGenParams =
       new DHKeyGenerationParameters(ExValues.cSharpFixedRandom, dhParams);
    DHKeyPairGenerator dhKeyPairGen = new DHKeyPairGenerator();

    dhKeyPairGen.Init(dhKeyGenParams);
    AsymmetricCipherKeyPair dhKeyPair = dhKeyPairGen.GenerateKeyPair();

    return dhKeyPair;
}
```

All safe primes listed in RFC7919 and RFC3526 are provided in the class `DHStandardGroups`.

In the simplest of cases, the DH Key agreement consists of two parties exchanging public keys. Party A generates their key pair $\{A_{pri}, A_{pub}\}$ and party B generates theirs $\{B_{pri}, B_{pub}\}$. Recall that A_{pri} is a random element of $\mathbb{Z}_p^*$ (or $GF(p)$), a say, and that A_{pub} is g^a (similarly for party B). The basic key exchange is then simply:

$$\text{Party A generates } \left(g^b\right)^a$$
$$\text{and that}$$
$$\text{Part B generates } \left(g^a\right)^b.$$

Example 21 below shows how to organise a basic DH key agreement. Note that, this type of exchange is not secure if you are re-using the same DH key pairs and in fact note that there is no authentication involved.

Example 21 – Basic DH Key Agreement

```
public static BigInteger partyABasicAgreement(DHPrivateKeyParameters privateKeyPartyA,
                                              DHPublicKeyParameters publicKeyPartyB)
{
    DHBasicAgreement keyAgreement = new DHBasicAgreement();
    keyAgreement.Init(privateKeyPartyA);
    BigInteger secret = keyAgreement.CalculateAgreement(publicKeyPartyB);

    return secret;
}

public static BigInteger partyBBasicAgreement(DHPrivateKeyParameters privateKeyPartyB,
                                              DHPublicKeyParameters publicKeyPartyA)
{
    DHBasicAgreement keyAgreement = new DHBasicAgreement();
    keyAgreement.Init(privateKeyPartyB);
    BigInteger secret = keyAgreement.CalculateAgreement(publicKeyPartyA);

    return secret;
}
```

A better or more secure method of using the DH key agreement is with the Matsumoto, Takashima, Imai MTI/A0 key agreement scheme. (Note that there are variations of this scheme MTI/B0, MTI/C0 and MTI/C1 – however the BC C# API only provides the MTI/A0 variation.) Why is this key agreement scheme better? Essentially it is a two phase type of the basic key agreement as given in Example 21 above. The main security it offers is that the first exchange of the public keys should

be authenticated in some manner. After that the second and subsequent exchanges are made using a randomly generated asymmetric key pair. (Note that, MTI and a related protocol MQV are used for authentication and are usually certificate based – to establish the "owner" of the public keys. The example below provides an outline on generating the secret agreement only and does not address the authenticity.)

Example 22 – MTI/A0 DH Key Agreement

```
public static BigInteger[] MTIA0Agreement(DHParameters dhParams)
{
    AsymmetricCipherKeyPair dhStaticKeyPairPartyA = generateDHKeyPair(dhParams);
    AsymmetricCipherKeyPair dhStaticKeyPairPartyB = generateDHKeyPair(dhParams);

    // Party A Generates an ephemeral key pair sending it to Party B
    DHAgreement keyAgreementPartyA = new DHAgreement();
    keyAgreementPartyA.Init((DHPrivateKeyParameters)dhStaticKeyPairPartyA.Private);
    BigInteger dhEphemeralPublicKeyA = keyAgreementPartyA.CalculateMessage();

    // Party B Generates an ephemeral key pair sending it to Party A
    DHAgreement keyAgreementPartyB = new DHAgreement();
    keyAgreementPartyB.Init((DHPrivateKeyParameters)dhStaticKeyPairPartyB.Private);
    BigInteger dhEphemeralPublicKeyB = keyAgreementPartyB.CalculateMessage();

    BigInteger secretPartyA = keyAgreementPartyA.CalculateAgreement(
            (DHPublicKeyParameters)dhStaticKeyPairPartyB.Public, dhEphemeralPublicKeyB);

    BigInteger secretPartyB = keyAgreementPartyB.CalculateAgreement(
            (DHPublicKeyParameters)dhStaticKeyPairPartyA.Public, dhEphemeralPublicKeyA);

    return  new BigInteger[] {secretPartyA, secretPartyB};
}
```

4.3 Elliptic Curve Diffie-Hellman Key Agreement

The Diffie-Hellman protocol can also be used with Elliptic Curves. The points (x, y) on an elliptic curve defined over a finite field form an abelian group. If this group has prime order, then it is a cyclic group and all points (x, y) satisfying the equation of the curve lie in the group with x, y in the field – in this case we say it has co-factor **1**. (Recall that a non-trivial finite group G has no proper subgroups except the trivial groups if and only if its order is prime.) If the elliptic curve group has order $h \cdot q$ where q is a large prime, then we apply the Diffie-Hellman protocol to the subgroup of order q – in this case we say the curve has a co-factor of $\boldsymbol{h}$. Moreover, not every point that satisfies the equation of the curve lies in the subgroup. However, any such point multiplied by h will lie in the subgroup. Thus in this case where the co-factor is not 1, the Diffie-Hellman protocol is slightly more complicated. A formulation based on incorporating the curve's co-factor is actually the one described by NIST SP 800-56A which also details the use of Elliptic Curves with Diffie-Hellman as well as the finite field method. The variant incorporating the curve's co-factor is known as Elliptic Curve Co-factor Diffie-Hellman or ECCDH for short.

Elliptic curves (used for cryptography) come in various flavours. The main two types of interest are

- elliptic curves over a finite prime field $\mathbb{Z}_p^*$ (or written as $GF(p)$) where $p > 3$ is a prime, and
- elliptic curves over a binary finite field $GF(2^m)$.

For curves over a prime field, the form of the elliptic curve used is $y^2 \equiv x^3 + ax + b(\mod p)$, where $a, b \in GF(p)$ and $4a^3 + 27b^2 \neq 0$. The domain parameters for such curves are given as

$$\boldsymbol{D = \{p, h, n, Type, a, b, G\}},$$

where $\boldsymbol{p}$ is the prime used for generating $GF(p)$, $\boldsymbol{h}$ is the co-factor, $\boldsymbol{n}$ is the order of the cyclic subgroup used (so that the order of the cyclic group generated by the elliptic curve is $\boldsymbol{h \cdot n}$), $\boldsymbol{Type}$ is a description of the curve, for example "Weierstrass curve", $\boldsymbol{a}$ and $\boldsymbol{b}$ are the coefficients defining the curve and $\boldsymbol{G}$ is a point on the curve which generates all points in the cyclic subgroup. There is an optional parameter $\boldsymbol{\{Seed, c\}}$ a random number which provides a seed for generating your own coefficients $\boldsymbol{a}, \boldsymbol{b}$ and a value $\boldsymbol{c}$ to ensure that y^2 in the above curve has a unique solution. This means that defining your own elliptic curve is task intensive. The example below demonstrates how to construct a set of domain parameters from scratch for the Curve-ID: brainpoolP160r1. We have taken the parameters from RFC5639.

Example 23 – Generating EC $GF(p)$ Parameters Directly

```
public static ECDomainParameters ecDomainParametersDirect()
{
    // Generating EC brainpoolP160r1 parameters
    BigInteger p = new BigInteger(1,Hex.DecodeStrict("E95E4A5F737059DC60DFC7AD95B3D8139515620F"));
    BigInteger a = new BigInteger(1,Hex.DecodeStrict("340E7BE2A280EB74E2BE61BADA745D97E8F7C300"));
    BigInteger b = new BigInteger(1,Hex.DecodeStrict("1E589A8595423412134FAA2DBDEC95C8D8675E58"));
    BigInteger n = new BigInteger(1,Hex.DecodeStrict("E95E4A5F737059DC60DF5991D45029409E60FC09"));
    BigInteger h = BigInteger.One;

    BigInteger generatorX =
        new BigInteger(1,Hex.DecodeStrict("BED5AF16EA3F6A4F62938C4631EB5AF7BDBCDBC3"));
    BigInteger generatorY =
        new BigInteger(1,Hex.DecodeStrict("1667CB477A1A8EC338F94741669C976316DA6321"));

    ECCurve curve = new FpCurve(p, a, b, n, h);
    ECPoint generatorPoint = curve.CreatePoint(generatorX, generatorY);

    return new ECDomainParameters(curve, generatorPoint, n, h);
}
```

Unless required you would not normally generate directly your own elliptic curve domain parameters. (In fact, you should never attempt to define your own elliptic curves for cryptographic use – there are many pitfalls and security considerations – where possible always use pre-defined curves and pre-defined curve parameters.) The BC C# API provides a list of known elliptic curves for cryptgraphic use. Appendix A.1 lists all elliptic curves over a finite prime field $\mathbb{Z}_p^*$. The example below demonstrates how using the internal ID of the curve we can generate the same set of domain parameters as that given in the example above.

Example 24 – Generating Built In EC $GF(p)$ Parameters

```
public static ECDomainParameters ecBuiltInDomainParameters()
{
    X9ECParameters ecParams = ECNamedCurveTable.GetByName("brainpoolp160r1");
    return new ECDomainParameters(
        ecParams.Curve, ecParams.G, ecParams.N, ecParams.H, ecParams.GetSeed() );
}
```

Elliptic curves over a binary field have the Weierstrass form $y^2 + xy = x^3 + ax^2 + b \pmod{2^m}$, where $a, b \in GF(2^m)$ and $b \neq 0$. The field $GF(2^m)$ is viewed as a vector space of polynomials over the field $GF(2)$. Thus an arbitrary element $\alpha \in GF(2^m)$ can be written as

$$\alpha = \alpha_{m-1}x^{m-1} + \alpha_{m-2}x^{m-2} + \cdots + \alpha_1 x + \alpha_0,$$

with $\alpha_j \in \{0, 1\}$. When performing arithmetic on the elliptic curve over $GF(2^m)$, then we must reduce the values obtained from the calculation to fit back into $GF(2^m)$. To do this we take the result of a calculation modulo an irreducible polynomial. For efficiency, two types of irreducible polynomials are used: Trinomials of the form $x^m + x^k + 1$, and Pentanomials (if a trinomial does not exist) of the form $x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$. The domain parameters for such binary curves are given as

$$\boldsymbol{D = \{m, h, n, Type, a, b, G, f(x)\}},$$

where $\boldsymbol{m}$ is the integer in the field $GF(2^m)$, $\boldsymbol{h}$ is the co-factor, $\boldsymbol{n}$ is the order of the cyclic subgroup used (so that the order of the cyclic group generated by the elliptic curve is $\boldsymbol{h \cdot n}$), $\boldsymbol{Type}$ is a description of the curve, for example "Weierstrass curve", $\boldsymbol{a}$ and $\boldsymbol{b}$ are the coefficients defining the curve and $\boldsymbol{G}$ is a point on the curve which generates all points in the cyclic subgroup, $\boldsymbol{f(x)}$ is the trinomial or pentanomial for the arithmetic reductions. Note that, the point $\boldsymbol{G}$ can come in two different "flavours": either it is given in the *polynomial basis* (the usual basis of a vector space in elementary Linear Algebra) or in a *normal basis* (which requires a root of unity of $GF(2^m)$) which makes certain arithmetic operations on the elliptic curve group easier – in particular the squaring operation. In this document the generating point $\boldsymbol{G}$ for elliptic curves over $GF(2^m)$ are given in the polynomial basis. (In fact the BC C# API currently does not support normal bases.)

The example below demonstrates how to construct a set of domain parameters from scratch for the Curve-ID: sect283k1, which is a 283-bit binary field Weierstrass curve (also known as K-283 curve or the ansit283k1 curve).

Example 25 – Generating Binary EC Parameters Directly

```
public static ECDomainParameters ecBinaryDomainParametersDirect()
{
    //Generating Binary EC sect283k1
    int m  = 283;
    int k1 = 5;
    int k2 = 7;
    int k3 = 12;
    BigInteger a = BigInteger.Zero;
    BigInteger b = BigInteger.One;
    BigInteger h = BigInteger.Four;

    BigInteger n =
       new BigInteger("1ffffffffffffffffffffffffffffffffffe9ae2ed07577265dff7f94451e061e163c61",16);

    BigInteger generatorX =
       new BigInteger("503213f78ca44883f1a3b8162f188e553cd265f23c1567a16876913b0c2ac2458492836",16);
    BigInteger generatorY =
       new BigInteger("1ccda380f1c9e318d90f95d07e5426fe87e45c0e8184698e45962364e34116177dd2259",16);

    ECCurve curve = new F2mCurve(m, k1, k2, k3, a, b, n, h);
    ECPoint generatorPoint = curve.CreatePoint(generatorX, generatorY);

    return new ECDomainParameters(curve, generatorPoint, n, h);
```

```
}
```

As with the comments made for curves over a prime field, you would not normally generate directly your own elliptic curve domain parameters. The BC C# API provides a list of known binary elliptic curves for cryptographic use. Appendix A.2 lists all elliptic curves over $GF(2^m)$ which are available in the BC C# API. The example below demonstrates how using the internal ID of the curve we can generate the same set of domain parameters as that given in the example above.

Example 26 – Generating Built In Binary EC Parameters

```
public static ECDomainParameters ecBuiltInBinaryDomainParameters()
{
    X9ECParameters ecParams = ECNamedCurveTable.GetByName("K-283");

    return new ECDomainParameters(
       ecParams.Curve, ecParams.G, ecParams.N, ecParams.H, ecParams.GetSeed());
}
```

This chapter was discussing key agreements, therefore, once we have our elliptic curve parameters we are ready to generate a pair of asymmetric keys. Generating keys and key agreement is similar to that described above for DH as the following examples demonstrate.

Example 27 – EC Diffie-Hellman Key Pair Generation

```
public static AsymmetricCipherKeyPair generateECDHKeyPair(ECDomainParameters ecParams)
{
    ECKeyGenerationParameters ecKeyGenParams =
       new ECKeyGenerationParameters(ecParams, ExValues.cSharpFixedRandom);
    ECKeyPairGenerator ecKeyPairGen = new ECKeyPairGenerator();
    ecKeyPairGen.Init(ecKeyGenParams);
    AsymmetricCipherKeyPair ecKeyPair = ecKeyPairGen.GenerateKeyPair();

    return ecKeyPair;
}
```

Example 28 – EC Diffie-Hellman Key Agreement

```
public static BigInteger partyABasicAgreement(ECPrivateKeyParameters privateKeyPartyA,
                                              ECPublicKeyParameters publicKeyPartyB)
{
    ECDHCBasicAgreement keyAgreement = new ECDHCBasicAgreement();
    keyAgreement.Init(privateKeyPartyA);

    BigInteger secret = keyAgreement.CalculateAgreement(publicKeyPartyB);

    return secret;
}

public static BigInteger partyBBasicAgreement(ECPrivateKeyParameters privateKeyPartyB,
                                              ECPublicKeyParameters publicKeyPartyA)
{
    ECDHCBasicAgreement keyAgreementParty = new ECDHCBasicAgreement();
    keyAgreementParty.Init(privateKeyPartyB);

    BigInteger secret = keyAgreementParty.CalculateAgreement(publicKeyPartyA);

    return secret;
}
```

In Examples 27 and 28 above we have used the sect283k1 elliptic curve which has a co-factor $h = 4$. Therefore, we need to call the object `ECDHCBasicAgreement` which will perform the necessary arithmetic taking the co-factor into account. If the co-factor $h = 1$, then we would replace the object `ECDHCBasicAgreement` with the object `ECDHBasicAgreement`.

5 Key Encapsulation and Key Wrapping

Key Wrapping in its most general definition is the process of encrypting ("wrapping") a critical key K_c say, with another K_w say.

As given in Section 4.1, *Key Encapsulation* is generally regarded as a specific Key Agreement using asymmetric encryption, where the public key is used to encrypt the symmetric key and the private key is used to decrypt the symmetric key for use in data exchange. Thus we differentiate between Key Wrapping and Key Encapsulation. (Note that, some authors do not differentiate between Key Wrapping – in our sense – and Key Encapsulation. For such authors, Key Encapsulation is another form of Key Wrapping.)

Key Wrapping and Key Encapsulation is a common method which is used for protecting a given key during transportation over an unsecured channel and is also common when storing a key, for example in a database. Since Key Wrapping and Key Encapsulation is a common occurrence it is not surprising that certain standards and best practices have evolved in achieving this. NIST provides standards for Key Wrapping with the use of symmetric key algorithms such as AES (see *NIST Special Publication 800-38F*) and asymmetric RSA key pairs – although in the case of RSA, the NIST terminology used, describes the wrapping function in connection with key transport. (Actually NIST, which provides various cryptographic standards, provides Key Encapsulation details under the heading of Key Transport –see *NIST Special Publication 800-56B Revision 2*.)

The main security concept to keep in mind with key wrapping is that it is fool hardy to wrap a key which is expected to have a particular security strength with one that does not at least have the equivalent security strength. Using a wrapping key with higher security strength is clearly better, however, this may not always be possible. Most of the standards used in the BC C# API come from the NIST documents (indicated above) and RFC 3394 and RFC5649 (which allows symmetric key wrapping with a different padding scheme to that of RFC 3394).

To get a feeling on how key wrapping is used: consider transporting a key over a channel (whether secured or not). The wrapping key K_w would be kept in an HSM (High Security Module) and the wrapping and unwrapping of the key K_c would occur within the HSM. This provides an example of *Key Management* – which is outside the scope of this document. The point being made, however, is that the K_w needs to be as secure (or better) than the key (or keys) you are wrapping and further the K_w needs to be protected.

5.1 Key Wrapping Using Symmetric Keys

The common standard using symmetric keys is a family referred to as AES-KW. This refers to using AES as the block cipher (with a key length of 128, 192 or 256 bits). The output generated is provided in blocks of 64 bits. In the simplest case, without any padding, we have the example below which is based upon RFC 3394 with the default initialisation vector $IV = A6A6A6A6A6A6A6A6$. The input is the K_w (the AES wrapping key or KEK given as `keyParam` in the example below); the data (or key) to be wrapped in blocks of 64 bits (given as `unwrappedData` in the example below); and the output is the encrypted data (or encrypted/wrapped key). Similarly, for unwrapping the wrapped data, we input the original K_w along with the previously wrapped data.

Example 29 – RFC3394 Key Wrapping Default IV

```
public static byte[] wrapKeyRFC3394(ICipherParameters keyParam, byte[] unwrappedData)
{
    IBlockCipher symmetricBlockCipher = new AesEngine();
    Rfc3394WrapEngine wrapEngine = new Rfc3394WrapEngine(symmetricBlockCipher);
    wrapEngine.Init(true, keyParam);

    return wrapEngine.Wrap(unwrappedData,0,unwrappedData.Length);
}

public static byte[] unwrapKeyRFC3394(ICipherParameters keyParam, byte[] wrappedData)
{
    IBlockCipher symmetricBlockCipher = new AesEngine();
    Rfc3394WrapEngine wrapEngine = new Rfc3394WrapEngine(symmetricBlockCipher);
    wrapEngine.Init(false, keyParam);

    return wrapEngine.Unwrap(wrappedData,0,wrappedData.Length);
}
```

The RFC 3394 standard also describes a variation where we can input our own IV. The example below demonstrates how you could accomplish this.

Example 30 – RFC3394 Key Wrapping With IV

```
public static byte[] wrapKeyRFC3394WithIV(ICipherParameters keyParam,
                                          byte[] IV, byte[] unwrappedData)
{
    ParametersWithIV keyParamWithIV = new ParametersWithIV(keyParam, IV);

    IBlockCipher symmetricBlockCipher = new AesEngine();
    Rfc3394WrapEngine wrapEngine = new Rfc3394WrapEngine(symmetricBlockCipher);
    wrapEngine.Init(true, keyParamWithIV);

    return wrapEngine.Wrap(unwrappedData,0,unwrappedData.Length);
}
```

```
public static byte[] unwrapKeyRFC3394WithIV(ICipherParameters keyParam,
                                            byte[] IV, byte[] wrappedData)
{
    ParametersWithIV keyParamWithIV = new ParametersWithIV(keyParam, IV);

    IBlockCipher symmetricBlockCipher = new AesEngine();
    Rfc3394WrapEngine wrapEngine = new Rfc3394WrapEngine(symmetricBlockCipher);
    wrapEngine.Init(false, keyParamWithIV);

    return wrapEngine.Unwrap(wrappedData,0,wrappedData.Length);
}
```

Appendix A – Built in Curves

Appendix A.1 – Prime Field Built in Curves

Name	IDs	OID
ANSSI FRP 256v1	FRP256v1	1.2.250.1.223.101.256.1
ECC Brainpool P160r1	brainpoolp160r1	1.3.36.3.3.2.8.1.1.1
ECC Brainpool P160t1	brainpoolp160t1	1.3.36.3.3.2.8.1.1.2
ECC Brainpool P192r1	brainpoolp192r1	1.3.36.3.3.2.8.1.1.3
ECC Brainpool P192t1	brainpoolp192t1	1.3.36.3.3.2.8.1.1.4
ECC Brainpool P224r1	brainpoolp224r1	1.3.36.3.3.2.8.1.1.5
ECC Brainpool P224t1	brainpoolp224t1	1.3.36.3.3.2.8.1.1.6
ECC Brainpool P256r1	brainpoolp256r1	1.3.36.3.3.2.8.1.1.7
ECC Brainpool P256t1	brainpoolp256t1	1.3.36.3.3.2.8.1.1.8
ECC Brainpool P320r1	brainpoolp320r1	1.3.36.3.3.2.8.1.1.9
ECC Brainpool P320t1	brainpoolp320t1	1.3.36.3.3.2.8.1.1.10
ECC Brainpool P384r1	brainpoolp384r1	1.3.36.3.3.2.8.1.1.11
ECC Brainpool P384t1	brainpoolp384t1	1.3.36.3.3.2.8.1.1.12
ECC Brainpool P512r1	brainpoolp512r1	1.3.36.3.3.2.8.1.1.13
ECC Brainpool P512t1	brainpoolp512t1	1.3.36.3.3.2.8.1.1.14
GOST RFC4357	GostR3410-2001-CryptoPro-A	
GOST RFC4357	GostR3410-2001-CryptoPro-B	
GOST RFC4357	GostR3410-2001-CryptoPro-C	
GOST RFC5832	tc26-gost-3410-2012-256-paramSetA	
GOST RFC7836	tc26-gost-3410-12-512-paramSetA	
GOST RFC7836	tc26-gost-3410-12-512-paramSetB	
GOST RFC5832	tc26_gost_3410_12_512_paramSetC	
NIST P-192	P-192	1.2.840.10045.3.1.1
NIST P-224	P-224	1.3.132.0.33
NIST P-256	P-256	1.2.840.10045.3.1.7
NIST P-384	P-384	1.3.132.0.34
NIST P-521	P-521	1.3.132.0.35
SEC secp112r1	secp112r1	1.3.132.0.6

Name	IDs	OID
SEC secp112r2	secp112r2	1.3.132.0.7
SEC secp128r1	secp128r1	1.3.132.0.28
SEC secp128r2	secp128r2	1.3.132.0.29
SEC secp160k1	secp160k1	1.3.132.0.9
SEC secp160r1	secp160r1	1.3.132.0.8
SEC secp160r2	secp160r2	1.3.132.0.30
SEC secp192k1	secp192k1	1.3.132.0.31
SEC secp192r1	secp192r1	1.2.840.10045.3.1.1
SEC secp224k1	secp224k1	1.3.132.0.32
SEC secp224r1	secp224r1	1.3.132.0.33
SEC secp256k1	secp256k1	1.3.132.0.10
SEC secp256r1	secp256r1	1.2.840.10045.3.1.7
SEC secp384r1	secp384r1	1.3.132.0.34
SEC secp521r1	secp521r1	1.3.132.0.35
SM2	sm2p256v1	1.2.156.10197.1.301
X9.62 prime192v1	prime192v1	1.2.840.10045.3.1.1
X9.62 prime192v2	prime192v2	1.2.840.10045.3.1.2
X9.62 prime192v3	prime192v3	1.2.840.10045.3.1.3
X9.62 prime239v1	prime239v1	1.2.840.10045.3.1.4
X9.62 prime239v2	prime239v2	1.2.840.10045.3.1.5
X9.62 prime239v3	prime239v3	1.2.840.10045.3.1.6
X9.62 prime256v1	prime256v1	1.2.840.10045.3.1.7

Appendix A.2 – Binary Field Built in Curves

Name	IDs	OID
NIST B-163	B-163	1.3.132.0.15
NIST B-233	B-233	1.3.132.0.27
NIST B-283	B-283	1.3.132.0.17
NIST B-409	B-409	1.3.132.0.37
NIST B-571	B-571	1.3.132.0.39
NIST K-163	K-163	1.3.132.0.1
NIST K-233	K-233	1.3.132.0.26
NIST K-283	K-283	1.3.132.0.16

Name	IDs	OID
NIST K-409	K-409	1.3.132.0.36
NIST K-571	K-571	1.3.132.0.38
SEC sect113r1	sect113r1	1.3.132.0.4
SEC sect113r2	sect113r2	1.3.132.0.5
SEC sect131r1	sect131r1	1.3.132.0.22
SEC sect131r2	sect131r2	1.3.132.0.23
SEC sect163k1	sect163k1	1.3.132.0.1
SEC sect163r1	sect163r1	1.3.132.0.2
SEC sect163r2	sect163r2	1.3.132.0.15
SEC sect193r1	sect193r1	1.3.132.0.24
SEC sect193r2	sect193r2	1.3.132.0.25
SEC sect233k1	sect233k1	1.3.132.0.26
SEC sect233r1	sect233r1	1.3.132.0.27
SEC sect239k1	sect239k1	1.3.132.0.3
SEC sect283k1	sect283k1	1.3.132.0.16
SEC sect409k1	sect409k1	1.3.132.0.36
SEC sect409r1	sect409r1	1.3.132.0.37
SEC sect571k1	sect571k1	1.3.132.0.38
SEC sect571r1	sect571r1	1.3.132.0.39
X9.62 c2pnb163v1	c2pnb163v1	1.2.840.10045.3.0.1
X9.62 c2pnb163v2	c2pnb163v2	1.2.840.10045.3.0.2
X9.62 c2pnb163v3	c2pnb163v3	1.2.840.10045.3.0.3
X9.62 c2pnb176w1	c2pnb176w1	1.2.840.10045.3.0.4
X9.62 c2tnb191v1	c2tnb191v1	1.2.840.10045.3.0.5
X9.62 c2tnb191v2	c2tnb191v2	1.2.840.10045.3.0.6
X9.62 c2tnb191v3	c2tnb191v3	1.2.840.10045.3.0.7
X9.62 c2pnb208w1	c2pnb208w1	1.2.840.10045.3.0.10
X9.62 c2tnb239v1	c2tnb239v1	1.2.840.10045.3.0.11
X9.62 c2tnb239v2	c2tnb239v2	1.2.840.10045.3.0.12
X9.62 c2tnb239v3	c2tnb239v3	1.2.840.10045.3.0.13
X9.62 c2pnb272w1	c2pnb272w1	1.2.840.10045.3.0.16
X9.62 c2pnb304w1	c2pnb304w1	1.2.840.10045.3.0.17
X9.62 c2tnb359v1	c2tnb359v1	1.2.840.10045.3.0.18
X9.62 c2pnb368w1	c2pnb368w1	1.2.840.10045.3.0.19
X9.62 c2tnb431r1	c2tnb431r1	1.2.840.10045.3.0.20

Appendix C – References

MORE TO FOLLOW