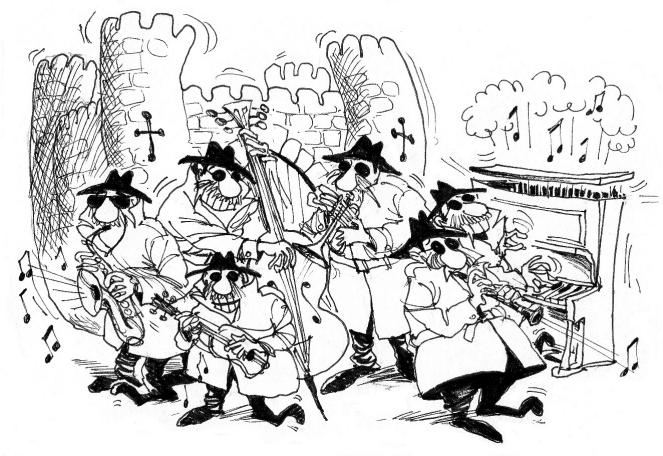
Legion of the Bouncy Castle Inc.
BC-FNA (Bouncy Castle FIPS .NET API)

User Guide

Version: 1.0.2 Date: 07/16/21



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.

Validation testing was performed by Acumen Security. For more information on validation or revalidations of the software contact:

Josh Kolstad
Laboratory Manager,
Acumen Security
18504 Office Park Dr
Montgomery Village, MD 20886
United States of America

joshua.kolstad@intertek.com

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 FIPS .NET module. It is a companion document to the “Legion of the Bouncy Castle Security Policy”. As an addendum to the security policy this document is meant to provide more detail on the module. It should be noted that the security policy is the authoritative source, and in event of a conflict between this document and the security policy, the version in the security policy should be accepted. Only the security policy has been reviewed and approved by the Cryptographic Module Validation Program (CMVP), the joint US – Canadian program for the validation of cryptographic products that overviews the FIPS process.

2 Installation and Startup

2.1 Installation

At the time of writing the bc-fips-1.0.2.dll, and the associated documentation xml if required, simply needs to be copied into the dependencies for the the project using it.

2.2 Startup

The module will execute its startup procedure once `CryptoStatus.IsReady()` is called. This will happen either by calling `CryptoStatus.IsReady()` directly or by indirectly calling `CryptoStatus.IsReady()` as a result of trying to access one of the module's services.

2.3 Approved Only Mode

Unless the module is otherwise configured, threads will start in general mode and have access to both approved and non-approved algorithms.

A thread in general mode can move into approved-only mode by calling:

```
CryptoServicesRegistrar.SetApprovedOnlyMode(true);
```

Once the call is executed the thread will have access only to NIST approved algorithms and will no longer have access to any CSPs created before it entered approved-only mode. Note also that one the call is executed the thread will also be unable to exit approved-only mode without exiting completely.

Unless otherwise stopped, threads in approved-only mode may share CSPs between each other, however they cannot share CSPs with threads that are not in approved-only mode.

3 Cipher Algorithms (Symmetric)

Cipher availability and the modes of the ciphers available depends on whether the module is running approved-only mode or not. Not all widely used cipher modes are recognised by FIPS so in the FIPS ciphers appear in both the approved mode and general mode table, as some extra modes of operation become available for FIPS ciphers if the module in not running in approved only mode.

3.1 Available in Approved Mode Operation

Only the FIPS recognised ciphers and modes are available in approved mode of operation.

Algorithm	Class	Modes
AES	FipsAes	ECB, CBC, CFB8, CFB128, OFB, CTR, CCM, GCM, FF1
TripleDES	FipsTripleDes	ECB, CBC, CFB8, CFB64, OFB, CTR

3.2 Available in General Operation

In general operation there are a wider range of ciphers available, in addition support is also provided for ciphers to use non-FIPS modes such as OpenPGPCFB.

Algorithm	Class	Modes
ARC4	Arc4	N/A
AES	Aes	OpenPGPCFB, FF3-1
Camellia	Camellia	CBC, CCM, CFB8, CFB128, CTR, ECB, GCM, OCB, OFB, OpenPGPCFB
ChaCha	ChaCha	eSTREAM, IETF (RFC 7539)
SEED	Seed	CBC, CCM, CFB8, CFB128, CTR, ECB, GCM, OFB
Serpent	Serpent	CBC, CCM, CFB8, CFB128, CTR, ECB, GCM, OFB
TripleDES	TripleDes	OpenPGPCFB

3.3 Paddings Available

FIPS is largely ambivalent towards padding mechanisms, so all the padding modes are available in both approved-mode and general operation.

Mode	Padding Type
ECB	NONE, PKCS7 (PKCS5), ISO10126-2, X9.23, ISO7816-4 (ISO9797-1), TBC
CBC	NONE, PKCS7 (PKCS5), ISO10126-2, X9.23, ISO7816-4 (ISO9797-1), TBC, CS1, CS2, CS3

While the padding types named after standards are largely self explanatory, TBC is “trailing bit

complement” as defined in Appendix A, NIST SP 800-38A, CS1, CS2, and CS3 are all cipher text stealing modes as defined in the Addendum to NIST SP 800-38A. CBC with CS3 is also equivalent to CTS mode in RFC 2040. The cipher text stealing modes are used a little differently in the API and have their own parameter generators. This is because they require use of both the cipher and the last two blocks of input to it.

3.3 Examples

Examples 3.3.1 and 3.3.2 can be applied to any block cipher, example 3.3.3 is applicable to any 16 bit block cipher, and example 3.3.4 can be applied to any stream cipher.

3.3.1 AES Encryption using CBC and PKCS5/7Padding

```
// create an AES service around a key
FipsAes.Key key = new FipsAes.Key(
                            Hex.Decode("aafd12f659cae63489b479e5076ddec2"));
IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

// define IV
byte[] iv = Hex.Decode("aafd12f659cae63489b479e5076ddec2");

// define input
byte[] input = Hex.Decode("000102030405060708090a0b0c0d0e0f”
                            + “ff0102030405060708090a0b0c0d0e0f");

// define an output stream to capture the encrypted data
MemoryOutputStream bOut = new MemoryOutputStream();

// set up the encryptor
IBlockCipherBuilder<IParameters<Algorithm>> encryptorBldr =
              provider.CreateBlockEncryptorBuilder(FipsAes.Cbc.WithIV(iv));

ICipher encryptor = encryptorBldr.BuildPaddedCipher(bOut, new Pkcs7Padding());

// encrypt the input
using (Stream encOut = encryptor.Stream)
{
    encOut.Write(input, 0, input.Length);
}

byte[] cipherText = bOut.ToArray()));

// decrypt the resulting cipher text
IBlockCipherBuilder<IParameters<Algorithm>> decryptorBldr =
              provider.CreateBlockDecryptorBuilder(FipsAes.Cbc.WithIV(iv));

ICipher decryptor = decryptorBldr.BuildPaddedCipher(
                        new MemoryInputStream(cipherText), new Pkcs7Padding());

byte[] plainText = null;

using (Stream decIn = decryptor.Stream)
{
    plainText = Streams.ReadAll(decIn);
}

// plainText will now refer to data which is the same as the original input.
```

3.3.2 AES Encryption using CTR mode

```
// create an AES service around a key
FipsAes.Key key = new FipsAes.Key(
                          Hex.Decode("aafd12f659cae63489b479e5076ddec2"));
IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

// define IV – note 15 bytes, so max message length 255 * 16 bytes
byte[] iv = Hex.Decode("000102030405060708090a0b0c0d0e");

// define input
byte[] input = Hex.Decode("000102030405060708090a0b0c0d0e0f”
                          + “ff0102030405060708090a0b0c0d0e0f");

// encrypt the input
MemoryOutputStream bOut = new MemoryOutputStream();

ICipherBuilder<IParameters<Algorithm>> encryptorBldr =
              provider.CreateEncryptorBuilder(FipsAes.Ctr.WithIV(iv));

ICipher encryptor = encryptorBldr.BuildCipher(bOut);

using (Stream encOut = encryptor.Stream)
{
    encOut.Write(input, 0, input.Length);
}

byte[] cipherText = bOut.ToArray()));

// decrypt the resulting cipher text
ICipherBuilder<IParameters<Algorithm>> decryptorBldr =
              provider.CreateDecryptorBuilder(FipsAes.Ctr.WithIV(iv));

ICipher decryptor = decryptorBldr.BuildCipher(
                        new MemoryInputStream(cipherText));

byte[] plainText = null;

using (Stream decIn = decryptor.Stream)
{
    plainText = Streams.ReadAll(decIn);
}

// plainText will now refer to data which is the same as the original input.
```

3.3.3 AES using GCM and an AEADParameterSpec

If you have to use JDK 1.5 or 1.6 there is no API in the Cipher class for introducing associated data. The AEADParameterSpec is provided to allow associated data to be prepended to the cipher text. In JDK 1.7, or later, Cipher.updateAAD() and the GCMParameterSpec can be used instead.

```
// associated data
byte[] aData = Hex.Decode("5627d7c5d9cbdcdedbe28c496e60a31a64779a327d753”
                        + ”ccdedcfb77546540f630d7656dc662b3b113156535dff466”
                        + “d189c1c2a17ef19de1257b9b1aba86b23693677f8ef91792”
                        + “a4e7e68bbadc058c9dc2153eb935c9ac6d2f3a0");

// plain text data
byte[] data = Hex.Decode("29da07d47becd5968b776ffd0a8f9fddd212dfabb”
```

```
            + "4f24636a2d9ccc4a68e46d9907ce55aef02c3a70fc7951d1f6b680fa5dd3c");

// generat a 128 bit AES key
FipsAes.Key key = CryptoServicesRegistrar.CreateGenerator(
                        FipsAes.KeyGen128, new SecureRandom()).GenerateKey();
IAeadCipherService provider = CryptoServicesRegistrar.CreateService(key);


// create a builder for the AEAD cipher with a tag size of 128 bits.
IAeadCipherBuilder<IParameters<FipsAlgorithm>> aeadEncryptorBldr =
               provider.CreateAeadEncryptorBuilder(
                           FipsAes.Gcm.WithIV(nonce).WithMacSize(128));

MemoryOutputStream bOut = new MemoryOutputStream();


// build the encryptor
IAeadCipher encryptor = aeadEncryptorBldr.BuildCipher(
                                                AeadUsage.AAD_FIRST, bOut);

// write out the associated data
encryptor.AadStream.Write(aData, 0, aData.Length);


// write out the plain text
using (Stream encOut = encryptor.Stream)
{
      encOut.Write(data, 0, data.Length);
}

byte[] output = bOut.ToArray();             // encrypted output with in-line tag
byte[] encMac = encryptor.GetMac().Collect();  // tag value by itself
```

3.3.4 Stream Encryption using ChaCha

This is an example of using the stream cipher ChaCha, as defined in RFC 7539 to do a basic encryption. Note: ChaCha is not available in approved-only mode.

```
byte[] key = Hex.Decode("000102030405060708090a0b0c0d0e0f"
                     + "101112131415161718191a1b1c1d1e1f");
byte[] nonce = Hex.Decode("000000000000004a00000000");


MemoryOutputStream bOut = new MemoryOutputStream();

// define the cipher with a new ChaCha key
IStreamCipherService provider = CryptoServicesRegistrar.CreateService(
                                          new ChaCha.Key(key));


ICipher chacha = provider.CreateEncryptorBuilder(
                             ChaCha.Ietf.WithIV(nonce)).BuildCipher(bOut);

// define input
byte[] input = Hex.Decode("000102030405060708090a0b0c0d0e0f"
                        + "ff000102030405060708090a0b0c0d0e0f");
```

```
// write out the plain text
using (Stream encOut = chacha.Stream)
{
      encOut.Write(input, 0, input.Length);
}

byte[] output = bOut.ToArray();             // encrypted output
```

3.3.5 FF3-1 Format Preserving Encryption using AES

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.

The first code fragment shows the encryption step. We're using a `BasicAlphabetMapper` class to map our alphabet (ascii digits) to the indexes that the FPE engine will actually use.

```
byte[] key = …
byte[] tweak = … // must be 8 bytes for FF3-1
String input = “99994444888833331111”;
String digits = "0123456789";

// Create a mapper from our alphabet to indexs
IAlphabetMapper alphabetMapper = new BasicAlphabetMapper(digits);

IAeadBlockCipherService aesService =
            CryptoServicesRegistrar.CreateService(new FipsAes.Key(key));

ICipherBuilder<FipsAes.FpeParameters> fpeEncEngine = aesService
      .CreateEncryptorBuilder(
            FipsAes.Ff3_1.WithRadix(alphabetMapper.Radix).WithTweak(tweak));


// convert our input to indexes for FPE
byte[] bytes = alphabetMapper.ConvertToIndexes(input.ToCharArray());
byte[] encryptedBytes = process(fpeEncEngine, bytes);

// map the encrypted array of index bytes back to our character space
String encryptedInput = new String(
                  alphabetMapper.ConvertToChars(encryptedBytes)));
```

The process method is like any other stream cipher – while the FPE algorithm is built on a block cipher it returns the same number of bytes of output as you give it as input. The following process() method just writes to a `MemoryStream` and returns the resulting byte array. Keep in mind that indexes go in and indexes come out – this is why we need to use the alphabetMapper the second time to get our encrypted bytes back into the right format.

```
private byte[] process(
    ICipherBuilder<FipsAes.FpeParameters> fpeEngine, byte[] bytes)
{
    MemoryStream stream = new MemoryStream();

    ICipher cipher = fpeEngine.BuildCipher(stream);

    cipher.Stream.Write(bytes, 0, bytes.Length);
    cipher.Stream.Close();

    return stream.ToArray();
}
```

Finally decryption is the same process but done using a decryptor which reverses the previous processing, but can use the same process() method.

```
ICipherBuilder<FipsAes.FpeParameters> fpeDecEngine =
              aesService.CreateDecryptorBuilder(FipsAes.Ff3_1
                           .WithRadix(alphabetMapper.Radix).WithTweak(tweak));

// convert our input to indexes for FPE
byte[] bytes = alphabetMapper.ConvertToIndexes(encryptedInput.ToCharArray());
byte[] decryptedBytes = process(fpeDecEngine, bytes);

// map the decrypted array of index bytes back to our character space
String recoveredInput = new String(
                  alphabetMapper.ConvertToChars(decryptedBytes)));
```

4 Key Agreement and Exchange Algorithms

Key agreement algorithms are available based around both elliptic curve and finite field. The post-quantum algorithm NewHope is provided for doing key exchange in non-fips mode. The basic primitives are provided in the low-level classes together with support for the use of a digest, a PR, or a KDF as detailed in NIST SP 800-56C.

Key confirmation is not supported directly by the API although a MAC and symmetric key can easily be derived in a single step using the KDF or PRF methods on the FipsEC.Cdh parameter object.

4.1 Available in Approved Mode Operation

Key Agreement Method	Class	Parameter Configuration
DH	FipsDH	FipsDH.DH
ECCDH	FipsEC	FipsEC.Cdh

4.2 Available in General Operation

Key Agreement Method	Class	Parameter Configuration
NewHope	NewHope	NewHope.Sha3_256

4.3 Examples

The following examples are all for elliptic curve. 4.3.2 and 4.3.3 also show how to introduce a key material generator.

4.3.1 Basic Agreement with Cofactor

Agreement based simply around the agreed value.

```
public byte[] BasicAgreementWithCofactor(
    AsymmetricECPrivateKey ourKey,
    AsymmetricECPublicKey otherPartyKey)
{
    IAgreementCalculatorService dhFact =
            CryptoServicesRegistrar.CreateService(ourKey);

    IAgreementCalculator<FipsEC.AgreementParameters> agreement =
                               dhFact.CreateAgreementCalculator(FipsEC.Cdh);

    return agreement.Calculate(otherPartyKey);
}
```

4.3.2 Basic Agreement with PRF

Agreement using HMAC SHA-256 as the PRF function to process the agreed value.

```
public byte[] AgreementWithCofactorAndPrf(
    AsymmetricECPrivateKey ourKey,
    AsymmetricECPublicKey otherPartyKey,
    byte[] salt)
{
```

```
    IAgreementCalculatorService dhFact =
            CryptoServicesRegistrar.CreateService(ourKey);

    IAgreementCalculator<FipsEC.AgreementParameters> agreement =
            dhFact.CreateAgreementCalculator(
              FipsEC.Cdh.WithKeyMaterialGenerator(
                        new FipsPrfKmg(FipsPrfAlgorithm.Sha256HMac, salt));

    return agreement.Calculate(otherPartyKey);
}
```

4.3.3 Basic Agreement with KDF

Agreement based on using the X9.63 KDF.

```
public byte[] AgreementWithCofactorAndKdf(
    AsymmetricECPrivateKey ourKey,
    AsymmetricECPublicKey otherPartyKey,
    byte[] salt)
{
    IAgreementCalculatorService dhFact =
            CryptoServicesRegistrar.CreateService(ourKey);

    IAgreementCalculator<FipsEC.AgreementParameters> agreement =
            dhFact.CreateAgreementCalculator(
              FipsEC.Cdh.WithKeyMaterialGenerator(
                            new FipsKdfKmg(FipsKdf.X963, salt, 32));

    return agreement.Calculate(otherPartyKey);
}
```

4.3.3 Key Exchange with NewHope

The post-quantum algorithm NewHope is different from a regular key agreement as only one party, the initiating party, has a public/private key pair. NewHope also uses a message digest to act as a “flattening functioning” to hide any biases in the key exchange calculation and is designed to produce an agreed 32 byte value between the initiator and the receiving party. Note that, unlike with regular Diffie-Hellman, a public/private key pair should never be reused.

A basic example follows:

```
AsymmetricKeyPair<AsymmetricNHPublicKey, AsymmetricNHPrivateKey> aliceKp =
                        CryptoServicesRegistrar.CreateGenerator(
                                        NewHope.KeyGen, new SecureRandom())
                                                .GenerateKeyPair();

AsymmetricNHPrivateKey alicePriv = aliceKp.PrivateKey;
AsymmetricNHPublicKey alicePub = aliceKp.PublicKey;

ExchangePair bobExchPair = CryptoServicesRegistrar
                        .CreateGenerator(NewHope.Sha3_256, new SecureRandom())
                                        .GenerateExchange(alicePub);

byte[] bobSharedKey = bobExchPair.GetSharedValue();
byte[] aliceSharedKey = CryptoServicesRegistrar.CreateService(alicePriv)
                                .CreateAgreementCalculator(NewHope.Sha3_256)
                                        .Calculate(bobExchPair.PublicKey);
```

In the example, aliceSharedKey and bobSharedKey are the agreed value. Note that Bob does not

have a private key, but that the GenerateExchange() method generates a public key Bob can send to Alice as part of its return value.

5 Key Derivation Functions

The standard key agreement KDFs and the TLS KDFs are supported.

5.1 Available in Approved Mode Operation

Derivation Method	Class	Parameter Configuration
TLS 1.0	FipsKdf	FipsKdf.Tls1_0
TLS 1.1	FipsKdf	FipsKdf.Tls1_1
TLS 1.2	FipsKdf	FipsKdf.Tls1_2
X 9.63	FipsKdf	FipsKdf.X963
SP800-56C Concatenation	FipsKdf	FipsKdf.Concatenation
HKDF	FipsKdf	FipsKdf.HKdf

5.2 Available in General Operation

There are no additional key derivation algorithms offered in general operation.

5.3 Examples

In addition to being used in conjunction with the FipsKdfKmg class for key material generation with the EC Diffie-Hellman algorithm, KDFs can also be invoked directly for general use or use in protocols such as TLS.

5.3.1 TLS 1.2 KDF

This example goes through the two stages of TLS 1.2 key material calculation. Undeclared variables are byte arrays.

```
IKdfCalculator<FipsKdf.TlsKdfParameters> kdfCalculator =
            CryptoServicesRegistrar.CreateService(FipsKdf.Tls1_2)
                                       .WithPrf(FipsShs.Sha256HMac)
                                       .From(pre_master_secret,
                                           FipsKdf.TlsStage.MASTER_SECRET,
                                           clientHello_random,
                                           serverHello_random));

byte[] genMaster = new byte[master_secret.Length];

kdfCalculator.GetResult(master_secret.Length).Collect(genMaster, 0);

// … server_random and client_random are exchanged.

kdfCalculator = CryptoServicesRegistrar.CreateService(FipsKdf.Tls1_2)
                                       .WithPrf(FipsShs.Sha256HMac)
                                       .From(master_secret,
                                         FipsKdf.TlsStage.KEY_EXPANSION,
                                         server_random,
                                        client_random));

byte[] genKeyBlock = new byte[key_block.Length];
```

```
kdfCalculator.GetResult (genKeyBlock.Length).Collect (genKeyBlock, 0);

// genKeyBlock now contains the final key material.
```

5.3.2 X9.63 KDF

This is the regular KDF used by Diffie-Hellman algorithms. In the following example it is also being used to generate 128 bits.

```
byte[] Z = Hex.Decode("1c7d7b5f0597b03d06a018466ed1a93e30ed4b04dc64ccdd");

byte[] output = new byte[32];

IKdfCalculator<FipsKdf.AgreementKdfParameters> kdfCalculator =
                          CryptoServicesRegistrar.CreateService(FipsKdf.X963).From(Z));

kdfCalculator.GetResult(output.Length).Collect(output, 0);
```

6 Key Wrapping Algorithms

Algorithms for key wrapping are supported for both asymmetric and symmetric keys.

6.1 Available in Approved Mode Operation

The following key wrapping techniques, based on SP800-38F, are available for use with symmetric ciphers in approved-mode.

Cipher	Algorithm	Class	Parameter Configuration
AES	KW	FipsAES	FipsAes.KW
AES	KWP	FipsAES	FipsAes.KWP
DESede	TKW	FipsTripleDES	FipsAes.TKW

The following key wrapping techniques are available using asymmetric ciphers in approved-mode.

Cipher	Algorithm	Class	Parameter Configuration
RSA	OAEP	FipsRsa	FipsRsa.WrapOaep

6.2 Available in General Operation

Cipher	Algorithm	Class	Parameter Configuration
Camellia	KW	Camellia	Camellia.KW
Camellia	KWP	Camellia	Camellia.KWP
ElGamal	OAEP	ElGamal	ElGamal.WrapOaep
ElGamal	PKCS#1.5	ElGamal	ElGamal.WrapPkcs1v15
Rsa	PKCS#1.5	Rsa	Rsa.WrapPkcs1v15
SEED	KW	Seed	Seed.KW
SEED	KWP	Seed	Seed.KWP
Serpent	KW	Serpent	Serpent.KW
Serpent	KWP	Serpent	Serpent.KWP

6.3 Examples

6.3.1 Key Wrapping using RSA

The following snippet shows an example of using RSA with OAEP to wrap a secret key in the low-level API. Note: that in both the wrapping and unwrapping process a SecureRandom is required in order to facilitate RSA blinding on the decryption.

```
public byte[] wrapKey(AsymmetricRsaPublicKey pubKey, byte[] keyData)
{
    IKeyWrapper<FipsRsa.OaepWrapParameters> keyWrapper =
            CryptoServicesRegistrar.CreateService(pubKey, new SecureRandom())
            .CreateKeyWrapper(FipsRsa.WrapOaep);

    return keyWrapper.Wrap(keyData).Collect();
```

```
}
```

The following snippet shows an example of unwrapping a wrapped secret key that has been wrapped using RSA with OAEP. Note: the SecureRandom is required for RSA blinding to protect the private key values.

```
public byte[] unwrapKey(AsymmetricRsaPrivateKey privKey, byte[] wrapData)
{
    IKeyUnwrapper<FipsRsa.OaepWrapParameters> keyUnwrapper =
            CryptoServicesRegistrar.CreateService(privKey, new SecureRandom())
                  .CreateKeyUnwrapper(FipsRsa.WrapOaep);

    return keyUnwrapper.Unwrap(wrapData, 0, wrapData.Length).Collect();
}
```

6.3.2 Key Wrapping using AES

In the following example the contents of the byte array `inputKeyBytes` will be wrapped using the KW key wrapping technique from FIPS SP800-38F.

```
byte[] inputKeyBytes = …; // bytes making up the key to be wrapped
byte[] keyBytes = …; // bytes making up AES key doing the wrapping

IAeadBlockCipherService factory =
            CryptoServicesRegistrar.CreateService(new FipsAes.Key(keyBytes));

IKeyWrapper<FipsAes.WrapParameters> wrapper =
                                          factory.CreateKeyWrapper(FipsAes.KW);

byte[] wrappedBytes = wrapper.Wrap(inputKeyBytes).Collect();
```

6.3.3 Key Wrapping using Camellia with padding using inverse function

In the below snippets it is assumed secKey is a valid Camellia key. It is also worth keeping in mind that keyToBeWrapped can also be a PublicKey or PrivateKey.

The first snippet shows a simple function which wraps a key – keyToBeWrapped.

```
public byte[] wrapKey(byte[] keyBytes, byte[] keyToBeWrapped)
{
    IAeadBlockCipherService factory =
           CryptoServicesRegistrar.CreateService(new Camellia.Key(keyBytes));

    IKeyWrapper<Camellia .WrapParameters> wrapper =
        factory.CreateKeyWrapper(Camellia.KWP.WithUsingInverseFunction(true));

    return wrapper.Wrap(keyToBeWrapped).Collect();
}
```

The second snippet shows the key unwrap process.

```
public byte[] unwrapKey(
          byte[] keyBytes, byte[] wrappedKey)
{
    IAeadBlockCipherService factory =
           CryptoServicesRegistrar.CreateService(new Camellia.Key(keyBytes));

    IKeyUnwrapper<Camellia .WrapParameters> unwrapper =
```

```
        factory.CreateKeyUnwrapper(Camellia.KWP.WithUsingInverseFunction(true));

    return unwrapper.Unwrap(wrappedKey, 0, wrappedKey.Length).Collect();
}
```

7 Mac Algorithms

A broad range of MAC and HMAC algorithms are supported by the BC FIPS provider.

7.1 Available in Approved Mode Operation

MAC Name	Class	Parameter Configuration
AES CMAC	FipsAes	FipsAes.CMac
AES GMAC	FipsAes	FipsAes.GMac
HMAC SHA-1	FipsShs	FipsShs.Sha1HMac
HMAC SHA-224	FipsShs	FipsShs.Sha224HMac
HMAC SHA-256	FipsShs	FipsShs.Sha256HMac
HMAC SHA-384	FipsShs	FipsShs.Sha384HMac
HMAC SHA-512	FipsShs	FipsShs.Sha512HMac
HMAC SHA-512(224)	FipsShs	FipsShs.Sha512_224HMac
HMAC SHA-512(256)	FipsShs	FipsShs.Sha512_256HMac
HMAC SHA3-224	FipsShs	FipsShs.Sha3_224HMac
HMAC SHA3-256	FipsShs	FipsShs.Sha3_256HMac
HMAC SHA3-384	FipsShs	FipsShs.Sha3_384HMac
HMAC SHA3-512	FipsShs	FipsShs.Sha3_512HMac
KMAC128	FipsShs	FipsShs.KMac128
KMAC256	FipsShs	FipsShs.KMac256
Triple-DES CMAC	FipsTripleDes	FipsTripleDes.CMac

If a HMAC is truncated using the WithMacSize() method on the parameter object, the right most bits are truncated as per the NIST standards.

7.2 Available in General Operation

MAC Name	Class	Parameter Configuration
Camellia CMAC	Camellia	Camellia.CMac
Camellia GMAC	Camellia	Camellia.GMac
Poly1305	Poly1305	Poly1305.Mac
SEED CMAC	Seed	Seed.CMac
SEED GMAC	Seed	Seed.GMac
Serpent CMAC	Serpent	Serpent.CMac
Serpent GMAC	Serpent	Serpent.GMac

7.3 Examples

7.3.1 AES using CMAC – 64 bit.

The following snippet shows a function returning a 64 bit CMAC MAC produced from the passed in aesKeyBytes and data. In this case the key has been specifically marked as to be used with CMAC, FipsAes.Alg could be also be used on key creation if a specific marking is not required.

```
public byte[] createMac(byte[] aesKeyBytes, byte[] data)
{
    IMacFactory<FipsAes.AuthenticationParameters> calcFact =
                CryptoServicesRegistrar.CreateService(
                            new FipsAes.Key(FipsAes.CMac, aesKeyBytes))
                .CreateMacFactory(FipsAes.CMac.WithMacSize(64));

    IStreamCalculator<IBlockResult> macCalculator = calcFact.CreateCalculator();

    Stream macStream = macCalculator.Stream;

    macStream.Write(data, 0, data.Length);

    macStream.Close();

    return calc.GetResult().Collect();
}
```

7.3.2 HMAC-SHA256

The following snippet provides a function using HMAC-SHA256 to generate a 128 bit HMAC from the passed in HMAC key bytes and data.

```
public byte[] createMac(byte[] hmacKey, byte[] data)
{
    IMacFactory<FipsShs.AuthenticationParameters> calcFact =
                CryptoServicesRegistrar.CreateService(
                            new FipsShs.Key(FipsShs.Sha256HMac, hmacKey))
                .CreateMacFactory(FipsShs.Sha256HMac.WithMacSize(128));

    IStreamCalculator<IBlockResult> macCalculator = calcFact.CreateCalculator();

    Stream macStream = macCalculator.Stream;

    macStream.Write(data, 0, data.Length);

    macStream.Close();

    return calc.GetResult().Collect();
}
```

7.3.3 Using Poly1305 with key generation

The following snippet provides an example of using Poly1305 with a randomly generated key to calculate the MAC on a byte array called data.

```
byte[] data = …

Poly1305.KeyGenerator keyGen = CryptoServicesRegistrar.CreateGenerator(
```

```
                                        Poly1305.KeyGen256, new SecureRandom());

IMacFactoryService provider = CryptoServicesRegistrar.CreateService(
                                                        keyGen.GenerateKey());

IStreamCalculator<IBlockResult> poly1305 =
                    provider.CreateMacFactory(Poly1305.Mac).CreateCalculator();

poly1305.Stream.Write(data, 0, data.Length);

poly1305.Stream.Close();

byte[] tag = poly1305.GetResult().Collect(); // the MAC value
```

8 Message Digest Algorithms

The BC FIPS module supports the full suite of NIST digests up to FIPS PUB-202 (SHA-3).

8.1 Available in Approved Mode Operation

Digest Name	Class	Parameter Configuration
SHA-1	FipsShs	FipsShs.Sha1
SHA-224	FipsShs	FipsShs.Sha224
SHA-256	FipsShs	FipsShs.Sha256
SHA-384	FipsShs	FipsShs.Sha384
SHA-512	FipsShs	FipsShs.Sha512
SHA-512(224)	FipsShs	FipsShs.Sha512_224
SHA-512(256)	FipsShs	FipsShs.Sha512_256
SHA3-224	FipsShs	FipsShs.Sha3_224
SHA3-256	FipsShs	FipsShs.Sha3_256
SHA3-384	FipsShs	FipsShs.Sha3_384
SHA3-512	FipsShs	FipsShs.Sha3_512
SHAKE128	FipsShs	FipsShs.Shake128
SHAKE256	FipsShs	FipsShs.Shake256
cSHAKE128	FipsShs	FipsShs.CShake128
cSHAKE256	FipsShs	FipsShs.CShake256
TupleHash128	FipShs	FipsShs.TupleHash128
TupleHash256	FipsShs	FipsShs.TupleHash256
ParallelHash128	FipShs	FipsShs.ParallelHash128
ParallelHash256	FipShs	FipsShs.ParallelHash256

8.2 Available in General Operation

There are currently no extra message digests available in general mode operation.

8.3 Examples

8.3.1 Use of SHA-256

The following example shows the creation and use of a digest calculator for SHA-256 using a byte array as input.

```
byte[] input = …;
IDigestFactory<FipsShs.Parameters> factory =
                        CryptoServicesRegistrar.CreateService(FipsShs.Sha256);

IStreamCalculator<IBlockResult> calculator = factory.CreateCalculator();

Stream digestStream = calculator.Stream;

digestStream.Write(input, 0, input.Length);

digestStream.Close();

byte[] digest = calculator.GetResult().Collect();
```

8.3.2 Use of SHAKE128

The following example shows the creation and use of the expandable output function SHAKE128 from the SHA3 family, again using a byte array as input and producing 100 bytes of output.

```
byte[] input = …;
IXofFactory<FipsShs.XofParameters> factory =
                        CryptoServicesRegistrar.CreateService(FipsSHS.Shake128);

IVariableStreamCalculator<IBlockResult> calculator = factory.CreateCalculator();

Stream digestStream = calculator.Stream;

digestStream.Write(input, 0, input.Length);

digestStream.Close();

IBlockResult result = calculator.GetResult(100);

byte[] output = result.Collect();
```

A feature of the two SHAKE functions is that you can “keep squeezing” and produce more output even after already requesting some, so another call like

```
byte[] extra = calculator.GetResult(100).Collect()
```

will return another 100 bytes of output.

9 Password Based Key Derivation Functions

Currently the only password based key derivation function available is the PBKDF2 function which is approved for use in FIPS mode of operation.

9.1 Available in Approved Mode Operation

Algorithm	Byte Encoding Used	Digest for PRF	Class
PBKDF2	UTF-8	SHA-1	FipsPbkd
PBKDF2	8-BIT/ASCII	SHA-1	FipsPbkd
PBKDF2	UTF-8	SHA-224	FipsPbkd
PBKDF2	UTF-8	SHA-256	FipsPbkd
PBKDF2	UTF-8	SHA-384	FipsPbkd
PBKDF2	UTF-8	SHA-512	FipsPbkd

9.2 Available in General Operation

Algorithm	Byte Encoding Used	Digest for PRF	Class
OpenSSL	8-BIT/ASCII	MD5	Pbkd
PKCS#12	PKCS#12	SHA-1	Pbkd
PKCS#12	PKCS#12	SHA-224	Pbkd
PKCS#12	PKCS#12	SHA-256	Pbkd
PKCS#12	PKCS#12	SHA-384	Pbkd
PKCS#12	PKCS#12	SHA-512	Pbkd

9.3 Examples

9.3.1 PBKDF2

The following snippet shows the generation of key material using PKCS#5's PBKDF2 function with HMAC SHA-256 being used as the pseudo-random-function underlying the key material generator and password, iterationCount, and salt providing the inputs.

```
public byte[] getSecretKey(char[] password, int iterationCount, byte[] salt,
                                                              int keySizeInBits)
{
    IPasswordBasedDeriverBuilder<FipsPbkd.Parameters> pbeDeriverBuilder =
         CryptoServicesRegistrar.CreateService(FipsPbkd.PbkdF2)
                                        .From(converter, password);

    IPasswordBasedDeriver<FipsPbkd.Parameters> pbeDeriver =
             pbeDeriverBuilder
```

```
                        .WithPrf(FipsShs.Sha256)
                        .WithSalt(Hex.Decode("0102030405060708090a0b0c0d0e0f"))
                        .WithIterationCount(1024)
                        .Build();

    return pbeDeriver.deriveKey(TargetKeyType.CIPHER, (keySizeInBits / 8));
}
```

The KeyType parameter above can also be set to KeyType.MAC. In the case of PBKDF2 the type parameter is ignored, but is in place in case other algorithms such as PKCS#12 need to be introduced later.

10 Random Number Generators

The module offers the 3 standard DRBGs defined in SP 800-90A. A DRBG generates a FipsSecureRandom object. Only a DRBG builder from the Org.BouncyCastle.Fips namespace can produce a FipsSecureRandom.

10.1 Available in Approved Mode Operation

The following DRBG types can be constructed in approved mode of operation.

DRBG Name	Low-level Class
HASH DRBG	FipsDrbg
HMAC DRBG	FipsDrbg
CTR DRBG	FipsDrbg

10.2 Available in General Operation

There are currently no general operation specific DRBG available.

10.3 Examples

10.3.1 Creation for SHA512 DRBG

This example creates a DRBG based on a default SecureRandom, in this case the DRBG will reseed itself using the generateSeed() method on the SecureRandom, and the nonce is generated using one of the recommended techniques in NIST SP 800-90A, a string of bits from the entropy source of ½ the security strength in size.

```
byte[] personalizationString = Strings.toUTF8ByteArray("Personalization String");

SecureRandom entropySource = new SecureRandom();

SecureRandom random = CryptoServicesRegistrar.CreateService(FipsDrbg.Sha512)
                                .FromEntropySource(entropySource,true)
                                .SetPersonalizationString(personalizationString))
                                .Build(
                                       entropySource.GenerateSeed((256 / (2 * 8))),
                                       true,
                                       Strings.toByteArray("Additional Input"));
```

11 Signature Algorithms

A range of signature algorithms are available both in the low-level API and the provider. Some non-FIPS variations of DSA, ECDSA and EdDSAare also available in the general operation set.

11.1 Available in Approved Mode Operation

In approved mode of operation DSA keys of 1024 bits are supported for signature verification only.

11.1.1 DSA

Signature	Class
SHA1 with DSA	FipsDsa
SHA224 with DSA	FipsDsa
SHA256 with DSA	FipsDsa
SHA384 with DSA	FipsDsa
SHA512 with DSA	FipsDsa
SHA512(224) with DSA	FipsDsa
SHA512(256) with DSA	FipsDsa

11.1.2 EC

Signature	Class
SHA1 with ECDSA	FipsEC
SHA224 with ECDSA	FipsEC
SHA256 with ECDSA	FipsEC
SHA384 with ECDSA	FipsEC
SHA512 with ECDSA	FipsEC
SHA512(224) with ECDSA	FipsEC
SHA512(256) with ECDSA	FipsEC

11.1.3 RSA

In approved mode of operation RSA keys of 1024 bit are supported for signature verification only.

11.1.3.1 PKCS1.5

Signature	Class	Parameter Configuration
SHA1 with RSA	FipsRsa	FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha1)
SHA224 with RSA	FipsRsa	FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha224)
SHA256 with RSA	FipsRsa	FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha256)
SHA384 with RSA	FipsRsa	FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha384)
SHA512 with RSA	FipsRsa	FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512)
SHA512(224) with RSA	FipsRsa	FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512_224)
SHA512(256) with RSA	FipsRsa	FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512_256)

11.1.3.2 PSS

Signature	Class	Parameter Configuration
PSS SHA1 with RSA	FipsRsa	FipsRsa.Pss.WithDigest(FipsShs.Sha1)
PSS SHA224 with RSA	FipsRsa	FipsRsa.Pss.WithDigest(FipsShs.Sha224)
PSS SHA256 with RSA	FipsRsa	FipsRsa.Pss.WithDigest(FipsShs.Sha256)
PSS SHA384 with RSA	FipsRsa	FipsRsa.Pss.WithDigest(FipsShs.Sha384)
PSS SHA512 with RSA	FipsRsa	FipsRsa.Pss.WithDigest(FipsShs.Sha512)
PSS SHA512(224) with RSA	FipsRsa	FipsRsa.Pss.WithDigest(FipsShs.Sha512_224)
PSS SHA512(256) with RSA	FipsRsa	FipsRsa.Pss.WithDigest(FipsShs.Sha512_256)

11.1.3.3 X9.31

Signature	Class	Parameter Configuration
X9.31 SHA1 with RSA	FipsRsa	FipsRsa.X931.WithDigest(FipsShs.Sha1)
X9.31 SHA224 with RSA	FipsRsa	FipsRsa.X931.WithDigest(FipsShs.Sha224)
X9.31 SHA256 with RSA	FipsRsa	FipsRsa.X931.WithDigest(FipsShs.Sha256)
X9.31 SHA384 with RSA	FipsRsa	FipsRsa.X931.WithDigest(FipsShs.Sha384)
X9.31 SHA512 with RSA	FipsRsa	FipsRsa.X931.WithDigest(FipsShs.Sha512)
X9.31 SHA512(224) with RSA	FipsRsa	FipsRsa.X931.WithDigest(FipsShs.Sha512_224)
X9.31 SHA512(256) with RSA	FipsRsa	FipsRsa.X931.WithDigest(FipsShs.Sha512_256)

11.2 Available in General Operation

11.2.1 DSA

11.2.1.1 Regular DSA

There are currently no extra DSA signature types available in general mode operation.

11.2.1.2 Deterministic DSA

Signature	Class	Parameter Configuration
Deterministic SHA1 with DSA	Dsa	Dsa.DDsa.WithDigest(FipsShs.Sha1)
Deterministic SHA224 with DSA	Dsa	Dsa.DDsa.WithDigest(FipsShs.Sha224)
Deterministic SHA256 with DSA	Dsa	Dsa.DDsa.WithDigest(FipsShs.Sha256)
Deterministic SHA384 with DSA	Dsa	Dsa.DDsa.WithDigest(FipsShs.Sha384)
Deterministic SHA512 with DSA	Dsa	Dsa.DDsa.WithDigest(FipsShs.Sha512)
Deterministic SHA512(224) with DSA	Dsa	Dsa.DDsa.WithDigest(FipsShs.Sha512_224)
Deterministic SHA512(256) with DSA	Dsa	Dsa.DDsa.WithDigest(FipsShs.Sha512_256)

11.2.3 ECDSA

11.2.3.1 Regular ECDSA

There are currently no extra ECDSA signature types available in general mode operation.

11.2.3.2 Deterministic ECDSA

Signature	Class	Parameter Configuration
Deterministic SHA1 with ECDSA	EC	EC.DDsa.WithDigest(FipsShs.Sha1)
Deterministic SHA224 with ECDSA	EC	EC.DDsa.WithDigest(FipsShs.Sha224)
Deterministic SHA256 with ECDSA	EC	EC.DDsa.WithDigest(FipsShs.Sha256)
Deterministic SHA384 with ECDSA	EC	EC.DDsa.WithDigest(FipsShs.Sha384)
Deterministic SHA512 with ECDSA	EC	EC.DDsa.WithDigest(FipsShs.Sha512)
Deterministic SHA512 with ECDSA	EC	EC.DDsa.WithDigest(FipsShs.Sha512_224)
Deterministic SHA512 with ECDSA	EC	EC.DDsa.WithDigest(FipsShs.Sha512_256)

11.2.4 EdDSA

DSA using Edwards Curves is supported for both Ed25519 and Ed448.

Signature	Class	Parameter Configuration
Ed25519	EdEC	EdEC.Ed25519

Ed448	EdEC	EdEC.Ed448

11.2.5 SPHINCS-256

The SPHINCS-256 signature algorithm is a post-quantum stateless hash based signature algorithm designed to provide long-term security on the order of 2^{128} even against attackers using quantum computers. The API currently supports two versions of SPHINCS, one based on SHA-512 and one based on SHA-3. The following table shows the two variants.

Signature	Class	Parameter Configuration
SPHINCS-256 with SHA-512	Sphincs	Sphincs.Sphincs256.WithDigest(FipsShs.Sha512)
SPHINCS-256 with SHA3-512	Sphincs	Sphincs.Sphincs256.WithDigest(FipsShs.Sha3_512)

11.3 Examples

11.3.1 RSA with SHA-1

The following sample shows a method for generating a signature in PKCS#1 v1.5 format.

Note: while PKCS#1 v1.5 signing does not use random data in the actual signature calculation, a SecureRandom is required in this case to provide blinding to help protect the private key.

```
public byte[] signData(AsymmetricRsaPrivateKey privKey, byte[] data)
{
    ISignatureFactoryService signatureFactoryProvider =
                        CryptoServicesRegistrar.CreateService(
                                                privKey, new SecureRandom());
    ISignatureFactory<FipsRsa.SignatureParameters> rsaSig =
                         signatureFactoryProvider.CreateSignatureFactory(
                                 FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha1));
    IStreamCalculator<IBlockResult> sigCalc = rsaSig.CreateCalculator();

    Stream sOut = sigCalc.Stream;

    sOut.Write(data, 0, data.Length);

    sOut.Close();

    return sigCalc.GetResult().Collect();
}
```

The following method can be used to verify the signature return by signData providing pubKey is the public key corresponding to privKey used in the signData method.

```
public boolean isVerified(
                 AsymmetricRsaPublicKey pubKey, byte[] signature, byte[] data)
{
    IVerifierFactoryService verifierFactoryProvider =
                        CryptoServicesRegistrar.CreateService(rsaPublic);
    IVerifierFactory<FipsRsa.SignatureParameters> rsaVer =
                        verifierFactoryProvider.CreateVerifierFactory(
                                  FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha1));
    IStreamCalculator<IVerifier> verCalc = rsaVer.CreateCalculator();
```

```
    sOut = verCalc.Stream;

    sOut.Write(data, 0, data.Length);

    sOut.Close();

    return verCalc.GetResult().IsVerified(signature);
}
```

11.3.2 ECDSA with SHA-256

The following sample shows a method for generating an ECDSA signature with SHA-256 using the JCA provider. This method will use the provider default SecureRandom for generating the random component for the ECDSA signature.

```
public byte[] signData(AsymmetricECPrivateKey privKey, byte[] data)
{
    ISignatureFactoryService signatureFactoryProvider =
                        CryptoServicesRegistrar.CreateService(
                                                privKey, new SecureRandom());
    ISignatureFactory<FipsRsa.SignatureParameters> ecDsaSig =
                         signatureFactoryProvider.CreateSignatureFactory(
                                    FipsEC.Dsa.WithDigest(FipsShs.Sha256));
    IStreamCalculator<IBlockResult> sigCalc = ecDsaSig.CreateCalculator();

    Stream sOut = sigCalc.Stream;

    sOut.Write(data, 0, data.Length);

    sOut.Close();

    return sigCalc.GetResult().Collect();
}
```

The following method can be used to verify the signature return by signData providing pubKey is the public key corresponding to privKey used in the signData method.

```
public boolean isVerified(AsymmetricECPublicKey pubKey,
                                                byte[] signature, byte[] data)
{
    IVerifierFactoryService verifierFactoryProvider =
                        CryptoServicesRegistrar.CreateService(rsaPublic);
    IVerifierFactory<FipsRsa.SignatureParameters> rsaVer =
                        verifierFactoryProvider.CreateVerifierFactory(
                                  FipsEC.Dsa.WithDigest(FipsShs.Sha256));
    IStreamCalculator<IVerifier> verCalc = rsaVer.CreateCalculator();

    sOut = verCalc.Stream;

    sOut.Write(data, 0, data.Length);

    sOut.Close();

    return verCalc.GetResult().IsVerified(signature);
}
```

11.3.3 SPHINCS-256 with SHA3-512

The following example shows how to generate a SPHINCS-256 key pair and then create and verify a signature based on SHA3-512. Two digests are allowed for key generation, Sha3_256 and Sha512_256 to go with the use of Sha3_512 and Sha512 for signature generation respectively.

```
Sphincs.KeyPairGenerator kpGen =
      CryptoServicesRegistrar.CreateGenerator(
                  new Sphincs.KeyGenerationParameters(FipsShs.Sha3_256),
                                                              new SecureRandom());

AsymmetricKeyPair<AsymmetricSphincsPublicKey, AsymmetricSphincsPrivateKey>
                                                    kp = kpGen.GenerateKeyPair();

ISignatureFactory<Sphincs.SignatureParameters> sigFactory =
                    CryptoServicesRegistrar.CreateService(kp.PrivateKey)
                          .CreateSignatureFactory(
                              Sphincs.Sphincs256.WithDigest(FipsShs.Sha3_512));

IStreamCalculator<IBlockResult> calc = sigFactory.CreateCalculator();

calc.Stream.Write(msg, 0, msg.Length);

calc.Stream.Close();

byte[] sig = calc.GetResult().Collect();

IVerifierFactory<Sphincs.SignatureParameters> verifierFactory =
                        CryptoServicesRegistrar.CreateService(kp.PublicKey)
                        .CreateVerifierFactory(
                              Sphincs.Sphincs256.WithDigest(FipsShs.Sha3_512));

IStreamCalculator<IVerifier> vcalc = verifierFactory.CreateCalculator();

vcalc.Stream.Write(msg, 0, msg.Length);

vcalc.Stream.Close();

if (!vcalc.GetResult().IsVerified(sig))
{
    Console.Out.WriteLine(“signature failed”);
}
```

Appendix A – System Properties

By default all the below properties are assumed to be false.

`Org.BouncyCastle.Rsa.AllowMultiUse` – in approved/unapproved mode the module will attempt to block an RSA modulus from being used for encryption if it has been used for signing, or visa-versa. If the module is not in approved mode it is possible to stop this from happening by setting **`Org.BouncyCastle.Rsa.AllowMultiUse`** to **`true`**.

`Org.BouncyCastle.TripleDes.AllowWeak` – setting this property to **`true`** will allow the use of TripleDES weak keys. This is only present as it is a requirement for CAVP testing.

`Org.BouncyCastle.Pkcs1.NotStrict` – some other providers of cryptography services fail to produce PKCS1 encoded block that are the correct length. Setting this property to **`true`** will relax the conformance check on the block length.

`Org.BouncyCastle.Fpe.Disable` – setting this property to **`true`** will disable support for format preserving encryption algorithms.

`Org.BouncyCastle.Fpe.Disable_Ff1` – setting this property to **`true`** will disable support for FF1 format preserving encryption only.

Appendix B – Built in Curves

The following curves are available in general mode operation for ECDSA and/or ECDH/ECCDH/ECMQV. In the case of approved only mode, only curves offering a security level of 112 bits or greater can be used.

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
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
NIST K-409	K-409	1.3.132.0.36
NIST K-571	K-571	1.3.132.0.38
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
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

Name	IDs	OID
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
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 C – Built in DH FFC Parameters

Name	ID	RFC
ffdhe2048	ffdhe2048	RFC 7919
ffdhe3072	ffdhe3072	RFC 7919
ffdhe4096	ffdhe4096	RFC 7919
ffdhe6144	ffdhe6144	RFC 7919
ffdhe8192	ffdhe8192	RFC 7919
2048-bit MODP Group	modp2048	RFC 3526
3072-bit MODP Group	modp3072	RFC 3526
4096-bit MODP Group	modp4096	RFC 3526
6144-bit MODP Group	modp6144	RFC 3526
8192-bit MODP Group	modp8192	RFC 3526

Appendix D – Troubleshooting

1. Application using the BC FIPS module periodically blocks, especially on start up.

The BC FIPS module is very careful about making sure it has access to sufficient entropy to generate good quality keys. Occasionally the external source the module is using will run out of entropy and block. The best way to deal with this is to make sure the execution environment has hardware RNG turned on if it is available.

Appendix D – Disclosures

Please note that just as patents can vary from jurisdiction to jurisdiction in any field of endeavour, the same applies to cryptographic algorithms where some techniques which are freely usable in some places are patented in others. It is your responsibility to make sure you understand the situation with patents as it applies to your circumstances.

There are no disclosures we are aware of at the moment connected with this module.

Appendix E – References

“Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program”, National Institute of Standards and Technology and the Communications Security Establishment Canada. May 4, 2021.

“BC-FNA (Bouncy Castle FIPS .NET API) Non-Proprietary FIPS 140-2 Cryptographic Module Security Policy”, Legion of the Bouncy Castle Inc. July, 2021.