Copyright and Trademark Notice

This document is licensed under a Creative Commons Attribution 3.0 Unported License (http://creativecommons.org/licenses/by/3.0/)

Sponsored By

http://www.windtalkersecurity.com

http://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.

The following people also contributed financially to this project:

Jeff Hildebrand.

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

Validation testing was performed by CGI Global IT Security Labs – Canada. For more information on the re-validations of the software contact:

Jason Cunningham
Program Manager, FIPS 140
CGI Global IT Security Labs - Canada
1410 Blair Place, 7th floor
Ottawa, ON K1J 9B9
Canada

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 ......................................................................................................................... 6
2 Installation and Startup ........................................................................................................ 7
   2.1 Installation ........................................................................................................................ 7
   2.2 Startup .............................................................................................................................. 7
   2.3 Approved Only Mode ....................................................................................................... 7
3 Cipher Algorithms (Symmetric) ........................................................................................... 8
   3.1 Available in Approved Mode Operation .......................................................................... 8
   3.2 Available in General Operation ....................................................................................... 8
   3.3 Paddings Available .......................................................................................................... 8
      3.3.1 AES Encryption using CBC and PKCS5/7Padding ..................................................... 9
      3.3.2 AES Encryption using CTR mode ............................................................................. 10
      3.3.3 AES using GCM and an AEADParameterSpec ......................................................... 10
      3.3.4 Stream Encryption using ChaCha ............................................................................. 11
4 Key Agreement and Exchange Algorithms ......................................................................... 13
   4.1 Available in Approved Mode Operation .......................................................................... 13
   4.2 Available in General Operation ....................................................................................... 13
   4.3 Examples .......................................................................................................................... 13
      4.3.1 Basic Agreement with Cofactor ............................................................................... 13
      4.3.2 Basic Agreement with PRF ..................................................................................... 13
      4.3.3 Basic Agreement with KDF .................................................................................... 14
      4.3.3 Key Exchange with NewHope ................................................................................ 14
5 Key Derivation Functions ..................................................................................................... 15
   5.1 Available in Approved Mode Operation .......................................................................... 15
   5.2 Available in General Operation ....................................................................................... 15
   5.3 Examples .......................................................................................................................... 15
      5.3.1 TLS 1.2 KDF ........................................................................................................... 15
      5.3.2 X9.63 KDF ............................................................................................................. 16
6 Key Wrapping Algorithms .................................................................................................. 17
   6.1 Available in Approved Mode Operation .......................................................................... 17
   6.2 Available in General Operation ....................................................................................... 17
   6.3 Examples .......................................................................................................................... 17
      6.3.1 Key Wrapping using RSA ....................................................................................... 17
      6.3.2 Key Wrapping using AES ....................................................................................... 18
      6.3.3 Key Wrapping using Camellia with padding using inverse function ....................... 18
7 Mac Algorithms ................................................................................................................... 20
   7.1 Available in Approved Mode Operation .......................................................................... 20
   7.2 Available in General Operation ....................................................................................... 20
   7.3 Examples .......................................................................................................................... 20
      7.3.1 AES using CMAC – 64 bit ....................................................................................... 20
      7.3.2 HMAC-SHA256 ....................................................................................................... 21
      7.3.3 Using Poly1305 with key generation ...................................................................... 21
8 Message Digest Algorithms ................................................................................................ 23
   8.1 Available in Approved Mode Operation .......................................................................... 23
   8.2 Available in General Operation ....................................................................................... 23
   8.3 Examples .......................................................................................................................... 23
      8.3.1 Use of SHA-256 ...................................................................................................... 23
      8.3.2 Use of SHAKE128 ................................................................................................. 24
9 Password Based Key Derivation Functions ........................................................................ 25
   9.1 Available in Approved Mode Operation .......................................................................... 25
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-1.0.0.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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Class</th>
<th>Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>FipsAes</td>
<td>ECB, CBC, CFB8, CFB128, OFB, CTR, CCM, GCM</td>
</tr>
<tr>
<td>TripleDES</td>
<td>FipsTripleDes</td>
<td>ECB, CBC, CFB8, CFB64, OFB, CTR</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Class</th>
<th>Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARC4</td>
<td>Arc4</td>
<td>N/A</td>
</tr>
<tr>
<td>AES</td>
<td>Aes</td>
<td>OpenPGPCFB</td>
</tr>
<tr>
<td>Camellia</td>
<td>Camellia</td>
<td>CBC, CCM, CFB8, CFB128, CTR, ECB, GCM, OCB, OFB, OpenPGPCFB</td>
</tr>
<tr>
<td>ChaCha</td>
<td>ChaCha</td>
<td>eSTREAM, IETF (RFC 7539)</td>
</tr>
<tr>
<td>SEED</td>
<td>Seed</td>
<td>CBC, CCM, CFB8, CFB128, CTR, ECB, GCM, OFB</td>
</tr>
<tr>
<td>Serpent</td>
<td>Serpent</td>
<td>CBC, CCM, CFB8, CFB128, CTR, ECB, GCM, OFB</td>
</tr>
<tr>
<td>TripleDES</td>
<td>TripleDes</td>
<td>OpenPGPCFB</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Padding Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECB</td>
<td>NONE, PKCS7 (PKCS5), ISO10126-2, X9.23, ISO7816-4 (ISO9797-1), TBC</td>
</tr>
<tr>
<td>CBC</td>
<td>NONE, PKCS7 (PKCS5), ISO10126-2, X9.23, ISO7816-4 (ISO9797-1), TBC, CS1, CS2, CS3</td>
</tr>
</tbody>
</table>

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

```csharp
// 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("5627d7c5d9cbdcde2b2c496e60a31a64779a327d753"
    + "c8cdecf8775964540f63d7856d6d6d6b3b113565355ff466"
    + "8d19c1c2a17e19de1257b9b1ba86b23693677f8ef91792"
    + "4e7e68bbadcc058c9dc2153eb9359c9ac6d2f3a0");

// plain text data
byte[] data = Hex.Decode("29da07d47becd5968b776ff0a8f9fdd212dfabb"
// generate 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("0000000000000004a00000000");

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"
    + "ff0102030405060708090a0b0c0d0e0f");
// write out the plain text
using (Stream encOut = chacha.Stream)
{
    encOut.Write(input, 0, input.Length);
}

byte[] output = bOut.ToArray(); // encrypted output
4 Key Agreement and Exchange Algorithms

Key agreement algorithms are available based around both elliptic curve. 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

<table>
<thead>
<tr>
<th>Key Agreement Method</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECCDH</td>
<td>FipsEC</td>
<td>FipsEC.Cdh</td>
</tr>
</tbody>
</table>

4.2 Available in General Operation

<table>
<thead>
<tr>
<th>Key Agreement Method</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewHope</td>
<td>NewHope</td>
<td>NewHope.Sha3_256</td>
</tr>
</tbody>
</table>

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.

```java
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.

```java
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.

```csharp
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:

```csharp
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

<table>
<thead>
<tr>
<th>Derivation Method</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS 1.0</td>
<td>FipsKdf</td>
<td>FipsKdf.Tls1_0</td>
</tr>
<tr>
<td>TLS 1.1</td>
<td>FipsKdf</td>
<td>FipsKdf.Tls1_1</td>
</tr>
<tr>
<td>TLS 1.2</td>
<td>FipsKdf</td>
<td>FipsKdf.Tls1_2</td>
</tr>
<tr>
<td>X 9.63</td>
<td>FipsKdf</td>
<td>FipsKdf.X963</td>
</tr>
<tr>
<td>Concatenation</td>
<td>FipsKdf</td>
<td>FipsKdf.Concatenation</td>
</tr>
</tbody>
</table>

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.

```csharp
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.

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Algorithm</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>KW</td>
<td>FipsAES</td>
<td>FipsAes.KW</td>
</tr>
<tr>
<td>AES</td>
<td>KWP</td>
<td>FipsAES</td>
<td>FipsAes.KWP</td>
</tr>
<tr>
<td>DESede</td>
<td>TKW</td>
<td>FipsTripleDES</td>
<td>FipsAes.TKW</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Algorithm</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>OAEP</td>
<td>FipsRsa</td>
<td>FipsRsa.WrapOaep</td>
</tr>
</tbody>
</table>

6.2 Available in General Operation

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Algorithm</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camellia</td>
<td>KW</td>
<td>Camellia</td>
<td>Camellia.KW</td>
</tr>
<tr>
<td>Camellia</td>
<td>KWP</td>
<td>Camellia</td>
<td>Camellia.KWP</td>
</tr>
<tr>
<td>ElGamal</td>
<td>OAEP</td>
<td>ElGamal</td>
<td>ElGamal.WrapOaep</td>
</tr>
<tr>
<td>ElGamal</td>
<td>PKCS#1.5</td>
<td>ElGamal</td>
<td>ElGamal.WrapPkcs1v15</td>
</tr>
<tr>
<td>RSA</td>
<td>PKCS#1.5</td>
<td>RSA</td>
<td>RSA.WrapPkcs1v15</td>
</tr>
<tr>
<td>SEED</td>
<td>KW</td>
<td>Seed</td>
<td>Seed.KW</td>
</tr>
<tr>
<td>SEED</td>
<td>KWP</td>
<td>Seed</td>
<td>Seed.KWP</td>
</tr>
<tr>
<td>Serpent</td>
<td>KW</td>
<td>Serpent</td>
<td>Serpent.KW</td>
</tr>
<tr>
<td>Serpent</td>
<td>KWP</td>
<td>Serpent</td>
<td>Serpent.KWP</td>
</tr>
</tbody>
</table>

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.

```java
public byte[] wrapKey(AsymmetricRsaPublicKey publicKey, byte[] keyData)
{
    IKeyWrapper<FipsRsa.OaepWrapParameters> keyWrapper =
        CryptoServicesRegistrar.CreateService(publicKey, 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.

```csharp
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.

```csharp
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`.

```csharp
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.

```csharp
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

<table>
<thead>
<tr>
<th>MAC Name</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES CMAC</td>
<td>FipsAes</td>
<td>FipsAes.CMac</td>
</tr>
<tr>
<td>AES GMAC</td>
<td>FipsAes</td>
<td>FipsAes.GMac</td>
</tr>
<tr>
<td>HMAC SHA-1</td>
<td>FipsShs</td>
<td>FipsShs.Sha1HMac</td>
</tr>
<tr>
<td>HMAC SHA-224</td>
<td>FipsShs</td>
<td>FipsShs.Sha224HMac</td>
</tr>
<tr>
<td>HMAC SHA-256</td>
<td>FipsShs</td>
<td>FipsShs.Sha256HMac</td>
</tr>
<tr>
<td>HMAC SHA-384</td>
<td>FipsShs</td>
<td>FipsShs.Sha384HMac</td>
</tr>
<tr>
<td>HMAC SHA-512</td>
<td>FipsShs</td>
<td>FipsShs.Sha512HMac</td>
</tr>
<tr>
<td>HMAC SHA-512(224)</td>
<td>FipsShs</td>
<td>FipsShs.Sha512_224HMac</td>
</tr>
<tr>
<td>HMAC SHA-512(256)</td>
<td>FipsShs</td>
<td>FipsShs.Sha512_256HMac</td>
</tr>
<tr>
<td>Triple-DES CMAC</td>
<td>FipsTripleDes</td>
<td>FipsTripleDes.CMac</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>MAC Name</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camellia CMAC</td>
<td>Camellia</td>
<td>Camellia.CMac</td>
</tr>
<tr>
<td>Camellia GMAC</td>
<td>Camellia</td>
<td>Camellia.GMac</td>
</tr>
<tr>
<td>Poly1305</td>
<td>Poly1305</td>
<td>Poly1305.Mac</td>
</tr>
<tr>
<td>SEED CMAC</td>
<td>Seed</td>
<td>Seed.CMac</td>
</tr>
<tr>
<td>SEED GMAC</td>
<td>Seed</td>
<td>Seed.GMac</td>
</tr>
<tr>
<td>Serpent CMAC</td>
<td>Serpent</td>
<td>Serpent.CMac</td>
</tr>
<tr>
<td>Serpent GMAC</td>
<td>Serpent</td>
<td>Serpent.GMac</td>
</tr>
</tbody>
</table>

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.

```java
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.

```java
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.

```java
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

<table>
<thead>
<tr>
<th>Digest Name</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>FipsShs</td>
<td>FipsShs.Sha1</td>
</tr>
<tr>
<td>SHA-224</td>
<td>FipsShs</td>
<td>FipsShs.Sha224</td>
</tr>
<tr>
<td>SHA-256</td>
<td>FipsShs</td>
<td>FipsShs.Sha256</td>
</tr>
<tr>
<td>SHA-384</td>
<td>FipsShs</td>
<td>FipsShs.Sha384</td>
</tr>
<tr>
<td>SHA-512</td>
<td>FipsShs</td>
<td>FipsShs.Sha512</td>
</tr>
<tr>
<td>SHA-512(224)</td>
<td>FipsShs</td>
<td>FipsShs.Sha512_224</td>
</tr>
<tr>
<td>SHA-512(256)</td>
<td>FipsShs</td>
<td>FipsShs.Sha512_256</td>
</tr>
<tr>
<td>SHA3-224</td>
<td>FipsShs</td>
<td>FipsShs.Sha3_224</td>
</tr>
<tr>
<td>SHA3-256</td>
<td>FipsShs</td>
<td>FipsShs.Sha3_256</td>
</tr>
<tr>
<td>SHA3-384</td>
<td>FipsShs</td>
<td>FipsShs.Sha3_384</td>
</tr>
<tr>
<td>SHA3-512</td>
<td>FipsShs</td>
<td>FipsShs.Sha3_512</td>
</tr>
<tr>
<td>SHAKE128</td>
<td>FipsShs</td>
<td>FipsShs.Shake128</td>
</tr>
<tr>
<td>SHAKE256</td>
<td>FipsShs</td>
<td>FipsShs.Shake256</td>
</tr>
</tbody>
</table>

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.

```csharp
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.

```csharp
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

```csharp
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Byte Encoding Used</th>
<th>Digest for PRF</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBKDF2</td>
<td>UTF-8</td>
<td>SHA-1</td>
<td>FipsPbkd</td>
</tr>
<tr>
<td>PBKDF2</td>
<td>8-BIT/ASCII</td>
<td>SHA-1</td>
<td>FipsPbkd</td>
</tr>
<tr>
<td>PBKDF2</td>
<td>UTF-8</td>
<td>SHA-224</td>
<td>FipsPbkd</td>
</tr>
<tr>
<td>PBKDF2</td>
<td>UTF-8</td>
<td>SHA-256</td>
<td>FipsPbkd</td>
</tr>
<tr>
<td>PBKDF2</td>
<td>UTF-8</td>
<td>SHA-384</td>
<td>FipsPbkd</td>
</tr>
<tr>
<td>PBKDF2</td>
<td>UTF-8</td>
<td>SHA-512</td>
<td>FipsPbkd</td>
</tr>
</tbody>
</table>

9.2 Available in General Operation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Byte Encoding Used</th>
<th>Digest for PRF</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL</td>
<td>8-BIT/ASCII</td>
<td>MD5</td>
<td>Pbkd</td>
</tr>
<tr>
<td>PKCS#12</td>
<td>PKCS#12</td>
<td>SHA-1</td>
<td>Pbkd</td>
</tr>
<tr>
<td>PKCS#12</td>
<td>PKCS#12</td>
<td>SHA-224</td>
<td>Pbkd</td>
</tr>
<tr>
<td>PKCS#12</td>
<td>PKCS#12</td>
<td>SHA-256</td>
<td>Pbkd</td>
</tr>
<tr>
<td>PKCS#12</td>
<td>PKCS#12</td>
<td>SHA-384</td>
<td>Pbkd</td>
</tr>
<tr>
<td>PKCS#12</td>
<td>PKCS#12</td>
<td>SHA-512</td>
<td>Pbkd</td>
</tr>
</tbody>
</table>

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.

```java
public byte[] getSecretKey(char[] password, int iterationCount, byte[] salt, int keySizeInBits) {
    IPasswordBasedDeriverBuilder<FipsPbkd.Parameters> pbeDeriverBuilder =
            CryptoServicesRegistrar.CreateService(FipsPbkd.PbkdfF2)
            .From(converter, password);

    IPasswordBasedDeriver<FipsPbkd.Parameters> pbeDeriver =
            pbeDeriverBuilder
```
.WithPrf(FipsShs.\texttt{Sha256})
 .WithSalt(Hex.Decode("0102030405060708090a0b0c0d0e0f"))
 .WithIterationCount(1024)
 .Build();

    return pbeDeriver.deriveKey(TargetKeyType.\texttt{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.

<table>
<thead>
<tr>
<th>DRBG Name</th>
<th>Low-level Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH DRBG</td>
<td>FipsDrbg</td>
</tr>
<tr>
<td>HMAC DRBG</td>
<td>FipsDrbg</td>
</tr>
<tr>
<td>CTR DRBG</td>
<td>FipsDrbg</td>
</tr>
</tbody>
</table>

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.

```java
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, and ECDSA are 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

<table>
<thead>
<tr>
<th>Signature</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1 with DSA</td>
<td>FipsDsa</td>
</tr>
<tr>
<td>SHA224 with DSA</td>
<td>FipsDsa</td>
</tr>
<tr>
<td>SHA256 with DSA</td>
<td>FipsDsa</td>
</tr>
<tr>
<td>SHA384 with DSA</td>
<td>FipsDsa</td>
</tr>
<tr>
<td>SHA512 with DSA</td>
<td>FipsDsa</td>
</tr>
<tr>
<td>SHA512(224) with DSA</td>
<td>FipsDsa</td>
</tr>
<tr>
<td>SHA512(256) with DSA</td>
<td>FipsDsa</td>
</tr>
</tbody>
</table>

11.1.2 EC

<table>
<thead>
<tr>
<th>Signature</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1 with ECDSA</td>
<td>FipsEC</td>
</tr>
<tr>
<td>SHA224 with ECDSA</td>
<td>FipsEC</td>
</tr>
<tr>
<td>SHA256 with ECDSA</td>
<td>FipsEC</td>
</tr>
<tr>
<td>SHA384 with ECDSA</td>
<td>FipsEC</td>
</tr>
<tr>
<td>SHA512 with ECDSA</td>
<td>FipsEC</td>
</tr>
<tr>
<td>SHA512(224) with ECDSA</td>
<td>FipsEC</td>
</tr>
<tr>
<td>SHA512(256) with ECDSA</td>
<td>FipsEC</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>Signature</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha1)</td>
</tr>
<tr>
<td>SHA224 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha224)</td>
</tr>
<tr>
<td>SHA256 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha256)</td>
</tr>
<tr>
<td>SHA384 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha384)</td>
</tr>
<tr>
<td>SHA512 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512)</td>
</tr>
<tr>
<td>SHA512(224) with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512_224)</td>
</tr>
<tr>
<td>SHA512(256) with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512_256)</td>
</tr>
</tbody>
</table>

### 11.1.3.2 PSS

<table>
<thead>
<tr>
<th>Signature</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSS SHA1 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pss.WithDigest(FipsShs.Sha1)</td>
</tr>
<tr>
<td>PSS SHA224 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pss.WithDigest(FipsShs.Sha224)</td>
</tr>
<tr>
<td>PSS SHA256 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pss.WithDigest(FipsShs.Sha256)</td>
</tr>
<tr>
<td>PSS SHA384 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pss.WithDigest(FipsShs.Sha384)</td>
</tr>
<tr>
<td>PSS SHA512 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pss.WithDigest(FipsShs.Sha512)</td>
</tr>
<tr>
<td>PSS SHA512(224) with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pss.WithDigest(FipsShs.Sha512_224)</td>
</tr>
<tr>
<td>PSS SHA512(256) with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.Pss.WithDigest(FipsShs.Sha512_256)</td>
</tr>
</tbody>
</table>

### 11.1.3.3 X9.31

<table>
<thead>
<tr>
<th>Signature</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>X9.31 SHA1 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.X931.WithDigest(FipsShs.Sha1)</td>
</tr>
<tr>
<td>X9.31 SHA224 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.X931.WithDigest(FipsShs.Sha224)</td>
</tr>
<tr>
<td>X9.31 SHA256 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.X931.WithDigest(FipsShs.Sha256)</td>
</tr>
<tr>
<td>X9.31 SHA512 with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.X931.WithDigest(FipsShs.Sha512)</td>
</tr>
<tr>
<td>X9.31 SHA512(224) with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.X931.WithDigest(FipsShs.Sha512_224)</td>
</tr>
<tr>
<td>X9.31 SHA512(256) with RSA</td>
<td>FipsRsa</td>
<td>FipsRsa.X931.WithDigest(FipsShs.Sha512_256)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Signature</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic SHA1 with DSA</td>
<td>Dsa</td>
<td>Dsa.DDsa.WithDigest(FipsShs.Sha1)</td>
</tr>
<tr>
<td>Deterministic SHA224 with DSA</td>
<td>Dsa</td>
<td>Dsa.DDsa.WithDigest(FipsShs.Sha224)</td>
</tr>
<tr>
<td>Deterministic SHA256 with DSA</td>
<td>Dsa</td>
<td>Dsa.DDsa.WithDigest(FipsShs.Sha256)</td>
</tr>
<tr>
<td>Deterministic SHA384 with DSA</td>
<td>Dsa</td>
<td>Dsa.DDsa.WithDigest(FipsShs.Sha384)</td>
</tr>
<tr>
<td>Deterministic SHA512 with DSA</td>
<td>Dsa</td>
<td>Dsa.DDsa.WithDigest(FipsShs.Sha512)</td>
</tr>
<tr>
<td>Deterministic SHA512(224) with DSA</td>
<td>Dsa</td>
<td>Dsa.DDsa.WithDigest(FipsShs.Sha512_224)</td>
</tr>
<tr>
<td>Deterministic SHA512(256) with DSA</td>
<td>Dsa</td>
<td>Dsa.DDsa.WithDigest(FipsShs.Sha512_256)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Signature</th>
<th>Class</th>
<th>Parameter Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic SHA1 with ECDSA</td>
<td>EC</td>
<td>EC.DDsa.WithDigest(FipsShs.Sha1)</td>
</tr>
<tr>
<td>Deterministic SHA224 with ECDSA</td>
<td>EC</td>
<td>EC.DDsa.WithDigest(FipsShs.Sha224)</td>
</tr>
<tr>
<td>Deterministic SHA256 with ECDSA</td>
<td>EC</td>
<td>EC.DDsa.WithDigest(FipsShs.Sha256)</td>
</tr>
<tr>
<td>Deterministic SHA384 with ECDSA</td>
<td>EC</td>
<td>EC.DDsa.WithDigest(FipsShs.Sha384)</td>
</tr>
<tr>
<td>Deterministic SHA512 with ECDSA</td>
<td>EC</td>
<td>EC.DDsa.WithDigest(FipsShs.Sha512)</td>
</tr>
<tr>
<td>Deterministic SHA512 with ECDSA</td>
<td>EC</td>
<td>EC.DDsa.WithDigest(FipsShs.Sha512_224)</td>
</tr>
<tr>
<td>Deterministic SHA512 with ECDSA</td>
<td>EC</td>
<td>EC.DDsa.WithDigest(FipsShs.Sha512_256)</td>
</tr>
</tbody>
</table>

11.2.4 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.
### 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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>IDs</th>
<th>OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSSI FRP 256v1</td>
<td>FRP256v1</td>
<td>1.2.250.1.223.101.256.1</td>
</tr>
<tr>
<td>ECC Brainpool P160r1</td>
<td>brainpoolp160r1</td>
<td>1.3.36.3.3.2.8.1.1.1</td>
</tr>
<tr>
<td>ECC Brainpool P160t1</td>
<td>brainpoolp160t1</td>
<td>1.3.36.3.3.2.8.1.1.2</td>
</tr>
<tr>
<td>ECC Brainpool P192r1</td>
<td>brainpoolp192r1</td>
<td>1.3.36.3.3.2.8.1.1.3</td>
</tr>
<tr>
<td>ECC Brainpool P192t1</td>
<td>brainpoolp192t1</td>
<td>1.3.36.3.3.2.8.1.1.4</td>
</tr>
<tr>
<td>ECC Brainpool P224r1</td>
<td>brainpoolp224r1</td>
<td>1.3.36.3.3.2.8.1.1.5</td>
</tr>
<tr>
<td>ECC Brainpool P224t1</td>
<td>brainpoolp224t1</td>
<td>1.3.36.3.3.2.8.1.1.6</td>
</tr>
<tr>
<td>ECC Brainpool P256r1</td>
<td>brainpoolp256r1</td>
<td>1.3.36.3.3.2.8.1.1.7</td>
</tr>
<tr>
<td>ECC Brainpool P256t1</td>
<td>brainpoolp256t1</td>
<td>1.3.36.3.3.2.8.1.1.8</td>
</tr>
<tr>
<td>ECC Brainpool P320r1</td>
<td>brainpoolp320r1</td>
<td>1.3.36.3.3.2.8.1.1.9</td>
</tr>
<tr>
<td>ECC Brainpool P320t1</td>
<td>brainpoolp320t1</td>
<td>1.3.36.3.3.2.8.1.1.10</td>
</tr>
<tr>
<td>ECC Brainpool P384r1</td>
<td>brainpoolp384r1</td>
<td>1.3.36.3.3.2.8.1.1.11</td>
</tr>
<tr>
<td>ECC Brainpool P384t1</td>
<td>brainpoolp384t1</td>
<td>1.3.36.3.3.2.8.1.1.12</td>
</tr>
<tr>
<td>ECC Brainpool P512r1</td>
<td>brainpoolp512r1</td>
<td>1.3.36.3.3.2.8.1.1.13</td>
</tr>
<tr>
<td>ECC Brainpool P512t1</td>
<td>brainpoolp512t1</td>
<td>1.3.36.3.3.2.8.1.1.14</td>
</tr>
<tr>
<td>NIST B-163</td>
<td>B-163</td>
<td>1.3.132.0.15</td>
</tr>
<tr>
<td>NIST B-233</td>
<td>B-233</td>
<td>1.3.132.0.27</td>
</tr>
<tr>
<td>NIST B-283</td>
<td>B-283</td>
<td>1.3.132.0.17</td>
</tr>
<tr>
<td>NIST B-409</td>
<td>B-409</td>
<td>1.3.132.0.37</td>
</tr>
<tr>
<td>NIST B-571</td>
<td>B-571</td>
<td>1.3.132.0.39</td>
</tr>
<tr>
<td>NIST K-163</td>
<td>K-163</td>
<td>1.3.132.0.1</td>
</tr>
<tr>
<td>NIST K-233</td>
<td>K-233</td>
<td>1.3.132.0.26</td>
</tr>
<tr>
<td>NIST K-283</td>
<td>K-283</td>
<td>1.3.132.0.16</td>
</tr>
<tr>
<td>NIST K-409</td>
<td>K-409</td>
<td>1.3.132.0.36</td>
</tr>
<tr>
<td>NIST K-571</td>
<td>K-571</td>
<td>1.3.132.0.38</td>
</tr>
<tr>
<td>NIST P-192</td>
<td>P-192</td>
<td>1.2.840.10045.3.1.1</td>
</tr>
<tr>
<td>NIST P-224</td>
<td>P-224</td>
<td>1.3.132.0.33</td>
</tr>
<tr>
<td>NIST P-256</td>
<td>P-256</td>
<td>1.2.840.10045.3.1.7</td>
</tr>
<tr>
<td>NIST P-384</td>
<td>P-384</td>
<td>1.3.132.0.34</td>
</tr>
<tr>
<td>NIST P-521</td>
<td>P-521</td>
<td>1.3.132.0.35</td>
</tr>
<tr>
<td>SEC secp112r1</td>
<td>secp112r1</td>
<td>1.3.132.0.6</td>
</tr>
<tr>
<td>Name</td>
<td>IDs</td>
<td>OID</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>SEC secp112r2</td>
<td>secp112r2</td>
<td>1.3.132.0.7</td>
</tr>
<tr>
<td>SEC secp128r1</td>
<td>secp128r1</td>
<td>1.3.132.0.28</td>
</tr>
<tr>
<td>SEC secp128r2</td>
<td>secp128r2</td>
<td>1.3.132.0.29</td>
</tr>
<tr>
<td>SEC secp160k1</td>
<td>secp160k1</td>
<td>1.3.132.0.9</td>
</tr>
<tr>
<td>SEC secp160r1</td>
<td>secp160r1</td>
<td>1.3.132.0.8</td>
</tr>
<tr>
<td>SEC secp160r2</td>
<td>secp160r2</td>
<td>1.3.132.0.30</td>
</tr>
<tr>
<td>SEC secp192k1</td>
<td>secp192k1</td>
<td>1.3.132.0.31</td>
</tr>
<tr>
<td>SEC secp192r1</td>
<td>secp192r1</td>
<td>1.2.840.10045.3.1.1</td>
</tr>
<tr>
<td>SEC secp224k1</td>
<td>secp224k1</td>
<td>1.3.132.0.32</td>
</tr>
<tr>
<td>SEC secp224r1</td>
<td>secp224r1</td>
<td>1.3.132.0.33</td>
</tr>
<tr>
<td>SEC secp256k1</td>
<td>secp256k1</td>
<td>1.3.132.0.10</td>
</tr>
<tr>
<td>SEC secp256r1</td>
<td>secp256r1</td>
<td>1.2.840.10045.3.1.7</td>
</tr>
<tr>
<td>SEC secp384r1</td>
<td>secp384r1</td>
<td>1.3.132.0.34</td>
</tr>
<tr>
<td>SEC secp521r1</td>
<td>secp521r1</td>
<td>1.3.132.0.35</td>
</tr>
<tr>
<td>SEC sect113r1</td>
<td>sect113r1</td>
<td>1.3.132.0.4</td>
</tr>
<tr>
<td>SEC sect113r2</td>
<td>sect113r2</td>
<td>1.3.132.0.5</td>
</tr>
<tr>
<td>SEC sect131r1</td>
<td>sect131r1</td>
<td>1.3.132.0.22</td>
</tr>
<tr>
<td>SEC sect131r2</td>
<td>sect131r2</td>
<td>1.3.132.0.23</td>
</tr>
<tr>
<td>SEC sect163k1</td>
<td>sect163k1</td>
<td>1.3.132.0.1</td>
</tr>
<tr>
<td>SEC sect163r1</td>
<td>sect163r1</td>
<td>1.3.132.0.2</td>
</tr>
<tr>
<td>SEC sect163r2</td>
<td>sect163r2</td>
<td>1.3.132.0.15</td>
</tr>
<tr>
<td>SEC sect193r1</td>
<td>sect193r1</td>
<td>1.3.132.0.24</td>
</tr>
<tr>
<td>SEC sect193r2</td>
<td>sect193r2</td>
<td>1.3.132.0.25</td>
</tr>
<tr>
<td>SEC sect233k1</td>
<td>sect233k1</td>
<td>1.3.132.0.26</td>
</tr>
<tr>
<td>SEC sect233r1</td>
<td>sect233r1</td>
<td>1.3.132.0.27</td>
</tr>
<tr>
<td>SEC sect239k1</td>
<td>sect239k1</td>
<td>1.3.132.0.3</td>
</tr>
<tr>
<td>SEC sect283k1</td>
<td>sect283k1</td>
<td>1.3.132.0.16</td>
</tr>
<tr>
<td>SEC sect409k1</td>
<td>sect409k1</td>
<td>1.3.132.0.36</td>
</tr>
<tr>
<td>SEC sect409r1</td>
<td>sect409r1</td>
<td>1.3.132.0.37</td>
</tr>
<tr>
<td>SEC sect571k1</td>
<td>sect571k1</td>
<td>1.3.132.0.38</td>
</tr>
<tr>
<td>SEC sect571r1</td>
<td>sect571r1</td>
<td>1.3.132.0.39</td>
</tr>
<tr>
<td>X9.62 c2pnb163v1</td>
<td>c2pnb163v1</td>
<td>1.2.840.10045.3.0.1</td>
</tr>
<tr>
<td>X9.62 c2pnb163v2</td>
<td>c2pnb163v2</td>
<td>1.2.840.10045.3.0.2</td>
</tr>
<tr>
<td>X9.62 c2pnb163v3</td>
<td>c2pnb163v3</td>
<td>1.2.840.10045.3.0.3</td>
</tr>
<tr>
<td>X9.62 c2pnb176w1</td>
<td>c2pnb176w1</td>
<td>1.2.840.10045.3.0.4</td>
</tr>
<tr>
<td>X9.62 c2tnb191v1</td>
<td>c2tnb191v1</td>
<td>1.2.840.10045.3.0.5</td>
</tr>
<tr>
<td>Name</td>
<td>IDs</td>
<td>OID</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>X9.62 c2tnb191v2</td>
<td>c2tnb191v2</td>
<td>1.2.840.10045.3.0.6</td>
</tr>
<tr>
<td>X9.62 c2tnb191v3</td>
<td>c2tnb191v3</td>
<td>1.2.840.10045.3.0.7</td>
</tr>
<tr>
<td>X9.62 c2pnb208w1</td>
<td>c2pnb208w1</td>
<td>1.2.840.10045.3.0.10</td>
</tr>
<tr>
<td>X9.62 c2tnb239v1</td>
<td>c2tnb239v1</td>
<td>1.2.840.10045.3.0.11</td>
</tr>
<tr>
<td>X9.62 c2tnb239v2</td>
<td>c2tnb239v2</td>
<td>1.2.840.10045.3.0.12</td>
</tr>
<tr>
<td>X9.62 c2tnb239v3</td>
<td>c2tnb239v3</td>
<td>1.2.840.10045.3.0.13</td>
</tr>
<tr>
<td>X9.62 c2pnb272w1</td>
<td>c2pnb272w1</td>
<td>1.2.840.10045.3.0.16</td>
</tr>
<tr>
<td>X9.62 c2pnb304w1</td>
<td>c2pnb304w1</td>
<td>1.2.840.10045.3.0.17</td>
</tr>
<tr>
<td>X9.62 c2tnb359v1</td>
<td>c2tnb359v1</td>
<td>1.2.840.10045.3.0.18</td>
</tr>
<tr>
<td>X9.62 c2pnb368w1</td>
<td>c2pnb368w1</td>
<td>1.2.840.10045.3.0.19</td>
</tr>
<tr>
<td>X9.62 c2tnb431r1</td>
<td>c2tnb431r1</td>
<td>1.2.840.10045.3.0.20</td>
</tr>
<tr>
<td>X9.62 prime192v1</td>
<td>prime192v1</td>
<td>1.2.840.10045.3.1.1</td>
</tr>
<tr>
<td>X9.62 prime192v2</td>
<td>prime192v2</td>
<td>1.2.840.10045.3.1.2</td>
</tr>
<tr>
<td>X9.62 prime192v3</td>
<td>prime192v3</td>
<td>1.2.840.10045.3.1.3</td>
</tr>
<tr>
<td>X9.62 prime239v1</td>
<td>prime239v1</td>
<td>1.2.840.10045.3.1.4</td>
</tr>
<tr>
<td>X9.62 prime239v2</td>
<td>prime239v2</td>
<td>1.2.840.10045.3.1.5</td>
</tr>
<tr>
<td>X9.62 prime239v3</td>
<td>prime239v3</td>
<td>1.2.840.10045.3.1.6</td>
</tr>
<tr>
<td>X9.62 prime256v1</td>
<td>prime256v1</td>
<td>1.2.840.10045.3.1.7</td>
</tr>
</tbody>
</table>
Appendix C – 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
