Legion of the Bouncy Castle Inc.
BC-FJA (Bouncy Castle FIPS Java API)

User Guide

Version: 2.0.0 Date: 09/17/24

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.

The following people and organisations also contributed financially to this project.

Organisations: Intertrust Technologies, Lobster GmbH, PrimeKey Solutions AB, and Yozons, Inc.

Individuals: Rui Abreu, Tim Brown, Joe Brutto, Drew Carey Buglione, Andrew Carroll, Natalie Marie Ellis, Barry Faba, Khansae1992, and Brian Phelps.

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

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

Apurva Varalikar
Laboratory Manager,
Intertek
18504 Office Park Dr
Montgomery Village, MD 20886
United States of America

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 Java 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.

This document assumes you are familiar with Java and at least have some familiarity with the Java Cryptography Architecture (the JCA) and the Java Cryptography Extension (the JCE).

1.1 About the FIPS Validation

The BC FIPS jar has been designed and implemented to meet FIPS 140-2, Level 1 requirements. As detailed in the security policy, providing you use the BC FIPS jar as is, the validation of the module applies to its use on the Java SE Runtime Environment 7 and the Java SE Runtime Environment 8. This is possible because the FIPS Implementation Guidance (the IG) allows affirmation of compliance providing a software module is run on the same single user environment as it was originally tested on, even if what underlies the single user environment is different (see section G.5). In our case the Java Virtual Machine represents the single user environment.

1.2 Use with the BC PKIX, OpenPGP (PG), and SMIME APIs

Separate JARs are provided for the additional Bouncy Castle APIs, these are the same as the regular ones, however the lightweight BC support classes have been removed and some internal version numbers have been changed in order to allow the BC FIPS jar and the jars for the additional APIs to work together correctly in OSGI compliant containers. The BC FIPS jar is compatible with the extra Bouncy Castle APIs from BC 1.53.

1.3 Formats and Restrictions on Input Data and Control Information

Data that can be entered into the module is primarily limited by the constraints imposed on the length of byte arrays in Java (2^{32} bytes) or in the case of structured data such as signatures, keys, and certificates, the additional limitations imposed by the relevant ASN.1 based standards applicable to the algorithms and the constraints specified by the FIPS standards applicable to the algorithms.

For ASN.1 encoded objects, where DER encoding is not required, BER, or some subset of BER, encoding can also be used.

In Java public, private, and secret keys all implement `Serializable` as do public key certificates. As this is the case the module is also capable of loading keys and certificates as previously serialised objects where it is otherwise FIPS compliant to do so.

Symmetric Secret Keys can also be entered as raw byte arrays, appropriate to the key size, through use of the Java `javax.crypto.spec.SecretKeySpec KeySpec` class.

Control information for configuration is passed to the module via system/security properties or Java permissions. The accepted values and inputs for these can be found in Appendix A and Appendix B of this user guide.

1.4 Using the Module in an Application for FIPS compliance

In order to meet FIPS requirements, if you develop an application with this module you need to make sure you follow the security policy provided with the module. The security policy represents the official reviewed document of how to make use of this module properly – it is the final word on the topic. If you need help following the security policy, or wish to confirm you are doing so correctly, we are happy to certify this via review of your application. If you are a support contract holder this is regarded as a consulting assignment, but you can use your consulting time towards it, and benefit from our discounted rate if any additional time is required.

1.5 Commercial Support

Commercial support contracts for this software are available from Crypto Workshop and also help support this project. The commercial support program also entitles you to early access to the next version of the FIPS module, any updates as soon as they are made, and the opportunity to request changes and enhancements. For further information about commercial support please email us info@cryptoworkshop.com.

1.6 If you need Changes or Enhancements

If the certification does not cover your Java runtime environment, or you need a different, possibly smaller, set of algorithms, we can help you. The module has been designed to subset and extend easily, and we are the best placed to do it – after all we designed it, and we would hope that the presence of this document shows we have definitely got the experience and we deliver.

We do not object to being involved in private validations either. We accept that there are sometimes situations where a private validation may make good sense for an organisation, either for competitive advantage, marketing, or risk management reasons. Often enhancements and sub-setting can be done using fixed rate contracts to help manage costs, and, again, the consulting time available in support contracts may be used towards this as well. So if you need further work around the module, please contact us and we will let you know what can be done.

2 Installation

Up till Java 8, the bc-fips jar can be installed in either jre/lib/ext for your JRE/JDK, or in many cases, on the general class path defined for the application you are running. In the event you do install on the bc-fips jar on the general class path be aware that sometimes the system class loader may make it impossible to use the classes in the bc-fips jar without causing an exception depending on how elements of the JCA/JCE are loaded elsewhere in the application. After Java 8 it is better to use the “--module-path” option to load the bc-fips jar and its associated libraries into your JVM for your application.

Note: in any of the cases below it is likely you will need to install the unrestricted policy files for the JCE in order to make full use of the algorithm set in the bc-fips jar.

2.1 Provider installation into the JRE

Once the bcfips jar is installed, the provider class BouncyCastleFipsProvider may need to be installed if it is required in the application globally.

For Java 8, installation of the provider can be done statically in the JVM by adding it to the provider definition to the java.security file in in the jre/lib/security directory for your JRE/JDK. For later versions of the JVM the provider can installed on start up by making use of “--module-path”, for example:

```
java –module-path <fips-jars-directory> java_application_main
```

where `fips-jars-directory` is a directory containing the BC FIPS jars you wish to use.

Note: with newer JVMs there are increasing problems with using the older class path techniques to add the FIPS jars, especially the provider jar, generally resulting in failures due to class annotation issues.

The provider can also be added during execution. If you wish to add the provider to the JVM globally during execution you can add the following imports to your code:

```
import java.security.Security
import org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
```

Then insert the line

```
Security.addProvider(new BouncyCastleFipsProvider())
```

The provider can then be used by referencing the name “BCFIPS”, for example:

```
Cipher c = Cipher.getInstance(“AES/CBC/PKCS5Padding”, “BCFIPS”);
```

Alternately if you do not wish to install the provider globally, but use it locally instead, it is possible to pass the provider to the getInstance() method on the JCA/JCE class you are creating an instance of. For example:

```
Cipher c = Cipher.getInstance(“AES/CBC/PKCS5Padding”,
                                    new BouncyCastleFipsProvider());
```

Note: if the provider object is created in FIPS approved mode then only FIPS approved mode algorithms will be available.

2.1.1 Use with JAVA_TOOL_OPTIONS for Java tools

It has been found that simply using the class path options will usually result in failures in the Java tool set, apparently due to class annotation issues connected with loading the BCFIPS provider, however it has also been found that if a JAVA_TOOL_OPTIONS environment variable which provides the module path details to the jars containing the BC distribution e.g:

```
JAVA_TOOL_OPTIONS=--module-path=<fips-jars-directory>
```

is used in connection with a tool such as the keytool then most things will work as expected, even in approved only mode.

2.2 Provider installation as a provider for the BCJSSE

The provider no longer supports the experimental FIPS mode that used to be available in the Oracle JSSE. This is partly because it has laregly been a no-op since Java 9, but also that the internal classes required to support it originally are restricted and in some cases now unavaliable.

If FIPS compliance is required for TLS, the BCJSSE needs to be installed as well.

Further information on the BCJSSE is available in the current version of:

“Legion of the Bouncy Castle Inc. Java (D)TLS API and JSSE Provider User Guide”

but, in brief, the BCJSSE, the minimal static provider configuration to support the JSSE is:

```
security.provider.1=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
security.provider.2=org.bouncycastle.jsse.provider.BouncyCastleJsseProvider fips:BCFIPS
security.provider.3=sun.security.provider.Sun
```

When using the JSSE in FIPS mode you will probably also find it requires the server and client private keys to be coming from a key store supported by the BCFIPS provider. Use the BCFKS for the key store format.

2.3 Provider configuration

The provider can also take a string as a constructor argument, either via the java.security file definition or when the constructor is called at runtime.

At the moment the configuration string is limited to setting the DRBG. The configuration string must always start with “C:” and finish with “ENABLE{ALL};”. The command for setting the actual DRBG to be used is DEFRND, so a configuration specifying the use of a SHA-256 DRBG would look like:

```
new BouncyCastleFipsProvider(“C:DEFRND[SHA256];ENABLE{ALL};”);
```

The DRBG type string is based on the available DRBGs from SP 800-90A. The possible values are: “SHA1”, “SHA224”, “SHA256”, “SHA384”, “SHA512”, “SHA512(224)”, “SHA512(256)”, “HMACSHA1”, “HMACSHA224”, “HMACSHA256”, “HMACSHA384”, “HMACSHA512”, “CTRAES128”, “CTRAES192”, “CTRAES256”, and “CTRDESEDE”.

Note: In order to make the default DRBG suitable for key generation, the default DRBG is configured to be prediction resistant and this can strain the JVMs entropy source especially if hardware RNG is not available.

In situations where the amount of entropy is constrained the default DRBG for the provider can be configured to use an entropy pool based on a SHA-512 SP 800-90A DRBG. To configure this use:

```
“C:HYBRID;ENABLE{ALL};”
```

or include the string `"HYBRID;"` in the previous command string setting the DRBG. After initial seeding the entropy pool will start a reseeding thread which it will begin polling once 20 samples have been taken since the last seeding and will do a reseed as soon as new entropy bytes are returned. The thread responsible for collecting entropy will show up in the thread dumps as the “BC FIPS Entropy Daemon”. The entropy thread will only run if HYBRID mode has been specified.

Prediction resistance can also be turned off by specifying false in the DEFRND parameters. e.g.

```
“C:DEFRND[SHA256,false];ENABLE{ALL};”
```

or

```
“C:DEFRND[false];ENABLE{ALL};”
```

2.4 Provider Verification

The module checksum, functionality, and versioning can be confirmed by executing the command:

```
java -cp bc-fips-2.0.0.jar org.bouncycastle.util.DumpInfo
```

which should display:

```
Version Info: BouncyCastle Security Provider (FIPS edition) v2.0.0
FIPS Ready Status: READY
Module SHA-256 HMAC:
164c8ae41945cb85fdc65666fc4de7301a65d29659ecd455ee5199c7d42d107e
```

Indicating the jar represents the software release BC-FJA 2.0.0, that it has successfully passed all its startup tests, and that the software release is confirmed to have a HMAC of:

`164c8ae41945cb85fdc65666fc4de7301a65d29659ecd455ee5199c7d42d107e`

3 Cipher Algorithms (Symmetric)

Cipher availability and the modes of the ciphers available depends on whether the module is running a 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	Low-level Class	Modes	JCE Cipher Name
AES	FipsAES	ECB, CBC, CFB8, CFB128, OFB, CTR, CCM, GCM, FF1	AES
TripleDES	FipsTripleDES	ECB, CBC, CFB8, CFB64, OFB, CTR	DESede

3.2 Available in General Operation

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

Algorithm	Low-level Class	Modes	JCE Cipher Name
AES	AES	EAX, OCB, OpenPGPCFB, FF3-1	AES
ARC4	ARC4	N/A	ARC4, RC4
Blowfish	Blowfish	CBC, CFB8, CFB64, CTR, EAX, ECB, OFB, OpenPGPCFB	Blowfish
Camellia	Camellia	CBC, CCM, CFB8, CFB128, CTR, EAX, ECB, GCM, OCB, OFB	Camellia
CAST5	CAST5	CBC, CFB8, CFB64, CTR, EAX, ECB, OFB, OpenPGPCFB	CAST5
ChaCha20	ChaCha20	N/A	ChaCha20
ChaCha20-Poly1305	ChaCha20	N/A	ChaCha20-Poly1305
DES	DES	CBC, CFB8, CFB64, CTR, EAX, ECB, OFB, OpenPGPCFB	DES
GOST28147	GOST28147	CBC, CFB8, CFB64, CTR, EAX, ECB, GCFB, GOFB, OFB	GOST28147
IDEA	IDEA	CBC, CFB8, CFB64, CTR, EAX, ECB, OFB,	IDEA

Algorithm	Low-level Class	Modes	JCE Cipher Name
		OpenPGPCFB	
RC2	RC2	CBC, CFB8, CFB64, CTR, EAX, ECB, OFB	RC2
SEED	SEED	CBC, CCM, CFB8, CFB128, CTR, EAX, ECB, GCM, OCB, OFB	SEED
Serpent	Serpent	CBC, CCM, CFB8, CFB128, CTR, EAX, ECB, GCM, OCB, OFB	Serpent
SHACAL-2	SHACAL2	CBC, CFB8, CFB256, CTR, EAX, ECB, OFB	SHACAL-2, SHACAL2
Triple-DES	TripleDES	EAX, OpenPGPCFB	DESede
Twofish	Twofish	CBC, CCM, CFB8, CFB128, CTR, EAX, ECB, GCM, OCB, OFB	Twofish

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	JCE Names
ECB	NONE, PKCS7 (PKCS5), ISO10126-2, X9.23, ISO7816-4, TBC	NoPadding, PKCS7Padding (PKCS5Padding), ISO10126-2Padding, X9.23Padding, ISO7816-4Padding, TBCPadding
CBC	NONE, PKCS7 (PKCS5), ISO10126-2, X9.23, ISO7816-4, TBC, CS1, CS2, CS3	NoPadding, PKCS7Padding (PKCS5Padding), ISO10126-2Padding, ISO7816-4Padding, X9.23Padding, TBCPadding, CS1Padding, CS2Padding, CS3Padding

In addition to the use of PKCS5Padding for PKCS7Padding, ISO9797-1Padding can also be used as an alias for ISO7816-4Padding.

While the padding types named after standards are 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.

3.3 Examples

The low-level examples make use of the following utility methods:

```
static byte[] encryptBytes(
  FipsOutputEncryptor outputEncryptor, byte[] plainText) throws IOException
 {
     ByteArrayOutputStream bOut = new ByteArrayOutputStream();
     CipherOutputStream encOut = outputEncryptor.getEncryptingStream(bOut);
```

```
        encOut.update(plainText);

        encOut.close();

        return bOut.toByteArray();
    }
```

and:

```
     static byte[] decryptBytes(FipsInputDecryptor inputDecryptor,
                                byte[] cipherText) throws IOException
    {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        InputStream encIn = inputDecryptor.getDecryptingStream(
                                   new ByteArrayInputStream(cipherText));
        int ch;

        while ((ch = encIn.read()) >= 0)
        {
            bOut.write(ch);
        }

        return bOut.toByteArray();
    }
```

3.3.1 AES Encryption using CBC and PKCS5/7Padding

```
   // ensure a FIPS DRBG in use.
   CryptoServicesRegistrar.setSecureRandom(
           FipsDRBG.SHA512_HMAC.fromEntropySource(
                new BasicEntropySourceProvider(new SecureRandom(), true))
           .build(null, true));

   byte[] iv = new byte[16];

   CryptoServicesRegistrar.getSecureRandom().nextBytes(iv);

   FipsSymmetricKeyGenerator<SymmetricSecretKey>  keyGen =
                new FipsAES.KeyGenerator(128,
                            CryptoServicesRegistrar.getSecureRandom());

   SymmetricSecretKey key = keyGen.generateKey();

   FipsSymmetricOperatorFactory<FipsAES.Parameters> fipsSymmetricFactory =
                                             new FipsAES.OperatorFactory();

   FipsOutputEncryptor<FipsAES.Parameters> outputEncryptor =
      fipsSymmetricFactory.createOutputEncryptor(key,
                      FipsAES.CBCwithPKCS7.withIV(iv));

   byte[] output = encryptBytes(outputEncryptor, new byte[16]);

   FipsInputDecryptor<FipsAES.Parameters> inputDecryptor =
      fipsSymmetricFactory.createInputDecryptor(key,
                      FipsAES.CBCwithPKCS7.withIV(iv));

   byte[] plain = decryptBytes(inputDecryptor, output);
```

3.3.2 JCE AES Encryption using CBC and PKCS5/7Padding

A note on this example: it will use the default provider DRBG if nothing has already been configured.

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BCFIPS");

keyGen.init(256);
SecretKey aesKey = keyGen.generateKey();

byte[] data = Hex.decode("000102030405060708090A0B0C0D0E0F");

Cipher enc = Cipher.getInstance("AES/CBC/PKCS5Padding", "BCFIPS");
enc.init(Cipher.ENCRYPT_MODE, aesKey);

byte[] encrypted = enc.doFinal(data);
byte[] iv = enc.getIV();

Cipher dec = Cipher.getInstance("AES/CBC/PKCS5Padding", "BCFIPS");
dec.init(Cipher.DECRYPT_MODE, aesKey, new IvParameterSpec(iv));


byte[] plain = dec.doFinal(encrypted);
```

3.3.3 JCE AES Encryption 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.

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BCFIPS");

keyGen.init(256);
SecretKey aesKey = keyGen.generateKey();

byte[] data = Hex.decode("000102030405060708090A0B0C0D0E0F");
byte[] assocData = Hex.decode("10111213141516171819");
byte[] nonce = Hex.decode("202122232425262728292a2b2c");

Cipher enc = Cipher.getInstance("AES/GCM/NoPadding", "BCFIPS");
enc.init(Cipher.ENCRYPT_MODE, aesKey,
                       new AEADParameterSpec(nonce, 96, assocData));

byte[] encrypted = enc.doFinal(data);

Cipher dec = Cipher.getInstance("AES/GCM/NoPadding", "BCFIPS");
dec.init(Cipher.DECRYPT_MODE, aesKey,
                     new AEADParameterSpec(nonce, 96, assocData));


byte[] plain = dec.doFinal(encrypted);
```

3.3.4 JCE AES Encryption using CTR a short IV

CTR mode can be used either with an internal counter or a provided one. An internal counter that limits the number of blocks that can be processed can be configured by providing the cipher with an IV which is less than the algorithm's block size. This will configure a limiting counter that is sizeof(cipher block) – sizeof(IV) bytes long. If the IV provided to CTR mode is the length of the block size of the algorithm the module will assume you are using an external (to the module) counter and simply add 1 to the IV as each block goes through.

If you are using a limiting counter, over, or under, flowing the counter will result in an IllegalStateException. The maximum size for a limiting counter is 8 bytes. In the example below an limiting counter of 4 bytes will be used as the IV is 12 bytes long, and the block size of AES is 16 bytes.

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BCFIPS");
keyGen.init(256);
SecretKey aesKey = keyGen.generateKey();

byte[] data = Hex.decode("000102030405060708090A0B0C0D0E0F");
byte[] iv = Hex.decode("000102030405060708090a0b");

Cipher enc = Cipher.getInstance("AES/CTR/NoPadding", "BCFIPS");

enc.init(Cipher.ENCRYPT_MODE, aesKey, new IvParameterSpec(iv));

byte[] encrypted = enc.doFinal(data);

Cipher dec = Cipher.getInstance("AES/GCM/NoPadding", "BCFIPS");

dec.init(Cipher.DECRYPT_MODE, aesKey, new IvParameterSpec(iv));


byte[] plain = dec.doFinal(encrypted);
```

3.3.5 JCE AES Format Preserving Encryption using FF1

The following code fragment shows an example of using Format Preserving Encryption (FPE) using the FF1 algorithm. The example makes use of the BC class FPEParameterSpec to pass in the parameters for the FPE algorithm.

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BCFIPS");
keyGen.init(256);
SecretKey aesKey = keyGen.generateKey();

Coming Soon!
```

4 Cipher Algorithms (Public Key)

There are no direct public/private key ciphers available in approved mode. Available ciphers are restricted to use for key wrapping and key transport, see section 7 and section 8 for details.

4.1 Available in General Operation

Cipher	Low-level Class	JCE Name	Modes
ELGAMAL	ElGamal	ElGamal	NONE, ECB
RSA	RSA	RSA	NONE, ECB

4.2 Paddings Available

Public key algorithms support the following padding mechanisms:

- NoPadding
- OAEPwithSHA-1andMGF1Padding – aliases: OAEPwithSHA1andMGF1Padding, OAEPPadding.
- OAEPwithSHA-224andMGF1Padding – aliases: OAEPwithSHA224andMGF1Padding
- OAEPwithSHA-256andMGF1Padding – aliases: OAEPwithSHA256andMGF1Padding
- OAEPwithSHA-384andMGF1Padding – aliases: OAEPwithSHA384andMGF1Padding
- OAEPwithSHA-512andMGF1Padding – aliases: OAEPwithSHA512andMGF1Padding
- PKCS1Padding

4.3 Examples

4.3.1 JCE RSA with PKCS1 Padding

```
public byte[] pkcs1Encrypt(RSAPublicKey pubKey, byte[] data)
{
    Cipher c = Cipher.getInstance(“RSA/NONE/PKCS1Padding”, “BCFIPS”);

    c.init(Cipher.ENCRYPT_MODE, pubKey);

    return c.doFinal(data);
}
```

4.3.1 JCE ElGamal with OAEP SHA-256 Padding

```
public byte[] oaepEncrypt(DHPublicKey pubKey, byte[] data)
{
    Cipher c = Cipher.getInstance(
                     “ELGAMAL/NONE/OAEPwithSHA256andMGF1Padding”, “BCFIPS”);
```

```
    c.init(Cipher.ENCRYPT_MODE, pubKey);

    return c.doFinal(data);
}
```

5 Key Agreement Algorithms

Key agreement algorithms are available based around both elliptic curve and the regular prime fields. The basic primitives are provided in the low-level classes together with support for the use of a digest or a PRF as detailed in NIST SP 800-56C.

At the JCE level the provider supports key confirmation as well. The use of key confirmation is signalled by an algorithm string containing a “/” being passed to KeyAgreement.generateSecret() which returns a new type of SecretKey - an AgreedKeyWithMacKey (see example 5.3.6 for details). The AgreedKeyWithMacKey has an additional method on it getMacKey() which returns a key that can be used for the MAC calculation in the confirmation step. In line with NIST recommendations the MAC key can be pro-actively zeroized.

5.1 Available in Approved Mode Operation

Key Agreement Method	KDF	Low-level Class	JCE Name
DH		FipsDH	DH
DHwithSHA1KDF	X9.63(SHA-1)		DHwithSHA1KDF
DHwithSHA224KDF	X9.63(SHA-224)		DHwithSHA224KDF
DHwithSHA256KDF	X9.63(SHA-256)		DHwithSHA256KDF
DHwithSHA384KDF	X9.63(SHA-384)		DHwithSHA384KDF
DHwithSHA512KDF	X9.63(SHA-512)		DHwithSHA512KDF
DHwithSHA512(224)KDF	X9.63 (SHA-512(224))		DHwithSHA512(224)KDF
DHwithSHA512(256)KDF	X9.63 (SHA-512(256))		DHwithSHA512(256)KDF
DHwithSHA1CKDF	Concatenation (SHA-1)		DHwithSHA1CKDF
DHwithSHA224CKDF	Concatenation (SHA-224)		DHwithSHA224CKDF
DHwithSHA256CKDF	Concatenation (SHA-256)		DHwithSHA256CKDF
DHwithSHA384CKDF	Concatenation (SHA-384)		DHwithSHA384CKDF
DHwithSHA512CKDF	Concatenation (SHA-512)		DHwithSHA512CKDF
DHwithSHA512(224)CKDF	Concatenation (SHA-512(224))		DHwithSHA512(224)CKDF
DHwithSHA512(256)CKDF	Concatenation (SHA-512(256))		DHwithSHA512(256)CKDF
MQV		FipsDH	
MQVwithSHA1KDF	X9.63(SHA-1)		MQVwithSHA1KDF
MQVwithSHA224KDF	X9.63(SHA-224)		MQVwithSHA224KDF

Key Agreement Method	KDF	Low-level Class	JCE Name
MQVwithSHA256KDF	X9.63(SHA-256)		MQVwithSHA256KDF
MQVwithSHA384KDF	X9.63(SHA-384)		MQVwithSHA384KDF
MQVwithSHA512KDF	X9.63(SHA-512)		MQVwithSHA512KDF
MQVwithSHA512(224)KDF	X9.63 (SHA-512(224))		MQVwithSHA512(224)KDF
MQVwithSHA512(256)KDF	X9.63 (SHA-512(256))		MQVwithSHA512(256)KDF
MQVwithSHA1CKDF	Concatenation (SHA-1)		MQVwithSHA1CKDF
MQVwithSHA224CKDF	Concatenation (SHA-224)		MQVwithSHA224CKDF
MQVwithSHA256CKDF	Concatenation (SHA-256)		MQVwithSHA256CKDF
MQVwithSHA384CKDF	Concatenation (SHA-384)		MQVwithSHA384CKDF
MQVwithSHA512CKDF	Concatenation (SHA-512)		MQVwithSHA512CKDF
MQVwithSHA512(224)CKDF	Concatenation (SHA-512(224))		MQVwithSHA512(224)CKDF
MQVwithSHA512(256)CKDF	Concatenation (SHA-512(256))		MQVwithSHA512(256)CKDF
ECDH		FipsEC	ECDH
ECDHwithSHA1KDF	X9.63(SHA-1)		ECDHwithSHA1KDF
ECDHwithSHA224KDF	X9.63(SHA-224)		ECDHwithSHA224KDF
ECDHwithSHA256KDF	X9.63(SHA-256)		ECDHwithSHA256KDF
ECDHwithSHA384KDF	X9.63(SHA-384)		ECDHwithSHA384KDF
ECDHwithSHA512KDF	X9.63(SHA-512)		ECDHwithSHA512KDF
ECCDH		FipsEC	ECCDH
ECCDHwithSHA1KDF	X9.63(SHA-1)		ECCDHwithSHA1KDF
ECCDHwithSHA224KDF	X9.63(SHA-224)		ECCDHwithSHA224KDF
ECCDHwithSHA256KDF	X9.63(SHA-256)		ECCDHwithSHA256KDF
ECCDHwithSHA384KDF	X9.63(SHA-384)		ECCDHwithSHA384KDF
ECCDHwithSHA512KDF	X9.63(SHA-512)		ECCDHwithSHA512KDF
ECCDHwithSHA1CKDF	Concatenation (SHA-1)		ECCDHwithSHA1KDF
ECCDHwithSHA224CKDF	Concatenation (SHA-224)		ECCDHwithSHA224CKDF
ECCDHwithSHA256CKDF	Concatenation (SHA-256)		ECCDHwithSHA256CKDF

Key Agreement Method	KDF	Low-level Class	JCE Name
ECCDHwithSHA384CKDF	Concatenation (SHA-384)		ECCDHwithSHA384CKDF
ECCDHwithSHA512CKDF	Concatenation (SHA-512)		ECCDHwithSHA512CKDF
ECCDHwithSHA512(224)CKDF	Concatenation (SHA-512(224))		ECCDHwithSHA512(224)CKDF
ECCDHwithSHA512(256)CKDF	Concatenation (SHA-512(256))		ECCDHwithSHA512(256)CKDF
ECMQV		FipsEC	ECMQV
ECMQVwithSHA1KDF	X9.63(SHA-1)		ECMQVwithSHA1KDF
ECMQVwithSHA224KDF	X9.63(SHA-224)		ECMQVwithSHA224KDF
ECMQVwithSHA256KDF	X9.63(SHA-256)		ECMQVwithSHA256KDF
ECMQVwithSHA384KDF	X9.63(SHA-384)		ECMQVwithSHA384KDF
ECMQVwithSHA512KDF	X9.63(SHA-512)		ECMQVwithSHA512KDF
ECMQVwithSHA1CKDF	Concatenation (SHA-1)		ECMQVwithSHA1KDF
ECMQVwithSHA224CKDF	Concatenation (SHA-224)		ECMQVwithSHA224CKDF
ECMQVwithSHA256CKDF	Concatenation (SHA-256)		ECMQVwithSHA256CKDF
ECMQVwithSHA384CKDF	Concatenation (SHA-384)		ECMQVwithSHA384CKDF
ECMQVwithSHA512CKDF	Concatenation (SHA-512)		ECMQVwithSHA512CKDF
ECMQVwithSHA512(224)CKDF	Concatenation (SHA-512(224))		ECMQVwithSHA512(224)CKDF
ECMQVwithSHA512(256)CKDF	Concatenation (SHA-512(256))		ECMQVwithSHA512(256)CKDF

Note: in approved mode ECDH will only work if the cofactor of the EC domain parameters being used is 1.

5.2 Available in General Operation

There are no additional key agreement algorithms offered in general operation, however ECDH can be used even with curves that do not have a cofactor of 1.

5.3 Examples

The following examples are all for elliptic curve, other than Cofactor-Diffie-Hellman (CDH), they can equally be applied to FipsDH, or the algorithm “DH” in the case of the JCE.

5.3.1 Basic Agreement

```
public byte[] basicAgreement(
```

```
    AsymmetricECPrivateKey ourKey,
    AsymmetricECPublicKey otherPartyKey)
{
    FipsEC.DHAgreementFactory agreementFactory =
                                    new FipsEC.DHAgreementFactory();
    FipsAgreement<FipsEC.AgreementParameters> agree =
                     agreementFactory.createAgreement(ourKey, FipsEC.DH);

    return agree.calculate(otherPartyKey);
}
```

5.3.2 Basic Agreement with Cofactor

```
public byte[] basicAgreementWithCofactor(
    AsymmetricECPrivateKey ourKey,
    AsymmetricECPublicKey otherPartyKey)
{
    FipsEC.DHAgreementFactory agreementFactory =
                                    new FipsEC.DHAgreementFactory();
    FipsAgreement<FipsEC.AgreementParameters> agree =
                     agreementFactory.createAgreement(ourKey, FipsEC.CDH);

    return agree.calculate(otherPartyKey);
}
```

5.3.3 Basic Agreement with PRF

```
public byte[] basicAgreementWithPRF(
    AsymmetricECPrivateKey ourKey,
    AsymmetricECPublicKey otherPartyKey,
    FipsKDF.PRF prfAlg,
    byte[] salt)
{
    FipsEC.DHAgreementFactory agreementFactory =
                                    new FipsEC.DHAgreementFactory();
    FipsEC.AgreementParameters params =
                         new FipsEC.AgreementParameters(FipsEC.DH, prfAlg, salt);
    FipsAgreement<FipsEC.AgreementParameters> agree =
                     agreementFactory.createAgreement(ourKey, params);

    return agree.calculate(otherPartyKey);
}
```

5.3.4 JCE Basic Agreement

```
public byte[] basicAgreement(
    ECPrivateKey ourKey,
    ECPublicKey otherPartyKey)
```

```
{
    KeyAgreement agree = KeyAgreement.getInstance("ECDH", "BCFIPS");

    agree.init(ourKey);

    agree.doPhase(otherPartyKey, true);

    return agree.generateSecret();
}
```

5.3.5 JCE One-pass MQV

MQV requires the use of ephemeral keys, in the case of one-pass MQV the other party's static key is assumed to be their ephemeral key.

```
public byte[] onePassMQV(
    ECPrivateKey ourKey,
    KeyPair      ourEphemeralKeys,
    ECPublicKey  otherPartyKey)
{
    KeyAgreement agree = KeyAgreement.getInstance("ECMQVwithSHA256KDF", "BCFIPS");

    agree.init(ourKey, new MQVParameterSpec(ourEphemeralKeys, otherPartyKey);

    agree.doPhase(otherPartyKey, true);

    return agree.generateSecret();
}
```

5.3.6 JCE One-pass MQV with key confirmation

The following sample returns a provider specific SecretKey type, a AgreedKeyWithMacKey, which allows a user to do key agreement with key confirmation. For all intents and purposes an AgreedKeyWithMacKey behaves like a secret key, but it also has an extra method on it for returning the MAC key associated with the agreement.

This sample also provides an example of how you can specify a key size and an algorithm with the BC FIPS provider. In the example below the the AgreedKeyWithMacKey will have a MAC key of 128 bits (algorithm name CMAC) and an encryption key of 256 bits (algorithm name AES).

```
public AgreedKeyWithMacKey onePassMQV(
    ECPrivateKey ourKey,
    KeyPair      ourEphemeralKeys,
    ECPublicKey  otherPartyKey)
{
    KeyAgreement agree = KeyAgreement.getInstance("ECMQVwithSHA256KDF", "BCFIPS");

    agree.init(ourKey, new MQVParameterSpec(ourEphemeralKeys, otherPartyKey);
```

```
    agree.doPhase(otherPartyKey, true);

    return (AgreedKeyWithMacKey)agree.generateSecret("CMAC[128]/AES[256]");
}
```

6 Key Derivation Functions

Most of the current KDFs in SP 800-135 and SP 800-108 are supported. In addition Scrypt is available when the Java module is not running in approved-only mode.

6.1 Available in Approved Mode Operation

Derivation Method	Low-level Class
Counter Mode	FipsKDF
Feedback Mode	FipsKDF
Double Pipeline Iteration Mode	FipsKDF
TLS 1.0	FipsKDF
TLS 1.1	FipsKDF
TLS 1.2	FipsKDF
SSH	FipsKDF
X 9.63	FipsKDF
Concatenation	FipsKDF
IKEv2	FipsKDF
SRTP	FipsKDF

6.2 Available in General Mode Operation

Derivation Method	Low-level Class
SCrypt	KDF

6.3 Examples

KDFs are currently not directly exposed in the JCE/JCA layer, although they are made use of internally by algorithms like Diffe-Hellman and also by the JSSE. They can be invoked directly using the low-level API.

6.3.1 Feedback Mode

An example of feedback mode using AES_CMAC as the PRF and an 8 bit counter. Such a KDF could be used to generate up to 4096 bytes (being 256 * 16 (the block size of AES_CMAC)). In the example below it is being used to generate 128 bits.

```
KDFOperatorFactory kdfOperatorFactory =
             new FipsKDF.FeedbackModeFactory();

KDFCalculator<FipsKDF.FeedbackModeParameters> kdfCalculator =
                kdfOperatorFactory.createKDFCalculator(
                    FipsKDF.FEEDBACK_MODE.withPRF(FipsKDF.PRF.AES_CMAC)
```

```
    .withRAndLocation(8, FipsKDF.CounterLocation.BEFORE_ITERATION_DATA)
    .using(KI, IV, FixedInputData));

byte[] out = new byte[16];

kdfCalculator.generateBytes(out);
```

6.3.2 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.

```
FipsKDFOperatorFactory<FipsKDF.AgreementKDFParameters> kdfOpt =
                         new FipsKDF.X963OperatorFactory();

KDFCalculator kdfCalculator = kdfOpt.createKDFCalculator(FipsKDF.X963.using(Z));


byte[] out = new byte[16];

kdfCalculator.generateBytes(out);
```

7 Key Stores

The BC FIPS provider supports two types of KeyStore formats.

Key Store Type	Format	JCA Name
PKCS12	ASN.1 BER/DER	PKCS12
BCFKS	DER	BCFKS

The PKCS12 key store is **not available in approved-mode** of operation due to the algorithms required for PBE key generation in the PKCS#12 standard. The PKCS12 key store supports the following variations:

- PKCS12-3DES-3DES: the default, uses Triple-DES for any encryption task.
- PKCS12-3DES-40RC2: the traditional one, uses Triple-DES for private keys and 40 bit RC2 for certificate protection.

The BCFKS key store is designed to be FIPS compliant. It is available in approved-mode operation and is also capable of storing some secret key types in addition to public/private keys and certificates. The BCFKS key store uses PBKDF2 with HMAC SHA512 for password to key conversion and AES CCM for encryption. Passwords are encoded for conversion into keys using PKCS#12 format (as in each 16 bit character is converted into 2 bytes).

7.1 Examples

7.1.1 BCFKS key store

This code fragment shows storage of a private key and its certificate chain, as well as the storage of an AES key and a Triple-DES key.

```
PrivateKey key = … // asymmetric private key
X509Certificate[] certs = … // certificate chain for corresponding public key


KeyStore store1 = KeyStore.getInstance("BCFKS", "BCFIPS");
store.load(null, null);
store.setKeyEntry("privkey", key, passwd, certs);

SecretKeySpec aesKey = new SecretKeySpec(
                        Hex.decode("000102030405060708090a0b0c0d0e0f"), "AES");
store.setKeyEntry("secret1", aesKey, "secretPwd1".toCharArray(), null);

SecretKeySpec edeKey = new SecretKeySpec(
                      Hex.decode("01010202040407070808b0b0d0d0e0e"), "DESede");
store.setKeyEntry("secret2", edeKey, "secretPwd2".toCharArray(), null);


ByteArrayOutputStream bOut = new ByteArrayOutputStream();
store.store(bOut, passwd);
```

8 Key Transport Algorithms

The BC FIPS provider supports the following two key transport methods:

Algorithm Name	Low-level Class	JCE Name
RSA-KTS-KEM-KWS	FipsRSA	RSA-KTS-KEM-KWS
RSA-KTS-OAEP	FipsRSA	RSA-KTS-OAEP

Both algorithms are supported in the JCE using SecretKeyFactory implementations. Owing to it being used in RFC 5990, RSA-KTS-KEM-KWS is also support as a Cipher, although when used under those circumstances it is not currently possible to use it with key confirmation.

8.1 Examples

8.1.1 JCE KTS-KEM-KWS

This is an example of RSA-KTS-KEM-KWS as used in RFC 5990. In this case we are assuming the wrapped key is an AES key. The wrapping algorithm for RSA-KTS-KEM-KWS to use is specified via a KTSParameterSpec. In this example the KEM process is used to calculate an AES-256 key to wrap the AES key held in the SecretKey object.

```
public byte[] wrapKey(RSAPublicKey publicKey, SecretKey secretKey)
    throws Exception
{
    Cipher wrapper = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");
    KTSParameterSpec ktsParameterSpec = new KTSParameterSpec.Builder(
                     NISTObjectIdentifiers.id_aes256_wrap.getId(), 256).build();

    wrapper.init(Cipher.WRAP_MODE, publicKey, ktsParameterSpec);

    return wrapper.wrap(secretKey);
}

public SecretKey unwrapKey(RSAPrivateKey privateKey, byte[] wrappedKey)
    throws Exception
{
    Cipher unwrapper = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");
    KTSParameterSpec ktsParameterSpec = new KTSParameterSpec.Builder(
                     NISTObjectIdentifiers.id_aes256_wrap.getId(), 256).build();

    unwrapper.init(Cipher.UNWRAP_MODE, privateKey, ktsParameterSpec);

    return (SecretKey)unwrapper.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
}
```

8.1.2 JCE KTS-KEM-KWS in approved mode

Note that in approved mode KTS-KEM-KWS has to be provided with OtherInfo, which is a byte array providing additional data for the KDF used with KEMS that helps distinguish the exchange.

Suggestions for what should go into OtherInfo are in SP 800-56A and a utility class org.bouncycastle.crypto.util.DEROtherInfo is provided for constructing OtherInfo data in line with the common guidelines. As far as the API is concerned OtherInfo is just a byte array.

```
public byte[] wrapKey(
    RSAPublicKey publicKey, SecretKey secretKey, byte[] otherInfo)
    throws Exception
{
    Cipher wrapper = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");
    KTSParameterSpec ktsParameterSpec = new KTSParameterSpec.Builder(
          NISTObjectIdentifiers.id_aes256_wrap.getId(), 256, otherInfo).build();

    wrapper.init(Cipher.WRAP_MODE, publicKey, ktsParameterSpec);

    return wrapper.wrap(secretKey);
}

public SecretKey unwrapKey(
    RSAPrivateKey privateKey, byte[] wrappedKey, byte[] otherInfo)
    throws Exception
{
    Cipher unwrapper = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");
    KTSParameterSpec ktsParameterSpec = new KTSParameterSpec.Builder(
          NISTObjectIdentifiers.id_aes256_wrap.getId(), 256, otherInfo).build();

    unwrapper.init(Cipher.UNWRAP_MODE, privateKey, ktsParameterSpec);

    return (SecretKey)unwrapper.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
}
```

9 Key Wrapping Algorithms

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

9.1 Available in Approved Mode Operation

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

Cipher	Algorithm	Low-level Class	JCE Name
AES	KW	FipsAES	AESKW, AESWrap
AES	KWP	FipsAES	AESKWP, AESWrapPad
DESede	TKW	FipsTripleDES	DESedeTKW, DESedeWrap

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

Cipher	Algorithm	Low-level Class	JCE Name
RSA	OAEP	FipsRSA	RSA/NONE/OAEPwithSHA1andMGF1Padding, RSA/NONE/OAEPwithSHA224andMGF1Padding, RSA/NONE/OAEPwithSHA256andMGF1Padding, RSA/NONE/OAEPwithSHA384andMGF1Padding, RSA/NONE/OAEPwithSHA512andMGF1Padding

9.2 Available in General Operation

Cipher	Algorithm	Low-level Class	JCE Name
AES	RFC3211	AES	AESRFC3211Wrap
Camellia	KW	Camellia	CamelliaKW, CamelliaWrap
Camellia	KWP	Camellia	CamelliaKWP, CamelliaWrapPad
DESede	RFC3211	TripleDES	DESedeRFC3211Wrap
DESede	RFC3217	TripleDES	DESedeRFC3217Wrap
SEED	KW	SEED	SEEDKW, SEEDWrap
SEED	KWP	SEED	SEEDKWP, SEEDWrapPad
Serpent	KW	Serpent	SerpentKW, SerpentWrap
Serpent	KWP	Serpent	SerpentKWP, SerpentWrapPad
Twofish	KW	Twofish	TwofishKW, TwofishWrap

Cipher	Algorithm	Low-level Class	JCE Name
Twofish	KWP	Twofish	TwofishKWP, TwofishWrapPad

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

Cipher	Algorithm	Low-level Class	JCE Name
ElGamal	OAEP	ElGamal	ElGamal/NONE/OAEPwithSHA1andMGF1Padding, ElGamal/NONE/OAEPwithSHA224andMGF1Padding, ElGamal/NONE/OAEPwithSHA256andMGF1Padding, ElGamal/NONE/OAEPwithSHA384andMGF1Padding, ElGamal/NONE/OAEPwithSHA512andMGF1Padding
ElGamal	PKCS#1	ElGamal	ElGamal/NONE/PKCS1Padding
RSA	PKCS#1	RSA	RSA/NONE/PKCS1Padding

9.3 Examples

9.3.1 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[] inputKeyBytes)
{
    FipsRSA.KeyWrapOperatorFactory wrapFact =
                                   new FipsRSA.KeyWrapOperatorFactory();

    FipsKeyWrapperUsingSecureRandom wrapper =
            wrapFact.createKeyWrapper(pubKey, FipsRSA.WRAP_OAEP)
                          .withSecureRandom(new SecureRandom());

    return wrapper.wrap(inputKeyBytes, 0, inputKeyBytes.length);
}
```

The following snippet shows an example of unwrapping a wrapped secret key that has been wrapped using RSA with OAEP in the low-level API.

```
public byte[] unwrapKey(AsymmetricRSAPrivateKey privKey, byte[] wrappedKeyBytes)
{
    FipsRSA.KeyWrapOperatorFactory wrapFact =
                                  new FipsRSA.KeyWrapOperatorFactory();

    FipsKeyUnwrapperUsingSecureRandom unwrapper =
            wrapFact.createKeyUnwrapper(privKey, FipsRSA.WRAP_OAEP)
                          .withSecureRandom(new SecureRandom());

    return wrapper.unwrap(wrappedKeyBytes, 0, wrappedKeyBytes.length);
}
```

9.3.2 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
SymmetricKey aesKey = new SymmetricSecretKey(FipsAES.KW, keyBytes);
FipsAES.KeyWrapOperatorFactory factory = new FipsAES.KeyWrapOperatorFactory();
KeyWrapper wrapper = factory.createKeyWrapper(aesKey, FipsAES.KW);

byte[] wrappedBytes = wrapper.wrap(inputKeyBytes, 0, inputKeyBytes.length);
```

9.3.3 JCE Key Wrapping using ElGamal

The following snippet shows an example of using ElGamal with OAEP to wrap a secret key.

```
public byte[] wrapKey(DHPublicKey pubKey, SecretKey keyToBeWrapped)
    throws Exception
{
    Cipher c = Cipher.getInstance(
                "ElGamal/NONE/OAEPwithSHA256andMGF1Padding", "BCFIPS");

    c.init(Cipher.WRAP_MODE, pubKey);

    return c.wrap(keyToBeWrapped);
}
```

The following snippet shows an example of unwrapping a wrapped secret key that has been wrapped using ElGamal with OAEP.

```
public SecretKey unwrapKey(
          DHPrivateKey privKey, byte[] wrappedKey, String wrappedKeyAlgorithm)
    throws Exception
{
    Cipher c = Cipher.getInstance(
                "ElGamal/NONE/OAEPwithSHA256andMGF1Padding", "BCFIPS");

    c.init(Cipher.UNWRAP_MODE, privKey);

    return (SecretKey)c.unwrap(out, wrappedKeyAlgorithm, Cipher.SECRET_KEY);
}
```

9.3.4 JCE Key Wrapping using Camellia with padding

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(SecretKey secKey, SecretKey keyToBeWrapped)
    throws Exception
{
    Cipher c = Cipher.getInstance(
                "CamelliaWrapPad", "BCFIPS");
```

```
    c.init(Cipher.WRAP_MODE, secKey);

    return c.wrap(keyToBeWrapped);
}
```

The second snippet shows the key unwrap process. The JCE also requires an algorithm name and key type (in this case Cipher.SECRET_KEY) to unwrap a key.

```
public SecretKey unwrapKey(
          SecretKey secKey, byte[] wrappedKey, String wrappedKeyAlgorithm)
    throws Exception
{
    Cipher c = Cipher.getInstance(
                 “CamelliaWrapPad”, “BCFIPS”);

    c.init(Cipher.UNWRAP_MODE, secKey);

    return (SecretKey)c.unwrap(out, wrappedKeyAlgorithm, Cipher.SECRET_KEY);
}
```

10 Mac Algorithms

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

10.1 Available in Approved Mode Operation

MAC Name	Low-level Class	JCE Name
AES CCM MAC	FipsAES	CCMMAC, AESCCMMAC, AES-CCMMAC
AES CMAC	FipsAES	CMAC, AESCMAC, AES-CMAC
AES GMAC	FipsAES	GMAC, AESGMAC, AES-GMAC
HMAC SHA-1	FipsSHS	HmacSHA1, Hmac128SHA1
HMAC SHA-224	FipsSHS	HmacSHA224, Hmac128SHA224
HMAC SHA-256	FipsSHS	HmacSHA256, Hmac128SHA256
HMAC SHA-384	FipsSHS	HmacSHA384, Hmac256SHA384
HMAC SHA-512	FipsSHS	HmacSHA512, Hmac256SHA512
HMAC SHA-512(224)	FipsSHS	HmacSHA512(224), Hmac128SHA512(224)
HMAC SHA-512(256)	FipsSHS	HmacSHA512(256), Hmac128SHA512(256)
HMAC SHA3-224	FipsSHS	HmacSHA3-224
HMAC SHA3-256	FipsSHS	HmacSHA3-256
HMAC SHA3-384	FipsSHS	HmacSHA3-384
HMAC SHA3-512	FipsSHS	HmacSHA3-512
KMAC128	FipsSHS	KMACwithSHAKE128
KMAC256	FipsSHS	KMACwithSHAKE256
Triple-DES CMAC	FipsTripleDES	DESedeCMAC, DESede-CMAC

HMAC algorithms in the form of HmacNdigestName where N is a number, produce truncated versions of the HMAC in question. The right most bits are truncated as per the NIST standards.

10.2 Available in General Operation

MAC Name	Low-level Class	JCE Name
Blowfish CMAC	Blowfish	BlowfishCMAC, Blowfish-CMAC
Camellia CCM MAC	Camellia	CamelliaCCMMAC, Camellia-CCMMAC
Camellia CMAC	Camellia	CamelliaCMAC, Camellia-CMAC
Camellia GMAC	Camellia	CamelliaGMAC, Camellia-GMAC

MAC Name	Low-level Class	JCE Name
CAST5 CMAC	CAST5	CAST5CMAC, CAST5-CMAC
DES CBC MAC	DES	DESMAC
DES CFB8 MAC	DES	DESMAC/CFB8
DES MAC 64	DES	DESMAC64
DES MAC 64 with ISO7816-4 padding	DES	DESMAC64WITHISO7816-4PADDING
DES ISO9797 MAC	DES	ISO9797ALG3MAC
DES ISO9797 MAC with ISO7816-4 padding	DES	ISO9797ALG3WITHISO7816-4PADDING
GOST28147 MAC	GOST28147	GOST28147MAC
HMAC GOST3411	SecureHash	HmacGOST3411
HMAC GOST3411-2012-256	SecureHash	HmacGOST3411-2012-256
HMAC GOST3411-2012-256	SecureHash	HmacGOST3411-2012-512
HMAC RIPEMD128	SecureHash	HmacRIPEMD128
HMAC RIPEMD160	SecureHash	HmacRIPEMD160
HMAC RIPEMD256	SecureHash	HmacRIPEMD256
HMAC RIPEMD320	SecureHash	HmacRIPEMD320
HMAC Tiger	SecureHash	HmacTiger
HMAC Whirlpool	SecureHash	HmacWhirlpool
IDEA CMAC	IDEA	IDEACMAC, IDEA-CMAC
IDEA CBC MAC	IDEA	IDEAMAC
IDEA CFB8 MAC	IDEA	IDEAMAC/CFB8
SEED CCMMAC	SEED	SEEDCCMMAC, SEED-CCMMAC
SEED CMAC	SEED	SEEDCMAC, SEED-CMAC
SEED GMAC	SEED	SEEDGMAC, SEED-GMAC
Serpent CCM MAC	Serpent	SerpentCCMMAC, Serpent-CCMMAC
Serpent CMAC	Serpent	SerpentCMAC, Serpent-CMAC
Serpent GMAC	Serpent	SerpentGMAC, Serpent-GMAC
SHACAL-2 CMAC	SHACAL2	SHACAL-2CMAC, SHACAL-2-CMAC
Triple-DES CBC MAC	TripleDES	DESedeMAC
Triple-DES CFB8 MAC	TripleDES	DESedeMAC/CFB8
Triple-DES CBC MAC 64	TripleDES	DESedeMAC64
Triple-DES CBC MAC 64 with ISO7816-4 padding	TripleDES	DESedeMAC64withISO7816-4Padding
Twofish CCM MAC	Twofish	TwofishCCMMAC, Twofish-CCMMAC
Twofish CMAC	Twofish	TwofishCMAC, Twofish-CMAC

MAC Name	Low-level Class	JCE Name
Twofish GMAC	Twofish	TwofishGMAC, Twofish-GMAC

10.3 Examples

10.3.1 AES using CMAC – 64 bit.

The following snippet shows a function returning a 64 bit CMAC MAC produced from the passed in aesKey and data.

```
public byte[] createMac(SymmetricKey aesKey, byte[] data)
    throws IOException
{
    FipsMACOperatorFactory<FipsAES.AuthParameters> fipsSymmetricFactory =
                                          new FipsAES.MACOperatorFactory();
    FipsOutputMACCalculator<FipsAES.AuthParameters> macCalculator =
         fipsSymmetricFactory.createOutputMACCalculator(
            aesKey, FipsAES.CMAC.withMACSize(64));
    OutputStream sOut = macCalculator.getMACStream();
    sOut.write(data);
    sOut.close();

    return macCalculator.getMAC();
}
```

10.3.2 JCE HMAC-SHA256

The following snippet provides a function using the JCE APIs to do HMAC-SHA256 from the passed in HMAC key and data.

```
public byte[] createMac(SecretKey hmacKey, byte[] data)
    throws Exception
{
    Mac mac = Mac.getInstance(“HmacSHA256, "BCFIPS");

    mac.init(hmacKey, name));

    return mac.doFinal(data);
}
```

11 Message Digest Algorithms

The BC FIPS provider supports the full suite of NIST digests up to FIPS PUB-202 (SHA-3), as well a variety of common ones used by other standards.

11.1 Available in Approved Mode Operation

Digest Name	Low-level Class	JCA Name
SHA-1	FipsSHS	SHA-1, SHA1
SHA-224	FipsSHS	SHA-224, SHA224
SHA-256	FipsSHS	SHA-256, SHA256
SHA-384	FipsSHS	SHA-384, SHA384
SHA-512	FipsSHS	SHA-512, SHA512
SHA-512(224)	FipsSHS	SHA-512(224), SHA512(224)
SHA-512(256)	FipSHS	SHA-512(256), SHA512(256)
SHA3-224	FipsSHS	SHA3-224
SHA3-256	FipsSHS	SHA3-256
SHA3-384	FipsSHS	SHA3-384
SHA3-512	FipsSHS	SHA3-512
SHAKE128	FipsSHS	SHAKE128
SHAKE256	FipsSHS	SHAKE256
cSHAKE128	FipsSHS	N/A
cSHAKE256	FipsSHS	N/A

11.2 Available in General Operation

Digest Name	Low-level Class	JCA Name
GOST3411	SecureHash	GOST3411
GOST3411-2012-256	SecureHash	GOST3411-2012-256
GOST3411-2012-512	SecureHash	GOST3411-2012-512
RIPEMD128	SecureHash	RIPEMD128
RIPEMD160	SecureHash	RIPEMD160
RIPEMD256	SecureHash	RIPEMD256
RIPEMD320	SecureHash	RIPEMD320

Digest Name	Low-level Class	JCA Name
Tiger	SecureHash	Tiger
Whirlpool	SecureHash	Whirlpool

11.3 Examples

11.3.1 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 = …;

FipsDigestOperatorFactory<FipsSHS.Parameters> factory =
                              new FipsSHS.OperatorFactory<FipsSHS.Parameters>();

OutputDigestCalculator<FipsSHS.Parameters> calculator =
     factory.createOutputDigestCalculator(FipsSHS.SHA_256);

OutputStream digestStream = calculator.getDigestStream();
digestStream.write(input);
digestStream.close();

byte[] digest = calculator.getDigest();
```

11.3.2 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 = …;

FipsXOFOperatorFactory<FipsSHS.Parameters> factory =
                           new FipsSHS.XOFOperatorFactory<FipsSHS.Parameters>();

OutputXOFCalculator<FipsSHS.Parameters> calculator =
     factory.createOutputXOFCalculator(FipsSHS.SHAKE128);

OutputStream xofStream = calculator.getFunctionStream();
xofStream.write(input);
xofStream.close();

byte[] output = calculator.getFunctionOutput(100);
```

An interesting feature of the expandable output functions is that you can keep going, hence another call like:

```
byte[] extra = calculator.getFunctionOutput(100);
```

will just grab another 100 bytes from the function stream. The calculator will keep producing output until either `reset()` is called or `getFunctionStream()` is called again (this will trigger a reset as more input will be assumed).

11.3.3 JCA use of SHA3-224

All digests are available using MessageDigest.getInstance() and their JCA name as follows:

```
byte[] input = …;
MessageDigest digest = MessageDigest.getInstance("SHA3-224", "BCFIPS");


byte[] result = digest.digest(input);
```

12 Password Based Key Derivation Functions

PBKD functions are supported from the PKCS standards as well as OpenSSL. It is important to note that only PBKDF2 is approved for use in FIPS mode of operation and that this affects the use of things such as PKCS#12 files, which cannot be used in FIPS mode as a result.

12.1 Available in Approved Mode Operation

Algorithm	Byte Encoding Used	Digest for PRF	Low-level Class	JCE Name
PBKDF2	UTF-8	SHA-1	FipsPBKD	PBKDF2, PBKDF2withHmacSHA1
PBKDF2	8-BIT/ASCII	SHA-1	FipsPBKD	PBKDF2with8BIT
PBKDF2	UTF-8	SHA-224	FipsPBKD	PBKDF2withHmacSHA224
PBKDF2	UTF-8	SHA-256	FipsPBKD	PBKDF2withHmacSHA256
PBKDF2	UTF-8	SHA-384	FipsPBKD	PBKDF2withHmacSHA384
PBKDF2	UTF-8	SHA-512	FipsPBKD	PBKDF2withHmacSHA512

12.2 Available in General Operation

Algorithm	Byte Encoding Used	Digest for PRF	Low-level Class	JCE Name
OpenSSL	8-BIT/ASCII	MD5	PBKD	PBKDF-OpenSSL
PBKDF2	UTF-8	GOST-3411	PBKD	PBKDF2withHmacGOST3411
PBKDF1	8-BIT/ASCII	SHA-1	PBKD	PBEwithSHA1andDES
PBKDF1	8-BIT/ASCII	MD5	PBKD	PBEwithMD5andDES
PBKDF1	8-BIT/ASCII	SHA-1	PBKD	PBEwithSHA1andRC2
PBKDF1	8-BIT/ASCII	MD5	PBKD	PBEwithMD5andRC2
PKCS#12	PKCS#12	SHA-1	PBKD	PBKDF-PKCS12
PKCS#12	PKCS#12	SHA-256	PBKD	PBKDF-PKCS12withSHA256
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and40bitRC4
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and128bitRC4
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and40bitRC2
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and128bitRC2
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and2-KeyDESede
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and3-KeyDESede
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and128BitAES-BC

Algorithm	Byte Encoding Used	Digest for PRF	Low-level Class	JCE Name
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and192BitAES-BC
PKCS#12	PKCS#12	SHA-1	PBKD	PBEwithSHA1and256BitAES-BC
PKCS#12	PKCS#12	SHA-256	PBKD	PBEwithSHA256and128BitAES-BC
PKCS#12	PKCS#12	SHA-256	PBKD	PBEwithSHA256and192BitAES-BC
PKCS#12	PKCS#12	SHA-256	PBKD	PBEwithSHA256and256BitAES-BC

12.3 Examples

12.3.1 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)
{
    PasswordBasedDeriver deriver = new FipsPBKD.DeriverFactory().createDeriver(
            FipsPBKD.PBKDF2.using(FipsSHS.Algorithm.SHA256_HMAC,
                                     PasswordConverter.UTF8.convert(password))
                         .withIterationCount(iterationCount)
                         .withSalt(salt)
    );
    return deriver.deriveKey(PasswordBasedDeriver.KeyType.CIPHER,
                                                    (keySizeInBits + 7) / 8);
}
```

The KeyType parameter above can also be set to KeyType.MAC. In some cases, such as PKCS#12, this will affect the values in the key material that is generated. In the case of PBKDF2 the type parameter is ignored.

12.3.3 PKCS#12

The following snippet generates key and IV material using PKCS#12. Note in this case the derivation function returns an array of byte[], where the first element of the array is the key material and the second element is the IV.

```
public byte[][] getSecretKeyAndIV(char[] password, int iterationCount,
                              byte[] salt, int keySizeInBits, int ivSizeInBytes)
{
    PasswordBasedDeriver deriver = new PBKD.DeriverFactory().createDeriver(
        PBKD.PKCS12.
            using(PasswordConverter.PKCS12, password)
            .withIterationCount(iterationCount).withSalt(salt));

    return deriver.deriveKeyAndIV(PasswordBasedDeriver.KeyType.CIPHER,
                                     (keySizeInBits + 7) / 8, ivSizeInBytes);
}
```

12.3.3 JCE PBKDF2

The following examples shows the same function as described in 11.3.1, except in this case it is defined in the context of the JCE.

```
public SecretKey getSecretKey(char[] password, int iterationCount, byte[] salt,
                                                              int keySizeInBits)

{
    SecretKeyFactory keyFact = SecretKeyFactory.getInstance(
                                              "PBKDF2WITHHMACSHA256", "BCFIPS");
    PBEKeySpec pbeSpec = new PBEKeySpec(password, salt, iterationCount,
keySizeInBits);

    return keyFact.generateSecret(pbeSpec);
}
```

13 Random Number Generators

For the most part SecureRandom objects need to be created using the low-level class as the JCA does not provide the ability to configure them and this is required for both FIPS and X9.31.

13.1 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

The BouncyCastleFipsProvider also makes available two SecureRandom which are based on the provider's configuration and use approved mode DRBGs. You can get access to the provider SecureRandom using the name “DEFAULT”, as in:

```
SecureRandom random = SecureRandom.getInstance(“DEFAULT”, “BCFIPS”);
```

The “DEFAULT” SecureRandom is configured to be prediction resistant.

The second SecureRandom is the “NONCEANDIV” SecureRandom which is suitable for nonce and IV material. The “NONCEANDIV” SecureRandom can be accessed by:

```
SecureRandom ivRandom = SecureRandom.getInstance(“NONCEANDIV”, “BCFIPS”);
```

13.2 Available in General Operation

The X9.31 PRNG is available in general operation, both Triple-DES and AES are supported.

DRBG Name	Low-level Class
X9.31	X931PRNG

13.3 Examples

13.3.1 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(new VMID().toString());

SecureRandom entropySource = new SecureRandom();

SecureRandom random = FipsDRBG.SHA512.fromEntropySource(entropySource,true)
     .setPersonalizationString(personalizationString))
```

```
    .build(
         entropySource.generateSeed((256 / (2 * 8))),
         true,
         Strings.toByteArray("Additional Input"));
```

13.3.2 Creation for AES X9.31

The following example creates a X9.31 PRNG base on a default SecureRandom.

```
X931PRNG.Builder bld = X931PRNG.AES_128.fromEntropySource(new SecureRandom(),true);
bld.setDateTimeVector(Hex.decode("259e67249288597a4d61e7c0e690afae"));

SecureRandom rand = bld.build(
                              new SymmetricSecretKey(X931PRNG.AES_128,
                              Hex.decode("f7d36762b9915f1ed585eb8e91700eb2")),
                              false);
```

14 Signature Algorithms

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

14.1 Available in Approved Mode Operation

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

14.1.1 DSA

Signature	Low-level Class	JCA Name
SHA1 with DSA	FipsDSA	SHA1withDSA
SHA224 with DSA	FipsDSA	SHA224withDSA
SHA256 with DSA	FipsDSA	SHA256withDSA
SHA384 with DSA	FipsDSA	SHA384withDSA
SHA512 with DSA	FipsDSA	SHA512withDSA
SHA512(224) with DSA	FipsDSA	SHA512(224)withDSA
SHA512(256) with DSA	FipsDSA	SHA512(256)withDSA
SHA3-224 with DSA	FipsDSA	SHA3-224withDSA
SHA3-256 with DSA	FipsDSA	SHA3-256withDSA
SHA3-384 with DSA	FipsDSA	SHA3-384withDSA
SHA3-512 with DSA	FipsDSA	SHA3-512withDSA

14.1.2 EC

Signature	Low-level Class	JCA Name
SHA1 with ECDSA	FipsEC	SHA1withECDSA
SHA224 with ECDSA	FipsEC	SHA224withECDSA
SHA256 with ECDSA	FipsEC	SHA256withECDSA
SHA384 with ECDSA	FipsEC	SHA384withECDSA
SHA512 with ECDSA	FipsEC	SHA512withECDSA
SHA512(224) with ECDSA	FipsEC	SHA512(224)withECDSA
SHA512(256) with ECDSA	FipsEC	SHA512(256)withECDSA
SHA3-224 with ECDSA	FipsEC	SHA3-224withDSA

Signature	Low-level Class	JCA Name
SHA3-256 with ECDSA	FipsEC	SHA3-256withECDSA
SHA3-384 with ECDSA	FipsEC	SHA3-384withECDSA
SHA3-512 with ECDSA	FipsEC	SHA3-512withECDSA

14.1.3 RSA

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

14.1.3.1 PKCS1.5

Signature	Low-level Class	JCA Name
SHA1 with RSA	FipsRSA	SHA1withRSA
SHA224 with RSA	FipsRSA	SHA224withRSA
SHA256 with RSA	FipsRSA	SHA256withRSA
SHA384 with RSA	FipsRSA	SHA384withRSA
SHA512 with RSA	FipsRSA	SHA512withRSA
SHA512(224) with RSA	FipsRSA	SHA512(224)withRSA
SHA512(256) with RSA	FipsRSA	SHA512(256)withRSA
SHA3-224 with RSA	FipsRSA	SHA3-224withRSA
SHA3-256 with RSA	FipsRSA	SHA3-256withRSA
SHA3-384 with RSA	FipsRSA	SHA3-384withRSA
SHA3-512 with RSA	FipsRSA	SHA3-512withRSA

14.1.3.2 PSS

Signature	Low-level Class	JCA Name
PSS SHA1 with RSA	FipsRSA	SHA1withRSAandMGF1, SHA1withRSA/PSS
PSS SHA224 with RSA	FipsRSA	SHA224withRSAandMGF1, SHA224withRSA/PSS
PSS SHA256 with RSA	FipsRSA	SHA256withRSAandMGF1, SHA256withRSA/PSS
PSS SHA384 with RSA	FipsRSA	SHA384withRSAandMGF1, SHA384withRSA/PSS
PSS SHA512 with RSA	FipsRSA	SHA512withRSAandMGF1,

Signature	Low-level Class	JCA Name
		SHA512withRSA/PSS
PSS SHA512(224) with RSA	FipsRSA	SHA512(224)withRSAandMGF1, SHA512(224)withRSA/PSS
PSS SHA512(256) with RSA	FipsRSA	SHA512(256)withRSAandMGF1, SHA512(256)withRSA/PSS
PSS SHA3-224 with RSA	FipsRSA	SHA3-224withRSAandMGF1, SHA3-224withRSA/PSS
PSS SHA3-256 with RSA	FipsRSA	SHA3-256withRSAandMGF1, SHA3-256withRSA/PSS
PSS SHA3-384 with RSA	FipsRSA	SHA3-384withRSAandMGF1, SHA3-384withRSA/PSS
PSS SHA3-512 with RSA	FipsRSA	SHA3-512withRSAandMGF1, SHA3-512withRSA/PSS

14.1.3.3 X9.31

Signature	Low-level Class	JCA Name
X9.31 SHA1 with RSA	FipsRSA	SHA1withRSA/X9.31
X9.31 SHA224 with RSA	FipsRSA	SHA224withRSA/X9.31
X9.31 SHA256 with RSA	FipsRSA	SHA256withRSA/X9.31
X9.31 SHA384 with RSA	FipsRSA	SHA384withRSA/X9.31
X9.31 SHA512 with RSA	FipsRSA	SHA512withRSA/X9.31
X9.31 SHA512(224) with RSA	FipsRSA	SHA512(224)withRSA/X9.31
X9.31 SHA512(256) with RSA	FipsRSA	SHA512(256)withRSA/X9.31

14.2 Available in General Operation

14.2.1 DSA

14.2.1.1 Regular DSA

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

14.2.1.2 Deterministic DSA

Signature	Low-level Class	JCA Name
Deterministic SHA1 with DSA	DSA	SHA1withDDSA
Deterministic SHA224 with DSA	DSA	SHA224withDDSA
Deterministic SHA256 with DSA	DSA	SHA256withDDSA

Signature	Low-level Class	JCA Name
Deterministic SHA384 with DSA	DSA	SHA384withDDSA
Deterministic SHA512 with DSA	DSA	SHA512withDDSA
Deterministic SHA512(224) with DSA	DSA	SHA512(224)withDDSA
Deterministic SHA512(256) with DSA	DSA	SHA512(256)withDDSA
Deterministic SHA3-224 with DSA	DSA	SHA3-224withDDSA
Deterministic SHA3-256 with DSA	DSA	SHA3-256withDDSA
Deterministic SHA3-384 with DSA	DSA	SHA3-384withDDSA
Deterministic SHA3-512 with DSA	DSA	SHA3-512withDDSA

14.2.2 DSTU4145

Signature	Low-level Class	JCA Name
GOST3411 with DSTU4145	DSTU4145	GOST3411withDSTU4145
GOST3411 with DSTU4145 little endian	DSTU4145	GOST3411withDSTU4145LE

14.2.3 ECDSA

14.2.3.1 Regular ECDSA

Signature	Low-level Class	JCA Name
RipeMD160 with ECDSA	EC	RipeMD160withECDSA

14.2.3.2 Deterministic ECDSA

Signature	Low-level Class	JCA Name
Deterministic SHA1 with ECDSA	EC	SHA1withECDDSA
Deterministic SHA224 with ECDSA	EC	SHA224withECDDSA
Deterministic SHA256 with ECDSA	EC	SHA256withECDDSA
Deterministic SHA384 with ECDSA	EC	SHA384withECDDSA
Deterministic SHA512 with ECDSA	EC	SHA512withECDDSA
Deterministic SHA512 with ECDSA	EC	SHA512(224)withECDDSA
Deterministic SHA512 with ECDSA	EC	SHA512(256)withECDDSA
Deterministic SHA3-224 with ECDSA	EC	SHA3-224withECDDSA
Deterministic SHA3-256 with ECDSA	EC	SHA3-256withECDDSA

Signature	Low-level Class	JCA Name
Deterministic SHA3-384 with ECDSA	EC	SHA3-384withECDDSA
Deterministic SHA3-512 with ECDSA	EC	SHA3-512withECDDSA

14.2.4 GOST3410-1994

Signature	Low-level Class	JCA Name
GOST3411 with GOST3410-1994	GOST3410	GOST3411withGOST3410

14.2.5 GOST3410-2001

Signature	Low-level Class	JCA Name
GOST3411 with GOST3410-2001	ECGOST3410	GOST3411withECGOST3410

14.2.6 LMS (HSS)

Signature	Low-level Class	JCA Name
LMS	LMS	LMS

14.2.7 RSA

14.2.6.1 ISO9796-2

Signature	Low-level Class	JCA Name
ISO9796-2 SHA1 with RSA	RSA	SHA1withRSA/ISO9796-2
ISO9796-2 SHA224 with RSA	RSA	SHA224withRSA/ISO9796-2
ISO9796-2 SHA256 with RSA	RSA	SHA256withRSA/ISO9796-2
ISO9796-2 SHA384 with RSA	RSA	SHA384withRSA/ISO9796-2
ISO9796-2 SHA512 with RSA	RSA	SHA512withRSA/ISO9796-2
ISO9796-2 SHA512(224) with RSA	RSA	SHA512(224)withRSA/ISO9796-2
ISO9796-2 SHA512(256) with RSA	RSA	SHA512(256)withRSA/ISO9796-2
ISO9796-2 RIPEMD128 with RSA	RSA	RIPEMD128withRSA/ISO9796-2
ISO9796-2 RIPEMD160 with RSA	RSA	RIPEMD160withRSA/ISO9796-2

14.2.6.2 ISO9796-2/PSS

Signature	Low-level Class	JCA Name
ISO9796-2 PSS SHA1 with RSA	RSA	SHA1withRSA/ISO9796-2PSS
ISO9796-2 PSS SHA224 with RSA	RSA	SHA224withRSA/ISO9796-2PSS
ISO9796-2 PSS SHA256 with RSA	RSA	SHA256withRSA/ISO9796-2PSS
ISO9796-2 PSS SHA384 with RSA	RSA	SHA384withRSA/ISO9796-2PSS
ISO9796-2 PSS SHA512 with RSA	RSA	SHA512withRSA/ISO9796-2PSS
ISO9796-2 PSS SHA512(224) with RSA	RSA	SHA512(224)withRSA/ISO9796-2PSS
ISO9796-2 PSS SHA512(256) with RSA	RSA	SHA512(256)withRSA/ISO9796-2PSS
ISO9796-2 PSS RIPEMD128 with RSA	RSA	RIPEMD128withRSA/ISO9796-2PSS
ISO9796-2 PSS RIPEMD160 with RSA	RSA	RIPEMD160withRSA/ISO9796-2PSS

14.2.6.3 PKCS1.5

Signature	Low-level Class	JCA Name
MD5 with RSA	RSA	MD5withRSA
RIPEMD128 with RSA	RSA	RIPEMD128withRSA
RIPEMD160 with RSA	RSA	RIPEMD160withRSA
RIPEMD256 with RSA	RSA	RIPEMD256withRSA

14.2.6.4 PSS

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

14.2.6.5 X9.31

Signature	Low-level Class	JCA Name
X9.31 RIPEMD128 with RSA	RSA	RIPEMD128withRSA/X9.31
X9.31 RIPEMD160 with RSA	RSA	RIPEMD160withRSA/X9.31
X9.31 Whirlpool with RSA	RSA	WhirlpoolwithRSA/X9.31

14.3 Examples

14.3.1 RSA with SHA-256

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)
```

```
    throws Exception
{
    FipsRSA.SignatureOperatorFactory signatureOperatorFactory =
                                             new FipsRSA.SignatureOperatorFactory();

    OutputSignerUsingSecureRandom<FipsRSA.PKCS1v15SignatureParameters> rsaSig =
                     signatureOperatorFactory.createSigner(privKey,
                     FipsRSA.PKCSv1_5.withDigestAlgorithm(
                                              FipsSHS.Algorithm.SHA256));

    rsaSig = rsaSig.withSecureRandom(new SecureRandom());

    OutputStream sOut = rsaSig.getSigningStream();
    sOut.write(data, 0, data.length);
    sOut.close();

    return rsaSig.getSignature();
}
```

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)
    throws Exception
{
    FipsRSA.SignatureOperatorFactory signatureOperatorFactory =
                                             new FipsRSA.SignatureOperatorFactory();

    OutputVerifier<FipsRSA.PKCS1v15SignatureParameters> rsaVer =
                 signatureOperatorFactory.createVerifier(rsaPublic,
                         FipsRSA.PKCSv1_5.withDigestAlgorithm(
                                              FipsSHS.Algorithm.SHA256));

    OutputStream vOut = rsaVer.getVerifyingStream();
    vOut.write(data, 0, data.length);
    vOut.close();

    return rsaVer.isVerified(signature);
}
```

14.3.2 JCA 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(ECPrivateKey privKey, byte[] data)
    throws Exception
{
    Signature sig = Signature.getInstance("SHA256withECDSA", "BCFIPS");

    sig.initSign(privKey);
    sig.update(dummySha1);

    return sig.sign();
}
```

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(ECPublicKey pubKey, byte[] signature, byte[] data)
    throws Exception
{
    Signature sig = Signature.getInstance("SHA256withECDSA", "BCFIPS");

    sig.initVerify(pubKey);

    sig.update(data);

    return sig.verify((signature);
}
```

Appendix A – System Properties

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

`ocsp.enable` – setting this property to true will enable support for OCSP

`ocsp.responderURL` – url provided as access point for the OCSP responder for the CertPath API to use.

org.bouncycastle.ec.allow_sha1_sig – setting this property to `true` will enable support for ECDSA signature generation based around the SHA-1 message digest.

`org.bouncycastle.ec.disable` – setting this property to `true` will disable support for EC cryptography.

`org.bouncycastle.ec.disable_f2m` – setting this property to `true` will disable support for EC Characteristic 2 (F2M) curves.

`org.bouncycastle.ec.disable_mqv` – setting this property to `true` will disable support for EC MQV.

`org.bouncycastle.entropy.factors` – setting this property will configure the two C values and the H value used for testing the entropy source according to Section 4.4 of SP 800-90B. See section 8.7 of the module's Security Policy for further details.

`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.

`org.bouncycastle.x509.enableCRLDP` – setting this property to true will enable checking of remote CRL distribution points.

`org.bouncycastle.dsa.FIPS186-2for1024bits` – this property only has an effect in unapproved mode. If legacy DSA parameters must be generated and the parameter size is 1024 setting this property to `true` will result in the FIPS 186-2 algorithm being used for parameter generation.

`org.bouncycastle.fips.approved_only` – if this property is set to *true* the module will start in approved mode and non-approved mode functionality will not be available**.**

`org.bouncycastle.jca.enable_jks` – enable the JKS key store for FIPS. In order to maintain compliance this key store can only be used for reading JKS keystores containing certificates. Default is false to prevent unnecessary conflicts with the regular Java one. If set to `true` the key store is enabled.

`org.bouncycastle.jsse.enable_md5` – if set to `true`, enable the MD5 digest in FIPS approved mode. Note, in the case of security related uses this is only provided for compatibility with the current version SP 800-52. This property is not set by default.

`org.bouncycastle.pkcs1.not_strict` – 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.pkix.disable_certpath` – in some cases it's easier to use the default CertPath implementation, rather than the one provided in the module. Setting this property to **true** will result in the BCFIPS CertPathValidator and CertPathBuilder not being created in an BouncyCastleFipsProvider that is instanced after the property is set.

`org.bouncycastle.rsa.allow_multi_use` – 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. It is possible to stop this from happening by setting **`org.bouncycastle.rsa.allow_multi_use`** to **`true`**.

`org.bouncycastle.rsa.allow_sha1_sig` – if set to **`true`**, allow the use of SHA-1 in PKCS1.5 and PSS signature generation. This property is not set by default.

`org.bouncycastle.tripledes.allow_weak` – 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.

Appendix B – Policy Permissions

If the BC FIPS module is used in association with a SecurityManager there are some Java permissions that need to be set in applications policy file, together with some optional ones that are specific to the BC FIPS module.

B.1 Java Permissions

In order to do the checksum validation of the jar

```
    permission java.lang.RuntimePermission "getProtectionDomain";
```

needs to be enabled in order for the module jar to examine its own contents.

In order to check for configuration properties the policy permission:

```
    permission java.util.PropertyPermission "java.runtime.name", "read";
```

also need to be provided.

The module also makes use of reflection to enable use of later than JDK 1.5 classes. In order to enable this the policy permission:

```
    permission java.lang.RuntimePermission "accessDeclaredMembers";
```

is required.

If the JCA/JCE provider is to be installed during execution, the policy permission:

```
    permission java.security.SecurityPermission "putProviderProperty.BCFIPS";
```

is also required.

B.2 Optional Permissions

B.2.1 Configuration of Approved/Unapproved Modes

CryptoServicesRegistrar calculates the default mode of operation based on the granting of

```
permission
   org.bouncycastle.crypto.CryptoServicesPermission “unapprovedModeEnabled”;
```

If this permission is granted by the security manager, then the JVM will start threads in a default of unapproved mode.

If this permission is not granted by the security manager, then the JVM will start threads in the approved mode only.

B.2.2 Use of CryptoServicesRegistrar.setApprovedMode(true)

If the JVM has been granted the use of unapproved mode services then a thread may move into approved mode by calling CryptoServicesRegistrar.setApprovedMode(true) if the permission:

```
permission
 org.bouncycastle.crypto.CryptoServicesPermission “changeToApprovedModeEnabled”;
```

is granted.

If the permission is not granted together and a thread is not already in approved mode then the call to CryptoServicesRegistrar.setApprovedMode(true) will result in an exception being thrown.

B.2.3 Key Export and Translation

```
 permission org.bouncycastle.crypto.CryptoServicesPermission “exportSecretKey”;
```

and

```
 permission org.bouncycastle.crypto.CryptoServicesPermission “exportPrivateKey”;
```

or

```
 permission org.bouncycastle.crypto.CryptoServicesPermission "exportKeys";
```

are required to do any exporting of CSPs outside of the module. These permissions are also required to be set to allow repackaging of keys between layers.

If neither of these permissions are set it is possible to import keys into the module and to generate keys within it, however without them the private values can never be displayed or persisted.

B.2.4 SSL Support

RSA PKCS#1.5 key wrap, NONEwithDSA, NONEwithECDSA, and NONEwithRSA require 2 additional policy settings if the BCFIPS provider is run with a SecurityManager present and in “FIPS only” mode – as a rule these algorithms are not FIPS approved, except where used for TLS and the policy settings reflect this. In the most general case this will need:

```
permission
    org.bouncycastle.crypto.CryptoServicesPermission "tlsAlgorithmsEnabled";
```

which is the same as setting the following two permissions together:

```
permission
     org.bouncycastle.crypto.CryptoServicesPermission "tlsNullDigestEnabled";

permission
   org.bouncycastle.crypto.CryptoServicesPermission "tlsPKCS15KeyWrapEnabled";
```

B.2.5 Setting of Default SecureRandom

Permission is required for threads to set the default secure random in the presence of a security manager.

The following permission is required to set the default secure random using the CryptoServicesRegistrar

```
permission
   org.bouncycastle.crypto.CryptoServicesPermission “defaultRandomConfig”;
```

B.2.6 Setting CryptoServicesRegistrar Properties

Permissions are required for threads to either set thread local properties or global properties in the CryptoServicesRegistrar. Possession of permission to set a global property in the CryptoServicesRegistrar automatically implies permission to set a thread local property.

The following permission is required to set a thread local property on the CryptoServicesRegistrar

```
permission
   org.bouncycastle.crypto.CryptoServicesPermission “threadLocalConfig”;
```

The following permission is required to set a global property on the CryptoServicesRegistrar

```
permission
   org.bouncycastle.crypto.CryptoServicesPermission “globalConfig”;
```

Appendix C – 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 D – Use with OSGI

In order to maintain some support the JSSE it was necessary to allow a specific sun classes, the TlsRsaPremasterSecretParameterSpec. For this reason many OSGI containers may have issues with the bc-fips jar. If this happens adding:

```
sun.security.internal.spec
```

to the

```
org.osgi.framework.system.packages.extra
```

configuration property for the container should deal with the issue.

Likewise with containers like JBoss it is also necessaary to include the path for “sun/security/internal/spec” in the module dependencies.

Appendix E – Public/Private Key Conversion

All public/private keys in both the JCE/JCA and the low-level API return the appropriate ASN.1 encoding via their getEncoded() methods.

So, for example, the following shows how to convert a JCA RSA key pair into low level keys:

```
KeyPairGenerator kpGen = KeyPairGenerator.getInstance("RSA", "BCFIPS");
kpGen.initialize(2048);

KeyPair kp = kpGen.generateKeyPair();

AsymmetricRSAPublicKey rsaPubKey =
    new AsymmetricRSAPublicKey(FipsRSA.ALGORITHM, kp.getPublic().getEncoded());


AsymmetricRSAPrivateKey rsaPrivKey =
   new AsymmetricRSAPrivateKey(FipsRSA.ALGORITHM, kp.getPrivate().getEncoded());
```

Going back the other way requires the use of the JCA KeyFactory class. In this case we use the getEncoded() method on the the two low-level keys, but in this case create the corresponding key specs which we then pass to the KeyFactory. For example:

```
KeyFactory keyFactory = KeyFactory.getInstance("RSA", "BCFIPS");

RSAPublicKey jcaPubKey = (RSAPublicKey)keyFactory.generatePublic(
                                new X509EncodedKeySpec(rsaPubKey.getEncoded()));

RSAPrivateKey jcaPrivKey = (RSAPrivateKey)keyFactory.generatePrivate(
                              new PKCS8EncodedKeySpec(rsaPrivKey.getEncoded()));
```

will convert the two low level keys created in the first code snippet back into their JCA counterparts.

Appendix F – The BCFKS file Format

The BCFKS KeyStore files are constructed using ASN.1 where the outer layer of the key store is described as follows:

```
ObjectStore ::= SEQUENCE {
      CHOICE {
            encryptedObjectStoreData EncryptedObjectStoreData,
            objectStoreData ObjectStoreData
      }
      integrityCheck ObjectStoreIntegrityCheck
}
```

In the case of the BCFKS store the CHOICE item is *EncryptedObjectStoreData* and the integrity check is HMAC SHA-512 using a 512 bit key derived from the KeyStore password. The algorithm used to derive the key is PKCS#5 Scheme 2 and the password is converted into bytes for the key deriver using the PKCS#12 scheme, so each Java character is converted into 2 bytes. A differentiator is also concatenated with the password so that integrity keys will not equal encryption keys.

EncryptedObjectStoreData is defined as:

```
EncryptedObjectStoreData ::= SEQUENCE {
      encryptionAlgorithm AlgorithmIdentifier
      encryptedContent OCTET STRING
}
```

In this case the algorithm used is 256 bit AES CCM. The outer key is again derived from the password using PKCS#5 scheme2 but uses a different salt as well as a differentiator specific for store encryption to avoid key overlap.

The *OCTET STRING* representing the encrypted content is an encrypted encoding of an *ObjectStoreData* structure.

```
ObjectStoreData ::= SEQUENCE {
       version INTEGER.
       dataSalt OCTET STRING,
       integrityAlgorithm AlgorithmIdentifier,
       creationDate GeneralizedTime,
       lastModifiedDate GeneralizedTime,
       objectDataSequence ObjectDataSequence,
       comment UTF8String OPTIONAL
}
```

with *ObjectDataSequence* defined as:

```
ObjectDataSequence ::= SEQUENCE OF ObjectData
```

and *ObjectData* defined as:

```
ObjectData ::= SEQUENCE {
       type             INTEGER,
       identifier       UTF8String,
       creationDate     GeneralizedTime,
       lastModifiedDate GeneralizedTime,
       data             OCTET STRING,
```

```
      comment          UTF8String OPTIONAL
}
```

In the case of a BCFKS key store, type will be one of *CERTIFCATE(0)*, *PRIVATE_KEY(1)*, *SECRET_KEY(2)*, *PROTECTED_PRIVATE_KEY(3)*, and *PROTECTED_SECRET_KEY(4)*. The last two types allow for some customisation of the encryption used for storing a private key or a secret key via the `setKeyEntry()` method that takes a byte array.

Certificates are stored using an encoding of the standard PKIX Certificate type.

In the case of private keys and secret keys, they are stored using either an encoding of an *EncryptedPrivateKeyData* object or an *EncryptedSecretKeyData* object.

EncryptedPrivateKeyData is defined as:

```
EncryptedPrivateKeyObjectData ::= SEQUENCE {
      encryptedPrivateKeyInfo EncryptedPrivateKeyInfo,
      certificates SEQUENCE OF Certificate
}
```

where the *encryptedPrivateKeyInfo* is generated using 256 bit AES CCM with a key derived from the password to store the key using the same scheme as with the store itself. A differentiator is also concatenated with the password to avoid key overlap in case the same password is used for a different purpose elsewhere in the KeyStore.

EncryptedSecretKeyData is defined as::

```
EncryptedSecretKeyData ::= SEQUENCE {
       keyEncryptionAlgorithm AlgorithmIdentifier,
      encryptedKeyData OCTET STRING
}
```

where the encrypted key data is an encrypted encoding of a *SecretKeyData* object defined as:

```
SecretKeyData ::= SEQUENCE {
      keyAlgorithm OBJECT IDENTIFIER,
      keyBytes OCTET STRING
}
```

The encoding of the *SecretKeyData* is also encrypted using 256 bit AES CCM using a key derived from the password to store the key using the same scheme as with the store itself. A differentiator is also concatenated with the password to avoid key overlap in case the same password is used for a different purpose elsewhere in the KeyStore.

Appendix G – Troubleshooting

1. Use of the BCFIPS provider results in exceptions reporting as: `"java.lang.SecurityException: Unsupported keysize or algorithm parameters"` or `"java.security.InvalidKeyException: Illegal key size".`

If you see one of these exceptions it means the unrestricted policy files for the JVM you are using have not been installed. For a standard JVM you can navigate to the policy files by going to http://www.oracle.com/technetwork/java/index.html and and then looking in the download area for your the JDK/JRE you are using (the links to the policy files are normally at the bottom of the page). Download the zip file provided, follow the instructions, making sure you are installing the files into the JVM you are running with, and you should find the exception stops happening.

Note: Providing maximum key sizes are not exceeded it is possible to use the BCFIPS provider without getting this exception, so a lack of these exceptions in early operation may simply mean nothing has happened to trigger them. If it suddenly starts happening the first thing to check is the policy files.

2. JVM using the BCFIPS provider periodically blocks, especially on start up.

The BCFIPS provider is very careful about making sure it has access to sufficient entropy to generate good quality keys. Occasionally the source the JVM 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.

In the case where this is not possible you can either implement an entropy pool based on a FIPS DRBG and use that to seed your DRBG, or in the case where you are using the JCA/JCE you can configure an entropy pool for the BCFIPS provider by using the HYBRID configuration option (see Chapter 2, “Installation”, Section 2.3).

3. JSSE – cannot find JKS key store type or key store not from BCFIPS provider.

If using the JSSE in FIPS mode, the key stores containing either the private server credentials, or the private client credentials, must be readable using the BCFIPS provider. The only key store type the BCFIPS provider has available that is FIPS compliant is the BCFKS key store type, so when using the JSSE in FIPS mode, the key stores for private key credentials need to be of the type BCFKS, and any container utilising the JSSE needs to be configured appropriately.

For example, in tomcat, the Connector config for SSL/TLS will usually need to include:

```
keystoreType="BCFKS"
keystoreProvider="BCFIPS"
```

To make sure the tomcat passes in the correct key store type to allow the JSSE to work in FIPS mode with the BCFIPS provider.

Appendix H – 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.

In the case of one standard we implement, derived from RFC 5753, we are aware of a disclosure requirement. This is to say, the module contains implementations of EC MQV as described in RFC 5753, “Use of ECC Algorithms in CMS”. In line with the conditions in:

http://www.ietf.org/ietf-ftp/IPR/certicom-ipr-rfc-5753.pdf

We state, where EC MQV has not otherwise been disabled:

“The use of this product or service is subject to the reasonable, non-discriminatory terms in the Intellectual Property Rights (IPR) Disclosures of Certicom Corp. at the IETF for Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS) implemented in the product or service.”

Appendix I – 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-FJA (Bouncy Castle FIPS Java API) Non-Proprietary FIPS 140-3 Cryptographic Module Security Policy, Software Version: 2.0.0”, Legion of the Bouncy Castle Inc. July 16, 2021.