SafeNet ProtectToolkit J

Reference Guide

gemalto
security to be free

Part Number 007-007556-007
Version 5.2

Trademarks

Gemalto Rebranding

In early 2015, Gemalto NV completed its acquisition of SafeNet, Inc. As part of the process of rationalizing the product portfolios between the two organizations, the HSM product portfolio has been streamlined under the SafeNet brand. As a result, the ProtectServer/ProtectToolkit product line has been rebranded as follows:

Old product name	New product name
Protect Server External 2 (PSE2)	SafeNet ProtectServer Network HSM
Protect Server Internal Express 2 (PSI-E2)	SafeNet ProtectServer PCIe HSM
ProtectToolkit	SafeNet ProtectToolkit

Disclaimer

All information herein is either public information or is the property of and owned solely by Gemalto NV. and/or its subsidiaries who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Nothing herein shall be construed as implying or granting to you any rights, by license, grant or otherwise, under any intellectual and/or industrial property rights of or concerning any of Gemalto's information.

This document can be used for informational, non-commercial, internal and personal use only provided that:

- The copyright notice below, the confidentiality and proprietary legend and this full warning notice appear in all copies.
- This document shall not be posted on any network computer or broadcast in any media and no modification of any part of this document shall be made.

Use for any other purpose is expressly prohibited and may result in severe civil and criminal liabilities.

The information contained in this document is provided "AS IS" without any warranty of any kind. Unless otherwise expressly agreed in writing, Gemalto makes no warranty as to the value or accuracy of information contained herein.

The document could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Furthermore, Gemalto reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time.

Gemalto hereby disclaims all warranties and conditions with regard to the information contained herein, including all implied warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall Gemalto be liable, whether in contract, tort or otherwise, for any indirect, special or consequential damages or any damages whatsoever including but not limited to damages resulting from loss of use, data, profits, revenues, or customers, arising out of or in connection with the use or performance of information contained in this document.

Gemalto does not and shall not warrant that this product will be resistant to all possible attacks and shall not incur, and disclaims, any liability in this respect. Even if each product is compliant with current security standards in force on the date of their design, security mechanisms' resistance necessarily evolves according to the state of the art in security and notably under the emergence of new attacks. Under no circumstances, shall Gemalto be held liable for any third party actions and in particular in case of any successful attack against systems or equipment incorporating Gemalto products. Gemalto disclaims any liability with respect to security for direct, indirect, incidental or consequential damages that

result from any use of its products. It is further stressed that independent testing and verification by the person using the product is particularly encouraged, especially in any application in which defective, incorrect or insecure functioning could result in damage to persons or property, denial of service or loss of privacy.

Technical Support

If you encounter a problem while installing, registering or operating this product, please make sure that you have read the documentation. If you cannot resolve the issue, please contact your supplier or Gemalto support. Gemalto support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between Gemalto and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Contact method	Contact	
Address	Gemalto NV 4690 Millennium Drive Belcamp, Maryland 21017 USA	
Phone	Global	+1 410-931-7520
	Australia	1800.020.183
	China	(86) 10 8851 9191
	France	0825 341000
	Germany	01803 7246269
	India	000.800.100.4290
	Netherlands	0800.022.2996
	New Zealand	0800.440.359
	Portugal	800.1302.029
	Singapore	800.863.499
	Spain	900.938.717
	Sweden	020.791.028
	Switzerland	0800.564.849
	United Kingdom	0800.056.3158
	United States	(800) 545-6608
Web	www.safenet-inc.com	
Support and Downloads	www.safenet-inc.com/support Provides access to the Gemalto Knowledge Base and quick downloads for various products.	
Technical Support Customer Portal	https://serviceportal.safenet-inc.com Existing customers with a Technical Support Customer Portal account can log in to manage incidents, get the latest software upgrades, and access the Gemalto Knowledge Base.	

Revision History

Revision	Date	Reason
A	14 March 2016	Release 5.2

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

C H A P T E R 1
INTRODUCTION

Who Should Read This Document?

This document is highly technical in nature and is intended for Java software developers using ProtectToolkit J.

Readers need to familiarise themselves with the contents of the SafeNet ProtectToolkit C documentation.

Product Overview

ProtectToolkit J is a JCA/JCE `Provider` for Java™ (from now on referred to as Java). It implements a number of cryptographic algorithms which are supported by the SafeNet hardware encryption adapter. The device supports encryption, message digests, key storage and message authentication.

This document assumes some knowledge of the Java programming language, the JCA/JCE application programming interfaces, and additionally some understanding of the underlying adapter interface which is based on PKCS#11 (Cryptoki). The ProtectToolkit C Administration Manual contains further information on Cryptoki. For general information on the JCA/JCE please consult the JCA/JCE API Overview and Tutorial (as well as the Javadoc API reference). Furthermore in general this document does not discuss the security properties of the various algorithms, for information on these please consult one of the standard cryptography texts.

ProtectToolkit J is known to the JCA/JCE by the provider name **`SAFENET`**. To request an algorithm implemented by this provider, the string **`"SAFENET"`** should be passed to the **`getInstance()`** method.

Working With Slots

ProtectToolkit J is capable of interfacing to multiple adapters. This is achieved by using different "virtual providers" which map to each of the adapters. The virtual providers are named **"`SAFENET.n`"** where **n** is the slot number as configured with the ProtectToolkit C runtime tools. The special provider **"`SAFENET`"** always maps to the first slot.

A provider class exists (**SAFENETProvider**) for each of the slots in the package **au.com.safenet.crypto.provider.slot<n>**. These providers may be statically installed. Alternatively they may be added dynamically by calling the **`SAFENETProvider.addProviders()`** method.

Resources Management

NOTE: One important consideration when using the SafeNet provider is the management of resources. In general creation of a provider instance, e.g. a Cipher object, Key object, will result in the consumption of resources within the adapter. These resources are much less than that of the main JVM and so the garbage collection is not tuned to its needs. Unfortunately this means that it is up to the application programmer to manage.

There are two main techniques that may be employed. The first is to explicitly track resource usage and invoke garbage collector on certain thresholds. For example after the creation of 100 "session" Key objects which are only required for a short transaction then discarded it may be necessary to run the garbage collector to clean up those unused instances.

The second technique requires some tuning of the Cryptoki configuration on the adapter. If ProtectToolkit J cannot create a new "session" with the adapter it invokes the garbage collection (in the hope that there are some old unused sessions awaiting clean up). By reducing the maximum number of sessions allowed by the adapter it is possible to tune the adapter to the applications requirements so that explicit resource management is not required.

CHAPTER 2
INSTALLATION

The **`Provider`** may be statically installed into the Java Runtime Environment by adding an entry, similar to the following, into the **`java.security`** properties file located in **`$JAVA_HOME/lib/security/java.security`**

security.provider.2 = au.com.safenet.crypto.provider.SAFENETProvider

Alternatively, the **`Provider`** may be installed dynamically by an application at runtime by using the **`java.security.Security.addProvider()`** method, for example:

Security.addProvider(new au.com.safenet.crypto.provider.SAFENETProvider());

If the **`Provider`** is to be used on a specific Slot (as described in previous Working With Slots section) then the format for the above references should be:

au.com.safenet.crypto.provider.slot<n>.SAFENETProvider

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3
SUPPORTED CIPHERS

ProtectToolkit J includes support for symmetric block and stream ciphers as well as support for the asymmetric RSA cipher. The following algorithms through the `javax.crypto.Cipher` interface:

Cipher Name	Key Length (bits)	Block Size (bits)	Cipher Modes	Padding
DES	64	64	ECB,CBC	PKCS5Padding, NoPadding
DESede	128,192	64	ECB,CBC	PKCS5Padding, NoPadding
AES	128,182,256	64	ECB,CBC	PKCS5 Padding, NoPadding
IDEA	128	64	ECB,CBC	PKCS5Padding, NoPadding
CAST128	8-128	64	ECB,CBC	PKCS5Padding, NoPadding
RC2	0-1024	64	ECB,CBC	PKCS5Padding, NoPadding
RC4	8-2048	N/A	ECB	NoPadding
PBEWithMD2AndDES	64	64	N/A	N/A
PBEWithMD5AndDES	64	64	N/A	N/A
PBEWithMD5AndCAST	128	128	N/A	N/A
PBEWithSHA1AndCAST	128	128	N/A	N/A
PBEWithSHA1AndTripleDES	128	128	N/A	N/A
RSA	512-4096	variable	ECB	PCKS1Padding, NoPadding

Here, the Cipher name is the name of the Cipher as known to the JCE. To request a particular algorithm, pass this name to the `Cipher.getInstance()` method. Some algorithms support different key length and the supported key lengths are listed in the above table. The block size is the size of data that is processed by the cipher. During encryption the amount of data processed must be a multiple of this size (unless padding is employed see below) and the encrypted output will therefore be a multiple of this size.

The ECB mode is Electronic codebook mode and CBC is cipher block chaining as defined in FIPS PUB 81: DES Modes of Operation. All ciphers will default to ECB mode.

PKCS#5 padding is defined in PKCS#5 and is the standard padding applied to block ciphers with a block size of 64 bits. DES, DESede, IDEA, CAST128 and RC2 all default to "NoPadding". When PKCS5Padding is employed with a block cipher, the input data for encryption can be any length and will be padded to the appropriate length before encryption.

PKCS#1 padding is defined in PKCS#1 and is the standard padding mechanism for the RSA cipher. When this padding mechanism is used, PKCS#1 padding will be performed on each block encrypted. For public-key encryption PKCS#1 type 1 blocks will be created, and for private-key encryption type 2 blocks will be created. When “NoPadding” is requested, no PKCS#1 packing is applied to the data and the processing is performed as per the X.509 (raw) RSA specification.

DES

This algorithm is a 64-bit block cipher with a 64-bit key, however the effective key size is only 56-bit as 8 bits of the key are used for parity bits. The algorithm described in FIPS PUB 46-2 (see http://www.itl.nist.gov/div897/pubs/fip46-2.htm).

DES Cipher Initialisation

This cipher supports both `ECB` and `CBC` modes, and may be used with `NoPadding` or `PKCS5Padding`. To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the following strings as the transformation:

- `DES`
- `DES/ECB/NoPadding`
- `DES/ECB/PKCS5Padding`
- `DES/CBC/NoPadding`
- `DES/CBC/PKCS5Padding`

Using the "DES" transformation the `Cipher` will default to `ECB` and `NoPadding`.

If the `NoPadding` padding mode is selected the input data must be a multiple of 8 bytes, otherwise the encrypted or decrypted result will be truncated. In `PKCS5Padding` arbitrary data lengths are accepted, the cipher-text will be padded to a multiple of 8 bytes as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plain text is returned.

This `Cipher` will accept a `javax.crypto.spec.SecretKeySpec` or `au.com.safenet.crypto.provider.CryptokiSecretKey` as the key parameter during initialisation. For details on using keys with the DES `Cipher` see section 2.1.2.

When the `Cipher` is intialised in `CBC` mode the Initialisation Vector (IV) may be specified by passing a `javax.crypto.spec.IvParameterSpec` instance to the `Cipher.init()` method. When decrypting in this mode a valid IV must be specified in the `Cipher.init()` method, for encryption however a random IV will be generated if none is specified (the IV may be retrieved using the `Cipher.getIV()` method).

The IV may be provided as a `java.security.AlgorithmParameters or a javax.crypto.spec.IvParameterSpec` instance. If the initialisation is done using an `AlgorithmParameters` instance it must be convertible to an `IvParameterSpec` using the `AlgorithmParameters.getParameterSpec()` method.

This `Cipher` does not support the `Cipher.getParameters()` method, this method will always return `null`. The only supported parameter for this class is the initialisation vector which may be determined using the `Cipher.getIV()` method.

DES Key

The DES `Cipher` requires either a `SecretKeySpec` or ProtectToolkit J provider DES Key during initialisation.

To create an appropriate `SecretKeySpec` simple pass an 8 byte array and the algorithm name "`DES`" to the `SecretKeySpec` constructor. For example:

```
byte[] keyBytes = { 0x01, 0x23, 0x45, 0x67,
                    0x89, 0xAB, 0xCD, 0xEF };
SecretKeySpec desKey = new SecretKeySpec(keyBytes, "DES");
```

Alternatively, a random ProtectToolkit J DES key can be generated randomly using the `KeyGenerator` as described in section 2.1.3, or from a provider independent form as described in section 2.1.4. The DES key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J DES key will return the string "`DES`" as its algorithm name, "`RAW`" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The keys generated in ProtectToolkit J will always be marked as sensitive, however it is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

DES KeyGenerator

The DES `KeyGenerator` is used to generate random DES keys. The generated key will be a hardware key that has the Cryptoki `CKA_EXTRACTABLE` and `CKA_SENSITIVE` attributes set. Since these keys are marked as sensitive their `getEncoded()` method will return null.

During initialisation the `strength` and `random` parameters are ignored as all keys are 64-bits and the hardware includes a cryptographically-secure random source.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

DES SecretKeyFactory

The DES `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the DES key is the `javax.crypto.spec.DESKeySpec` class.

Keys generated using the `SecretKeyFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

For example, to create the provider based key from it's provider independent form:

```
byte[] keyBytes = { 0x01, 0x23, 0x45, 0x67,
                    0x89, 0xAB, 0xCD, 0xEF };
DESKeySpec desKeySpec = new DESKeySpec(keyBytes);
SecretKeyFactory desKeyFact =
              SecretKeyFactory.getInstance("DES", "SAFENET");
SecretKey desKey = desKeyFact.generateSecret(desKeySpec);
```

DES Example Code

The following example code will create a random DES key, then create a DES cipher in `CBC` mode with `PKCS5Padding`. Next it initialises the cipher for encryption using the newly created key, we then save the initialisation vector and encrypt the string "hello world".

To perform the decryption we re-initialize the cipher in decrypt mode, with the same key and the initialization vector that was created during encryption.

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES",
                                                "SAFENET");
Key desKey = keyGen.generateKey();
Cipher desCipher = Cipher.getInstance("DES/CBC/PKCS5Padding",
                                      "SAFENET");
desCipher.init(Cipher.ENCRYPT_MODE, desKey);
byte[] iv = desCipher.getIV();
byte[] cipherText = desCipher.doFinal(
                                    "hello world".getBytes());

desCipher.init(Cipher.DECRYPT_MODE, desKey,
                          new IvParameterSpec(iv));
byte[] plainText = desCipher.doFinal(cipherText);
```

DESede

This algorithm, known as triple-DES, is a 64-bit block cipher with a 192-bit key, although 24 bits of the key are parity bits. This algorithm works by splitting the 192-bit key into three 64-bit keys and then applying the basic DES cipher, firstly in the encrypt mode, secondly in the decrypt mode and finally in the encrypt mode. The algorithm is described in ANSI X9.17. It is also possible to use a double length key (128 bits), in this case the first key is re-used as the final key.

DESede Cipher Initialisation

This cipher supports both `ECB` and `CBC` modes, and may be used with `NoPadding` or `PKCS5Padding`. To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the following strings as the transformation:

- `DESede`
- `DESede/ECB/NoPadding`
- `DESede/ECB/PKCS5Padding`
- `DESede/CBC/NoPadding`
- `DESede/CBC/PKCS5Padding`

Using the "DESede" transformation the `Cipher` will default to `ECB` and `NoPadding`.

If the `NoPadding` padding mode is selected the input data must be a multiple of 8 bytes, otherwise the encrypted or decrypted result will be truncated. In `PKCS5Padding` arbitrary data lengths are accepted, the cipher-text will be padded to a multiple of 8 bytes as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plain text is returned.

This `Cipher` will accept a `javax.crypto.spec.SecretKeySpec` or `au.com.safenet.crypto.provider.CryptokiSecretKey` as the key parameter during initialisation. For details on using keys with the DESede `Cipher` see below for DESede Example Code.

When the `Cipher` is intialised in `CBC` mode the Initialisation Vector (IV) may be specified by passing a `javax.crypto.spec.IvParameterSpec` instance to the `Cipher.init()` method. When decrypting in this mode a valid IV must be specified in the `Cipher.init()` method, for encryption however a random IV will be generated if none is specified (the IV may be retrieved using the `Cipher.getIV()` method).

The IV may also be provided as a `java.security.AlgorithmParameters or a javax.crypto.spec.IvParameterSpec` instance. If the initialisation is done using an `AlgorithmParameters` instance it must be convertible to an `IvParameterSpec` using the `AlgorithmParameters.getParameterSpec()` method.

This `Cipher` does not support the `Cipher.getParameters()` method, this method will always return `null`. The only supported parameter for this class is the initialisation vector which may be determined using the `Cipher.getIV()` method.

DESede Key

The DESede `Cipher` requires either a `SecretKeySpec` or ProtectToolkit J provider DESede Key during initialisation. The DESede key may be either a double or triple length key.

To create an appropriate `SecretKeySpec` simple pass a 16 or 24-byte array and the algorithm name "`DESede`" to the `SecretKeySpec` constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec desEdeKey = new SecretKeySpec(keyBytes,
                                            "DESede");
```

Alternatively, a random ProtectToolkit J DESede key can be generated using the `KeyGenerator` as described in section 2.2.3, or from a provider independent form as described in section 2.2.4. The DESede key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J DESede key will return the string "`DESede`" as its algorithm name, "`RAW`" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The keys generated in ProtectToolkit J will always be marked as sensitive, however it is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

DESede KeyGenerator

The DESede `KeyGenerator` is used to generate random DESede double or triple length keys. The generated key will be a hardware key that has the Cryptoki `CKA_EXTRACTABLE` and `CKA_SENSITIVE` attributes set. Since these keys are marked as sensitive their `getEncoded()` method will return null.

During initialisation the `strength` parameter may be 128 to specify a double length key or 196 to specify a triple length key. If no strength is specified a triple length key will be generated. The `random` parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

DESede SecretKeyFactory

The DESede `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the DESede key is the `javax.crypto.spec.DESedeKeySpec` class.

Keys generated using the `SecretKeyFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

For example, to create the provider based key from it's provider independent form (in this case we are generating a triple length key, specify 16 bytes for a double length key):

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
DESedeKeySpec desEdeKeySpec = new DESedeKeySpec(keyBytes);
SecretKeyFactory desEdeKeyFact =
           SecretKeyFactory.getInstance("DESede", "SAFENET");
SecretKey desEdeKey =
           desEdeKeyFact.generateSecret(desEdeKeySpec);
```

DESede Example Code

See section 2.1.5 for the simple DES example, to convert the example to use DESede simply use "`DESede`" in place of "`DES`".

AES

This algorithm is an implementation of AES which is a 64bit block cipher with a variable length key, either 128, 192 or 256 bits long.

AES Cipher Initialisation

This cipher supports both `ECB` and `CBC` modes, and may be used with `NoPadding` or `PKCS5Padding`. To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the following strings as the transformation:

- `AES`
- `AES/ECB/NoPadding`
- `AES/CBC/NoPadding`
- `AES/CBC/PKCS5Padding`

Using the "AES" transformation the `Cipher` will default to `ECB` and `NoPadding`.

If the `NoPadding` padding mode is selected the input data must be a multiple of 8 bytes, otherwise the encrypted or decrypted result will be truncated. In `PKCS5Padding` arbitrary data lengths are accepted, the cipher-text will be padded to a multiple of 8 bytes as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plain text is returned.

This `Cipher` will accept a `javax.crypto.spec.SecretKeySpec` or `au.com.safenet.crypto.provider.CryptokiSecretKey` as the key parameter during initialisation. For details on using keys with the AES `Cipher` see section 2.3.2.

When the `Cipher` is intialised in `CBC` mode the Initialisation Vector (IV) may be specified by passing a `javax.crypto.spec.IvParameterSpec` instance to the `Cipher.init()` method. When decrypting in this mode a valid IV must be specified in the `Cipher.init()` method, for encryption however a random IV will be generated if none is specified (the IV may be retrieved using the `Cipher.getIV()` method).

The IV may also be provided as a `java.security.AlgorithmParameters or a javax.crypto.spec.IvParameterSpec` instance. If the initialisation is done using an `AlgorithmParameters` instance it must be convertible to an `IvParameterSpec` using the `AlgorithmParameters.getParameterSpec()` method.

This `Cipher` does not support the `Cipher.getParameters()` method, this method will always return `null`. The only supported parameter for this class is the initialisation vector which may be determined using the `Cipher.getIV()` method.

AES Key

The AES `Cipher` requires either a `SecretKeySpec` or ProtectToolkit J provider AES Key during initialisation. AES keys can either be 128, 192 or 256 bits long.

To create an appropriate `SecretKeySpec` simply pass a 16, 24 or 32 byte array and the algorithm name "AES" to the `SecretKeySpec` constructor.

For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xB6, 0xDC, 0x34,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec aesKey = new SecretKeySpec(keyBytes, "AES");
```

Alternatively, a random ProtectToolkit J AES key can be generated using the `KeyGenerator` as described in section 2.3.3, or, a provider independent form as described in section 2.3.4. The AES key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J AES key will return the string "`AES`" as its algorithm name, "`RAW`" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The keys generated in ProtectToolkit J will always be marked as sensitive, however it is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

AES KeyGenerator

The AES `KeyGenerator` is used to generate random AES keys. The generated key will be a hardware key that has the Cryptoki `CKA_EXTRACTABLE` and `CKA_SENSITIVE` attributes set. Since these keys are marked as sensitive their `getEncoded()` method will return null.

During initialisation the `strength` parameter may only be one 128, 192 or 256 bits, with the default size being 128 bits. The `random` parameter is ignored as the hardware includes a cryptographically-secure random source

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

AES SecretKeyFactory

The AES `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the AES key is the `au.com.safenet.crypto.spec.AESKeySpec` class.

Keys generated using the `SecretKeyFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

For example, to create the provider based key from it's provider independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
AESKeySpec ideaKeySpec = new AESKeySpec(keyBytes);
SecretKeyFactory aesKeyFact =
           SecretKeyFactory.getInstance("AES", "SAFENET");
SecretKey aesKey = aesKeyFact.generateSecret(aesKeySpec);
```

AES Example Code

See above DES Example Code for the simple DES example, to convert the example to use AES simply use "`AES`" in place of "`DES`".

IDEA

This algorithm is a 64-bit block cipher with a 128-bit key. This algorithm is patented in Europe and the United States by Ascom-Tech Ag, (see http://www.ascom.com/), however no license fee is required for non-commercial use. A description of the cipher may be found at http://www.ascom.ch/infosec/idea/techspecs.html.

IDEA Cipher Initialisation

This cipher supports both `ECB` and `CBC` modes, and may be used with `NoPadding` or `PKCS5Padding`. To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the following strings as the transformation:

- `IDEA`
- `IDEA/ECB/NoPadding`
- `IDEA/ECB/PKCS5Padding`
- `IDEA/CBC/NoPadding`
- `IDEA/CBC/PKCS5Padding`

Using the "IDEA" transformation the `Cipher` will default to `ECB` and `NoPadding`.

If the `NoPadding` padding mode is selected the input data must be a multiple of 8 bytes, otherwise the encrypted or decrypted result will be truncated. In `PKCS5Padding` arbitrary data lengths are accepted, the cipher-text will be padded to a multiple of 8 bytes as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plain text is returned.

This `Cipher` will accept a `javax.crypto.spec.SecretKeySpec` or `au.com.safenet.crypto.provider.CryptokiSecretKey` as the key parameter during initialisation. For details on using keys with the IDEA `Cipher` see section 2.3.2.

When the `Cipher` is intialised in `CBC` mode the Initialisation Vector (IV) may be specified by passing a `javax.crypto.spec.IvParameterSpec` instance to the `Cipher.init()` method. When decrypting in this mode a valid IV must be specified in the `Cipher.init()` method, for encryption however a random IV will be generated if none is specified (the IV may be retrieved using the `Cipher.getIV()` method).

The IV may also be provided as a `java.security.AlgorithmParameters or a javax.crypto.spec.IvParameterSpec` instance. If the initialisation is done using an `AlgorithmParameters` instance it must be convertible to an `IvParameterSpec` using the `AlgorithmParameters.getParameterSpec()` method.

This `Cipher` does not support the `Cipher.getParameters()` method, this method will always return `null`. The only supported parameter for this class is the initialisation vector which may be determined using the `Cipher.getIV()` method.

IDEA Key

The IDEA `Cipher` requires either a `SecretKeySpec` or ProtectToolkit J provider IDEA Key during initialisation. The IDEA key is always 128 bits long.

To create an appropriate `SecretKeySpec` simple pass a 16 byte array and the algorithm name "`IDEA`" to the `SecretKeySpec` constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xB6, 0xDC, 0x34,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec ideaKey = new SecretKeySpec(keyBytes, "IDEA");
```

Alternatively, a random ProtectToolkit J IDEA key can be generated using the `KeyGenerator` as described in section 2.3.3, or, a provider independent form as described in section 2.3.4. The IDEA key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J IDEA key will return the string "`IDEA`" as its algorithm name, "`RAW`" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The keys generated in ProtectToolkit J will always be marked as sensitive, however it is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

IDEA KeyGenerator

The IDEA `KeyGenerator` is used to generate random IDEA keys. The generated key will be a hardware key that has the Cryptoki `CKA_EXTRACTABLE` and `CKA_SENSITIVE` attributes set. Since these keys are marked as sensitive their `getEncoded()` method will return null.

During initialisation the `strength` and `random` parameters are ignored as all keys are 128-bits and the hardware includes a cryptographically-secure random source.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

IDEA SecretKeyFactory

The IDEA `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the IDEA key is the `au.com.safenet.crypto.spec.IDEAKeySpec` class.

Keys generated using the `SecretKeyFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

For example, to create the provider based key from it's provider independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
IDEAKeySpec ideaKeySpec = new IDEAKeySpec(keyBytes);
SecretKeyFactory ideaKeyFact =

           SecretKeyFactory.getInstance("IDEA", "SAFENET");

SecretKey ideaKey = ideaKeyFact.generateSecret(ideaKeySpec);
```

IDEA Example Code

See section 2.1.5 for the simple DES example, to convert the example to use IDEA simply use "`IDEA`" in place of "`DES`".

CAST128

This algorithm is an implementation of CAST-128 which is a 64-bit block cipher with a variable length key from 8 to 128 bits. The algorithm is described in RFC-2144, see http://www.ietf.org/rfc/rfc2144.txt.

CAST128 Cipher Initialisation

This cipher supports both `ECB` and `CBC` modes, and may be used with `NoPadding` or `PKCS5Padding`. To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the following strings as the transformation:

- `CAST128`
- `CAST128/ECB/NoPadding`
- `CAST128/ECB/PKCS5Padding`
- `CAST128/CBC/NoPadding`
- `CAST128/CBC/PKCS5Padding`

Using the "`CAST128`" transformation the `Cipher` will default to `ECB` and `NoPadding`.

If the `NoPadding` padding mode is selected the input data must be a multiple of 8 bytes, otherwise the encrypted or decrypted result will be truncated. In `PKCS5Padding` arbitrary data lengths are accepted, the cipher-text will be padded to a multiple of 8 bytes as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plain text is returned.

This `Cipher` will accept a `javax.crypto.spec.SecretKeySpec` or `au.com.safenet.crypto.provider.CryptokiSecretKey` as the key parameter during initialisation. For details on using keys with the CAST128 `Cipher` see CAST128 Key above

When the `Cipher` is intialised in `CBC` mode the Initialisation Vector (IV) may be specified by passing a `javax.crypto.spec.IvParameterSpec` instance to the `Cipher.init()` method. When decrypting in this mode a valid IV must be specified in the `Cipher.init()` method, for encryption however a random IV will be generated if none is specified (the IV may be retrieved using the `Cipher.getIV()` method).

The IV may also be provided as a `java.security.AlgorithmParameters or a javax.crypto.spec.IvParameterSpec` instance. If the initialisation is done using an `AlgorithmParameters` instance it must be convertible to an `IvParameterSpec` using the `AlgorithmParameters.getParameterSpec()` method.

This `Cipher` does not support the `Cipher.getParameters()` method, this will method return `null`. The only supported parameter for this class is the initialisation vector which may be determined using the `Cipher.getIV()` method.

CAST128 Key

The CAST128 `Cipher` requires either a `SecretKeySpec` or ProtectToolkit J provider CAST128 Key during initialisation. The CAST128 key may be any length of 8 to 128 bits.

To create an appropriate `SecretKeySpec` simple pass an array of up to 16 bytes and the algorithm name "`CAST128`" to the `SecretKeySpec` constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec castKey = new SecretKeySpec(keyBytes,
                                          "CAST128");
```

Alternatively, a random ProtectToolkit J CAST128 key can be generated using the `KeyGenerator` as described in section 2.4.3, or from a provider independent form as described in section 2.4.4. The CAST128 key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J CAST128 key will return the string "`CAST128`" as its algorithm name, "`RAW`" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The keys generated in ProtectToolkit J will always be marked as sensitive, however it is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

CAST128 KeyGenerator

The CAST128 `KeyGenerator` is used to generate random CAST128. The generated key will be a hardware key that has the Cryptoki `CKA_EXTRACTABLE` and `CKA_SENSITIVE` attributes set. Since these keys are marked as sensitive their `getEncoded()` method will return null.

During initialisation the `strength` parameter may be any length from 8 to 128. The default key size is 128 bits. The `random` parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

CAST128 SecretKeyFactory

The CAST128 `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the CAST128 key is the `au.com.safenet.crypto.spec.CASTKeySpec` class.

Keys generated using the `SecretKeyFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

For example, to create the provider based key from it's provider independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
CAST128KeySpec castKeySpec = new CAST128KeySpec(keyBytes);
SecretKeyFactory castKeyFact =
           SecretKeyFactory.getInstance("CAST128", "SAFENET");
SecretKey castKey=castKeyFact.generateSecret(castEdeKeySpec);
```

CAST128 Example Code

See DES Example Code for the simple DES example, to convert the example to use CAST128 simply use "`CAST128`" in place of "`DES`".

RC2

This algorithm is a 64-bit block cipher with a variable length key usually 40-bit or 128-bit. RC2 was designed by Ron Rivest and is a trademark of RSA Data Security. For more information on this algorithm see RFC-2268.

RC2 Cipher Initialisation

This cipher supports both `ECB` and `CBC` modes, and may be used with `NoPadding` or `PKCS5Padding`. To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the following strings as the transformation:

- `RC2`
- `RC2/ECB/NoPadding`
- `RC2/ECB/PKCS5Padding`
- `RC2/CBC/NoPadding`
- `RC2/CBC/PKCS5Padding`

Using the "RC2" transformation the `Cipher` will default to `ECB` and `NoPadding`.

If the `NoPadding` padding mode is selected the input data must be a multiple of 8 bytes, otherwise the encrypted or decrypted result will be truncated. In `PKCS5Padding` arbitrary data lengths are accepted, the cipher-text will be padded to a multiple of 8 bytes as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plain text is returned.

This `Cipher` will accept a `javax.crypto.spec.SecretKeySpec` or `au.com.safenet.crypto.provider.CryptokiSecretKey` as the key parameter during initialisation. For details on using keys with the RC2 `Cipher` see section 2.5.2.

The RC2 `Cipher` may also be initialised with an instance of the `javax.crypto.spec.RC2ParameterSpec` class. With this class it is possible to supply an initialisation vector and an effective key size. If the `Cipher` is not initialised in this way the effective key size will default to 128.

The IV may also be provided as a `java.security.AlgorithmParameters` instance. If the initialisation is done using an `AlgorithmParameters` instance, it must be convertible to an `IvParameterSpec` using the `AlgorithmParameters.getParameterSpec()` method.

This `Cipher` does not support the `Cipher.getParameters()` method, this method will always return `null`. The only supported parameter for this class is the initialisation vector which may be determined using the `Cipher.getIV()` method.

RC2 Key

The RC2 `Cipher` requires either a `SecretKeySpec` or ProtectToolkit J provider RC2 Key during initialisation. The RC2 key may be any length of 8 to 1024 bits.

To create an appropriate `SecretKeySpec` simple pass an array of up to 128 bytes and the algorithm name "`RC2`" to the `SecretKeySpec` constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec rc2Key = new SecretKeySpec(keyBytes, "RC2");
```

Alternatively, a random ProtectToolkit J RC2 key can be generated using the `KeyGenerator` as described in section 2.5.3, or from a provider independent form as described in section 2.5.4. The RC2 key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J RC2 key will return the string "`RC2`" as its algorithm name, "`RAW`" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The keys generated in ProtectToolkit J will always be marked as sensitive, however it is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

RC2 KeyGenerator

The RC2 `KeyGenerator` is used to generate random RC2. The generated key will be a hardware key that has the Cryptoki `CKA_EXTRACTABLE` and `CKA_SENSITIVE` attributes set. Since these keys are marked as sensitive their `getEncoded()` method will return null.

During initialisation the `strength` parameter may be any multiple of 8 of 8 to 1024 inclusive. The default key size is 128 bits. The `random` parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

RC2 SecretKeyFactory

The RC2 `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the RC2 key is the `au.com.safenet.crypto.spec.RC2KeySpec` class.

Keys generated using the `SecretKeyFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

For example, to create the provider based key from it's provider independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
RC2KeySpec rc2KeySpec = new RC2KeySpec(keyBytes);
SecretKeyFactory rc2KeyFact =
           SecretKeyFactory.getInstance("RC2", "SAFENET");
SecretKey rc2Key = rc2KeyFact.generateSecret(castEdeKeySpec);
```

RC2 Example Code

See section 2.1.5 for the simple DES example, to convert the example to use RC2 simply use "`RC2`" in place of "`DES`".

Replace the `IvParameterSpec` call with the `RC2ParameterSpec` call, as illustrated in the following code example:

```
KeyGenerator keyGen = KeyGenerator.getInstance("RC2","SAFENET");
Key rcKey = keyGen.generateKey();
Cipher rc2Cipher = Cipher.getInstance("RC2/CBC/PKCS5Padding","SAFENET");
rc2Cipher.init(Cipher.ENCRYPT_MODE, rcKey);
byte[] iv = rc2Cipher.getIV();
byte[] cipherText = rc2Cipher.doFinal("hello world".getBytes());
rc2Cipher.init(Cipher.DECRYPT_MODE, rcKey,new RC2ParameterSpec(iv));
byte[] plainText = rc2Cipher.doFinal(cipherText);
```

RC4

This algorithm is a stream cipher with a variable length key usually 40-bit or 128-bit. RC4 is a trademark of RSA Data Security. A description of the algorithm may be found in *Applied Cryptography* by Bruce Schneier.

RC4 Cipher Initialisation

Since the RC4 Cipher is a stream cipher it always operates in the same mode which may be specified by the transformations "`RC4`" or "`RC4/ECB/NoPadding`". To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the valid transformation strings.

The size of the output of this cipher will always be the same as that of the input.

This `Cipher` will accept a `javax.crypto.spec.SecretKeySpec` or `au.com.safenet.crypto.provider.CryptokiSecretKey` as the key parameter during initialisation. For details on using keys with the RC4 `Cipher` see section 2.6.2.

This `Cipher` does not support initialisation with algorithm parameters and so the `Cipher.getParameters()` method, this will always return `null`.

RC4 Key

The RC4 `Cipher` requires either a `SecretKeySpec` or ProtectToolkit J provider RC4 Key during initialisation. The RC4 key may be any length of 8 to 2048 bits.

To create an appropriate `SecretKeySpec` simple pass an array of up to 256 bytes and the algorithm name "`RC4`" to the `SecretKeySpec` constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };

SecretKeySpec desKey = new SecretKeySpec(keyBytes, "RC4");
```

Alternatively, a random ProtectToolkit J RC4 key can be generated using the `KeyGenerator` as described in section 2.6.3, or, a provider independent form as described in section 2.6.4. The RC4 key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J RC4 key will return the string "`RC4`" as its algorithm name, "`RAW`" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The keys generated in ProtectToolkit J will always be marked as sensitive, however it is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

RC4 KeyGenerator

The RC4 `KeyGenerator` is used to generate random RC4. The generated key will be a hardware key that has the Cryptoki `CKA_EXTRACTABLE` and `CKA_SENSITIVE` attributes set. Since these keys are marked as sensitive their `getEncoded()` method will return null.

During initialisation the `strength` parameter may be any length from 8 to 2048. The default key size is 128 bits. The `random` parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

RC4 SecretKeyFactory

The RC4 `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the RC4 key is the `au.com.safenet.crypto.spec.RC4KeySpec` class.

Keys generated using the `SecretKeyFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

For example, to create the provider based key from it's provider independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
RC4KeySpec castKeySpec = new RC4KeySpec(keyBytes);
SecretKeyFactory castKeyFact =
           SecretKeyFactory.getInstance("RC4", "SAFENET");
SecretKey castKey=castKeyFact.generateSecret(castEdeKeySpec);
```

RC4 Example Code

The following example code will create a random RC4 key, then create a RC4 cipher. Next it initialises the cipher for encryption using the newly created key, we then save the initialisation vector and encrypt the string "hello world".

To perform the decryption we simply re-initialise the cipher in decrypt mode, with the same key. In this case there is no need to process the initialisation vector as there is none with the RC4 algorithm.

```
KeyGenerator keyGen = KeyGenerator.getInstance("RC4",
                                                "SAFENET");
Key rc4Key = keyGen.generateKey();
Cipher rc4Cipher = Cipher.getInstance("RC4", "SAFENET");
rc4Cipher.init(Cipher.ENCRYPT_MODE, rc4Key);
byte[] cipherText = rc4Cipher.doFinal(
                                    "hello world".getBytes());

rc4Cipher.init(Cipher.DECRYPT_MODE, rc4Key);
byte[] plainText = desCipher.doFinal(cipherText);
```

PBE Ciphers

A PBE Cipher is a password based cipher. It provides for keying of a cipher based on a user supplied password. PKCS#5 is the standard which defines the generic PBE algorithm used by all the PBE algorithms except for the PBEWithSHA1AndTripleDES algorithm which uses PKCS#12 (see http://www.rsasecurity.com/rsalabs/pkcs/pkcs-12/index.html). A particular PBE implementation will combine a message digest algorithm (such as MD5) with a symmetric encryption algorithm (DES for example).

ProtectToolkit J includes five password based Ciphers, `PBEWithMD2AndDES`, `PBEWithMD5AndDES` , `PBEWithMD5AndCAST`, `PBEWithSHA1AndCAST`, and `PBEWithSHA1AndTripleDES`.These ciphers are essentially identical, the only difference being that the first uses MD2 in the password generation, the second two MD5 and the last two MD5. As the names suggest these ciphers use either DES, CAST or TripleDES as their encryption algorithm, and so are 64-bit block ciphers. They are all operated with the block cipher in CBC mode however the initialisation vector is determined from the password so there is no need to supply its value.

PBE Cipher Initialization

A PBE Cipher will always operate with the underlying `Cipher` in a specific mode. For ProtectToolkit J the DES `Cipher` will operate in `CBC` mode with `PCKS5Padding`. Thus the only valid transformations that may be passed to the Cipher.getInstance() method are `PBEWithMD2AndDES`, `PBEWithMD5AndDES` , `PBEWithMD5AndCAST`, `PBEWithSHA1AndCAST`, or `PBEWithSHA1AndTripleDES`.

This `Cipher` will only accept a ProtectToolkit J provider PBE key as the key parameter during initialisation. To create such a `Key` use the PBE `SecretKeyFactory` in section 2.6.2.

This `Cipher` also requires initialisation with a valid `PBEParameterSpec` instance, (or an `AlgorithmParameters` instance that can be converted to the generic form via the `getParameterSpec()` method). This parameters instance is used to supply the salt and iteration count parameters to the PBE `Cipher`. This is a required parameter, there are no defaults and so the `Cipher.getParameters()` method, this will always return `null`.

PBE Key

The PBE `Cipher` instances require initialisation with a ProtectToolkit J provider PBE key. Instances of this type may be created using the PBE SecretKeyFactory. The PBE `SecretKeyFactory` is used to construct ProtectToolkit J keys from their provider-independent form. The provider independent form of the PBE key is the `javax.crypto.spec.PBEKeySpec` class.

For example, to create the provider based key from it's provider independent form:

```
PBEKeySpec pbeKS = new PBEKeySpec("password".toCharArray())
SecretKeyFactory pbeKF = SecretKeyFactory.getInstance("PBE",
                                                      "SAFENET");
Key key = pbeKF.generateSecret(pbeKS);
```

The ProtectToolkit J PBE key will return the string "`PBE`" as its algorithm name, "`RAW`" as its encoding. However, this key class does not support encoding and so will return `null` from the `getEncoded()` method.

PBE Example Code

The following example code will create a PBE key with the string "password", convert this into a ProtectToolkit J PBE key, then create a PBE cipher. Next it initialises the cipher for encryption using the newly created key and the PBE parameters with a salt of "salt" and an iteration count of 5. Finally we encrypt the string "hello world".

To perform the decryption we simply re-initialise the cipher in decrypt mode, with the same key and parameters.

```
PBEKeySpec pbeKS = new PBEKeySpec("password".toCharArray())
SecretKeyFactory pbeKF = SecretKeyFactory.getInstance("PBE",
                                                      "SAFENET");
Key pbeKey = pbeKF.generateSecret(pbeKS);
PBEParameterSpec pbeParams =
                    new PBEParameterSpec("salt".getBytes, 5);

Cipher pbeCipher = Cipher.getInstance("PBEWithMD5andDES",
                                                      "SAFENET");
pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParams);
byte[] cipherText = pbeCipher.doFinal(
                              "hello world".getBytes());
pbeCipher.init(Cipher.DECRYPT_MODE, pbeKey, pbeParams);
byte[] plainText = pbeCipher.doFinal(cipherText);
```

RSA

This algorithm is a block cipher with a variable length key whose block size is equal to the key size. RSA is patented in the United States by RSA Data Security. The RSA cipher will operate in one of two modes depending on the padding requested. If "`PKCS1Padding`" is requested the processing is performed as described in PKCS#1. If "`NoPadding`" is requested then the processing is performed as specified in X.509 for raw RSA.

NOTE: Currently the RSA `Cipher` only supports encryption or decryption of a single block. Any attempt to pass more data than a single block will result in a `RuntimeException`.

RSA Cipher Initialisation

This cipher supports both only `ECB` mode, and may be used with `NoPadding` or `PKCS1Padding`. To create an instance of this class use the `Cipher.getInstance()` method with "`SAFENET`" as the provider and one of the following strings as the transformation:

- `RSA`
- `RSA/ECB/NoPadding`
- `RSA/ECB/PKCS1Padding`

Using the "RSA" transformation the `Cipher` will default to `ECB` and `PKCS1Padding`. The `NoPadding` option will result in "RAW" RSA, where each block will be 0 padded.

The block size of this cipher is dependent on the key size in use. The block size is equal to the number of bytes of the RSA modulus. If the modulus is *k* bytes long then the encrypted output size is always *k*. For the "`NoPadding`" mode the plain text input must be equal to or less than *k*, with the "`PKCS1Padding`" mode the plain text input must be equal to or less than *k*-11 bytes.

This `Cipher` will only accept a ProtectToolkit J provider based key during initialisation. This key must be generated by the ProtectToolkit J RSA `KeyFactory`, `KeyPairGenerator` or `KeyStore`. For details on using keys with the RSA `Cipher` see RSA Key.

This `Cipher` does not support initialisation with algorithm parameters and so the `Cipher.getParameters()` method, this will always return `null`.

RSA Key

The RSA `Cipher` requires either a ProtectToolkit J RSA public, or private Key during initialisation. The RSA key may be any length between 512 and 4096 bits (inclusive).

A new ProtectToolkit J RSA key can be generated randomly using the `KeyPairGenerator` as described in section 2.8.3, or, a from provider independent form as described in section 2.8.4. The RSA key may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J RSA key will return the string "`RSA`" as its algorithm name, the public key type will return "`X.509`" as its encoding (the private key types will return "`RAW`") as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available (private keys will return `null` from the `getEncoded()` method). If the public key is available the getEncoded() method will return the key as a DER encoded X.509 `SubjectPublicKeyInfo` block containing the public key as defined in PKCS#1.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as sensitive. The public keys generated in ProtectToolkit J will not be marked as sensitive, and the private keys generated in ProtectToolkit J will always be marked as sensitive. It is possible to access any Cryptoki keys stored on the device and it is possible that the attributes of these keys have been modified.

RSA KeyGenerator

The RSA `KeyPairGenerator` is used to generate random RSA key pairs. The generated key pair will consist of two hardware keys, the public key and a private key with the Cryptoki `CKA_SENSITIVE` attribute set. The public exponent for this key generator is fixed to the Fermat-4 value (hex 0x100001).

During initialisation the `strength` parameter may be any length from 512 to 4096. The default key size is 1024 bits. The `random` parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `Signature` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

RSA KeyPairFactory

The RSA `KeyPairFactory` is used to construct ProtectToolkit J keys from their provider-independent form. There are three standard provider independent forms for RSA keys, one for public keys, and two for private keys. They are `java.security.spec.RSAPublicKeySpec`, `java.security.spec.RSAPrivateKeySpec, and java.security.spec.RSAPrivateCrtKeySpec`

Additionally there is the `au.com. safenet.crypto.spec.AsciiEncodedKeySpec` class which can be used for keys encoded as hexadecimal strings. For more information on this `KeySpec` see section 8.3.

Keys generated using the `KeyPairFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

To convert one of these supported KeySpec classes into a ProtectToolkit J provider key:

```
KeyFactory rsaKeyFact = KeyFactory.getInstance("RSA",
                                                "SAFENET");
PublicKey pubKey = rsaKeyFact.generatePublic(pubKeySpec);
PrivateKey privKey = rsaKeyFact.generatePrivate(privKeySpec);
```

The RSA `KeyFactory` cannot currently convert ProtectToolkit J keys into their provider independent format so the `getKeySpec()` method will throw an `InvalidKeySpecException`. The class also cannot perform any key translation via the `translateKey()` method.

RSA Example Code

The following example code will create a random RSA key pair, then create a RSA cipher in `ECB` mode with `PKCS1Padding`. Next it initialises the cipher for encryption using the public key from newly created key pair, finally we encrypt the string "hello world".

To perform the decryption we re-initialise the cipher in decrypt mode, with the private key from the key pair.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA",
                                                       "SAFENET");
KeyPair rsaPair = keyGen.generateKeyPair();
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding",
                                      "SAFENET");
rsaCipher.init(Cipher.ENCRYPT_MODE, rsaPair.getPublic());
byte[] cipherText = rsaCipher.doFinal(
                                    "hello world".getBytes());

rsaCipher.init(Cipher.DECRYPT_MODE, rsaPair.getPrivate());
byte[] plainText = rsaCipher.doFinal(cipherText);
```

CHAPTER 4
CIPHER ALGORITHM PARAMETERS

Currently, ProtectToolkit J does not support algorithm parameters.

Calls to `Cipher.getParameters()` will always return `null`. Additionally, the provider does not include any `java.security.AlgorithmParameters` classes.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5
SUPPORTED SIGNATURE ALGORITHMS

The following Signature algorithms are available with the `Provider` through the `java.security.Signature` interface.

- MD2withRSA
- MD5withRSA
- SHA1withRSA
- SHA224withRSA
- SHA256withRSA
- SHA384withRSA
- SHA512withRSA
- SHA1withDSA
- PKCS#1RSA
- X.509RSA
- DSARaw
- RIPEMD128withRSA
- RIPEMD160withRSA

MD2withRSA

This Signature class implements the algorithm "MD2 with RSA" as defined in PKCS#1. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block and then finally RSA encrypt the DER encoded block.

To initialise this requires a ProtectToolkit J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher (section 2.8) for information on RSA keys.

This algorithm is provided for compatibility only, newer applications should use either MD5withRSA or SHA1withRSA.

The following example will sign the message "hello world" with a pre-existing RSA private key and then verify it with the corresponding public key.

```
KeyPair rsaPair; // pre existing key pair
Signature rsaSig = Signature.getInstance("MD2withRSA", "SAFENET");
rsaSig.initSign(rsaPair.getPrivate());
rsaSig.update("hello world".getBytes());
byte[] sig = rsaSig.sign();

rsaSig.initVerify(rsaPair.getPublic());
rsaSig.update("hello world".getBytes());
if (rsaSig.verify(sig)) {
   System.out.println("Signature okay");
}
else {
   System.out.println("Signature fails verification");
}
```

MD5withRSA

This Signature class implements the algorithm "MD5 with RSA" as defined in PKCS#1. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block and then finally RSA encrypt the DER encoded block.

To initialise this requires a ProtectToolkit J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher (section 2.8) for information on RSA keys.

See section 4.1 for a simple example on using this algorithm, simply modify the algorithm name used to "`MD5withRSA`".

SHA1withRSA

This Signature class implements the algorithm "RSASSA-PKCS1-v1_5" as defined in PKCS#1. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block and then finally RSA encrypt the DER encoded block.

To initialise this requires a ProtectToolkit J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher (section 2.8) for information on RSA keys.

Where there is no requirement for backwards compatibility this is the recommended RSA signature algorithm to use as there are no known weaknesses in the SHA1 message digest algorithm whereas known weaknesses exist in the MD2 and MD5 message digest algorithms.

See section 4.1 for a simple example on using this algorithm, simply modify the algorithm name used to "`SHA1withRSA`".

SHA224withRSA

This signature class is similar to SHA1withRSA, except it produces a signature from a digest length of 224 bits.

See MD2withRSA for a simple example on using this algorithm; simply modify the algorithm name used to "`SHA224withRSA`".

SHA256withRSA

This signature class is similar to SHA1withRSA, except it produces a signature from a digest length of 256 bits.

See MD2withRSA for a simple example on using this algorithm; simply modify the algorithm name used to "`SHA256withRSA`".

SHA384withRSA

This signature class is similar to SHA1withRSA, except it produces a signature from a digest length of 384 bits.

See MD2withRSA for a simple example on using this algorithm; simply modify the algorithm name used to "`SHA384withRSA`".

SHA512withRSA

This signature class is similar to SHA1withRSA, except it produces a signature from a digest length of 512 bits.

See MD2withRSA for a simple example on using this algorithm, simply modify the algorithm name used to "`SHA512withRSA`".

SHA1withDSA

This Signature class implements the Digital Signature Algorithm (DSA) as defined in FIPS PUB 186, which is also compatible with the Sun provided Signature algorithm of the same name. This algorithm will perform a message digest (using SHA1) of the data to be signed and then sign that data using DSA. The result of a sign operation using this algorithm will be a DER encoded block containing a sequence of the two integer values r and s.

To initialise this requires a ProtectToolkit J DSA key, either a private key for signing or a public key for signature verification. The following sections detail how to generate ProtectToolkit J provider DSA keys.

DSA Key

The DSA `Signature` requires either a ProtectToolkit J DSA public, or private Key during initialisation. The DSA key may be any length between 512 and 4096 bits (inclusive).

A new ProtectToolkit J DSA key pair can be generated randomly using the `KeyPairGenerator` as described in section 4.5.2, or, a from provider independent form as described in section 4.5.3. The DSA keys may also be stored in the ProtectToolkit J KeyStore as described in section 8.1.

The ProtectToolkit J DSA public and private keys will return the string "`DSA`" as the algorithm name, "`RAW`" as the encoding type and `null` for the encoding.

DSA KeyGenerator

The DSA `KeyPairGenerator` is used to generate random DSA key pairs. The generated key pair will consist of two hardware keys, the public key and a private key with the Cryptoki `CKA_SENSITIVE` attribute set. Each key will also share the same set of DSA parameters.

During initialisation the `strength` parameter may be either 512 or 4096. The default key size is 1024 bits. The `random` parameter is ignored as the hardware includes a cryptographically-secure random source. Any provided `AlgorithmParameterSpec` parameters will also be ignored (this precludes generation of keys with non-default parameters). The DSA parameters used for the 512 and 1024 bit keys are as specified in the Java Cryptography Architecture Specification.

Keys generated using the `KeyGenerator` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Signature` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

The following example will generate a new random 1024 bit key pair:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance(
       "DSA", "SAFENET");
KeyPair dsaPair = keyGen.generateKeyPair();
```

DSA KeyPairFactory

The DSA `KeyPairFactory` is used to construct ProtectToolkit J keys from their provider-independent form. There are two standard provider independent forms for DSA keys, one for public keys, and one for private keys. They are `java.security.spec.DSAPublicKeySpec, and java.security.spec.DSAPrivateKeySpec.`

Keys generated using the `KeyPairFactory` are not thread-safe. That is, a ProtectToolkit J Key instance may only be used by a single `Cipher` instance (as well as a single `MAC` instance) at any given time. See section 7.0 for a discussion on threading and ProtectToolkit J keys.

To convert one of these supported KeySpec classes into a ProtectToolkit J provider key:

```
KeyFactory dsaKeyFact = KeyFactory.getInstance("DSA",
                                                "SAFENET");
PublicKey pubKey = dsaKeyFact.generatePublic(pubKeySpec);
PrivateKey privKey = dsaKeyFact.generatePrivate(privKeySpec);
```

The DSA `KeyFactory` cannot currently convert ProtectToolkit J keys into their provider independent format so the `getKeySpec()` method will throw an `InvalidKeySpecException`. The class also cannot perform any key translation via the `translateKey()` method.

DSA Example Code

The following example code will create a random DSA key pair, then create a DSA `Signature`. We will then use this instance to sign the message “hello world” and then verify that signature using the public key.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA",
                                                       "SAFENET");
KeyPair rsaPair = keyGen.generateKeyPair();
Signature dsaSig = Signature.getInstance("DSA",
                                         "SAFENET");
dsaSig.initSign(dsaPair.getPrivate());
dsaSig.update(“hello world”.getBytes());
byte[] sig = dsaSig.sign();

dsaSig.initVerify(dsaPair.getPublic();
dsaSig.update(“hello world”.getBytes());
if (dsaSig.verify()) {
    System.out.println(“Signature okay”);
}
else {
    System.out.println(“Signature fails verification”);
}
```

PKCS#1RSA

This signature algorithm will produce a PKCS#1 encoded block (block type 01) containing the private-key encrypted message. The message length must be `k-11` bytes long where `k` is the length of the RSA modulus. The generated signature will be `k` bytes long.

X.509RSA

This signature algorithm will perform "raw" RSA exponentiation on the input message by converting it to an integer, most-significant byte first, and converting the result to a byte string, most-significant byte first. The input message, considered as an integer, must be less than the modulus. Where necessary the input message is padded by prepending the message with 0-valued bytes.

This algorithm is intended for compatibility with applications that do not follow the PKCS#1 block format.

DSARaw

This signature algorithm will perform "raw" RSA exponentiation on the input message by converting it to an integer, most-significant byte first, and converting the result to a byte string, most-significant byte first. The input message, considered as an integer, must be less than the modulus. Where necessary the input message is padded by prepending the message with 0-valued bytes.

This algorithm is intended for compatibility with applications that do not follow the PKCS#1 block format.

RIPEMD128withRSA

This Signature class implements the algorithm "MD5 with RSA" as defined in PKCS#1 however it uses the message digest algorithm RIPEMD128 in place of MD5. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block and then finally RSA encrypt the DER encoded block.

To initialise this requires a ProtectToolkit J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher (section 2.8) for information on RSA keys.

See MD2withRSA above for a simple example on using this algorithm, simply modify the algorithm name used to "`RIPEMD128withRSA`".

RIPEMD160withRSA

This Signature class implements the algorithm "MD5 with RSA" as defined in PKCS#1 however it uses the message digest algorithm RIPEMD160 in place of MD5. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block and then finally RSA encrypt the DER encoded block.

To initialise this requires a ProtectToolkit J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher (section 2.8) for information on RSA keys.

See section 4.1 for a simple example on using this algorithm, simply modify the algorithm name used to "`RIPEMD128withRSA`".

CHAPTER 6
SUPPORTED MAC ALGORITHMS

The following MAC algorithms are available with the `Provider` through the `javax.crypto.Mac` interface.

- DES
- DESede
- DESedeX919
- IDEA
- CAST128
- RC2
- HMAC/MD2
- HMAC/MD5
- HMAC/SHA1
- HMAC/SHA224
- HMAC/SHA256
- HMAC/SHA384
- HMAC/SHA512

DES MAC

This algorithm is compatible with FIPS PUB 113 (see http://www.itl.nist.gov/div897/pubs/fip113.htm) as well as ANSI X9.9.

The MAC may be initialized using any valid DES key (see section 2.1.2). The result MAC value will be a four byte array.

DESede MAC

This algorithm is compatible with FIPS PUB 113 (see http://www.itl.nist.gov/div897/pubs/fip113.htm).

The MAC may be initialised using any valid DESede key (see section 2.2.2). The result MAC value will be a four byte array.

DESedeX919 MAC

This MAC implements the triple DES MAC algorithm as defined in X9.19 (or ISO 9807).

The MAC may be initialised using any valid DESede key (see section 2.2.2). The result MAC value will be a four byte array.

IDEA MAC

This algorithm is compatible with FIPS PUB 113 (see http://www.itl.nist.gov/div897/pubs/fip113.htm).

The MAC may be initialized using any valid IDEA key (see section 2.3.2). The result MAC value will be a four byte array.

CAST128 MAC

This algorithm is compatible with FIPS PUB 113 (see http://www.itl.nist.gov/div897/pubs/fip113.htm).

The MAC may be initialised using any valid CAST128 key (see section 2.4.2). The result MAC value will be a four byte array.

RC2

This algorithm is compatible with FIPS PUB 113 (see http://www.itl.nist.gov/div897/pubs/fip113.htm).

The MAC may be initialized using any valid RC2 key (see section 2.5.2). The result MAC value will be a four byte array.

HMAC/MD2

This HMAC implements the HMAC algorithm as defined in RFC 2104 (see http://www.ietf.org/rfc/rfc2194.txt) using the message digest function MD2. The result MAC value will be a 16 byte array.

The MAC may be initialized using a SecretKeySpec with the algorithm name "HMAC/MD2". It is also possible to initialize this MAC using any of the secret keys generated by one of the KeyGenerator classes or KeyFactory classes as detailed in section 2.

HMAC/MD5

This HMAC implements the HMAC algorithm as defined in RFC 2104 (see http://www.ietf.org/rfc/rfc2194.txt) using the message digest function MD5. The result MAC value will be a 16 byte array.

The MAC may be initialized using a SecretKeySpec with the algorithm name "HMAC/MD5". It is also possible to initialize this MAC using any of the secret keys generated by one of the KeyGenerator classes or KeyFactory classes as detailed in section 2.

HMAC/SHA1

This HMAC implements the HMAC algorithm as defined in RFC 2104 (see http://www.ietf.org/rfc/rfc2194.txt) using the message digest function SHA1. The result MAC value will be a 20 byte array.

The MAC may be initialized using a SecretKeySpec with the algorithm name "HMAC/SHA1". It is also possible to initialise this MAC using any of the secret keys generated by one of the KeyGenerator classes or KeyFactory classes as detailed in section 2.

HMAC/SHA224

This HMAC implements the HMAC algorithm as defined in RFC 2104 (see http://www.ietf.org/rfc/rfc2194.txt) using the message digest function SHA224. The result MAC value will be a 28 byte array.

The MAC may be initialized using a SecretKeySpec with the algorithm name "HMAC/SHA224". It is also possible to initialise this MAC using any of the secret keys generated by one of the KeyGenerator classes or KeyFactory classes as detailed in section 2.

HMAC/SHA256

This HMAC implements the HMAC algorithm as defined in RFC 2104 (see http://www.ietf.org/rfc/rfc2194.txt) using the message digest function SHA256. The result MAC value will be a 32 byte array.

The MAC may be initialized using a SecretKeySpec with the algorithm name "HMAC/SHA256". It is also possible to initialise this MAC using any of the secret keys generated by one of the KeyGenerator classes or KeyFactory classes as detailed in section 2.

HMAC/SHA384

This HMAC implements the HMAC algorithm as defined in RFC 2104 (see http://www.ietf.org/rfc/rfc2194.txt) using the message digest function SHA384. The result MAC value will be a 48 byte array.

The MAC may be initialized using a SecretKeySpec with the algorithm name "HMAC/SHA384". It is also possible to initialise this MAC using any of the secret keys generated by one of the KeyGenerator classes or KeyFactory classes as detailed in section 2.

HMAC/SHA512

This HMAC implements the HMAC algorithm as defined in RFC 2104 (see http://www.ietf.org/rfc/rfc2194.txt) using the message digest function SHA512. The result MAC value will be a 64 byte array.

The MAC may be initialized using a SecretKeySpec with the algorithm name "HMAC/SHA512". It is also possible to initialise this MAC using any of the secret keys generated by one of the KeyGenerator classes or KeyFactory classes as detailed in section 2.

Sample MAC Code

This simple code fragment will generate a MAC code (based on a randomly generated DES key) for the bytes in the string "hello world".

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES", "SAFENET");
Key desKey = keyGen.generateKey();
Mac desMac = Mac.getInstance("DES", "SAFENET");
desMac.init(desKey);

byte[] mac = desMac.doFinal("hello world".getBytes());
```

CHAPTER 7
SUPPORTED MESSAGE DIGEST ALGORITHMS

The following standard message digest algorithms are supported by the `Provider` through the `java.security.MessageDigest` interface.

Message Digest Name	Digest Length (bits)
MD2	128
MD5	128
SHA-1	160
SHA-224	224
SHA-256	256
SHA-384	384
SHA-512	512
RIPEMD128	128
RIPEMD160	160

MD2

This message digest algorithm produces a 128-bit digest. The algorithm is described in RFC-1319, see http://www.ietf.org/rfc/rfc1319.txt. This algorithm is provided for compatibility only and is not recommended for other purposes. Instances of this algorithm are not clone-able.

To create a MD2 message digest for the message "hello world":

```
MessageDigest md2 = MessageDigest.getInstance("MD2", "SAFENET");
byte[] digest = md2.digest("hello world".getBytes());
```

MD5

This message digest algorithm produces a 128-bit digest. The algorithm is described in RFC-1321, see http://www.ietf.org/rfc/rfc1321.txt. This algorithm is provided for compatibility only and is not recommended for other purposes. Instances of this algorithm are not clone-able.

To create a MD5 message digest for the message "hello world":

```
MessageDigest md5 = MessageDigest.getInstance("MD5", "SAFENET");
byte[] digest = md5.digest("hello world".getBytes());
```

SHA-1

The SHA-1 message digest algorithm produces a 160-bit digest. The algorithm is described in FIPS PUB 180-1, see http://www.itl.nist.gov/div897/pubs/fip180-1.htm. Instances of this algorithm are not cloneable.

To create a SHA-1 message digest for the message “hello world”:

```
MessageDigest sha1 = MessageDigest.getInstance(“SHA-1”, “SAFENET”);
byte[] digest = sha1.digest(“hello world”.getBytes());
```

SHA-224

The SHA-224 message digest algorithm produces a 224-bit digest. The algorithm is described in FIPS PUB 180-1, see http://www.itl.nist.gov/div897/pubs/fip180-1.htm. Instances of this algorithm are not cloneable.

To create a SHA-224 message digest for the message “hello world”:

```
MessageDigest sha256 = MessageDigest.getInstance(“SHA-224”,
“SAFENET”);
byte[] digest = sha224.digest(“hello world”.getBytes());
```

SHA-256

The SHA-256 message digest algorithm produces a 256-bit digest. The algorithm is described in FIPS PUB 180-1, see http://www.itl.nist.gov/div897/pubs/fip180-1.htm. Instances of this algorithm are not cloneable.

To create a SHA-256 message digest for the message “hello world”:

```
MessageDigest sha256 = MessageDigest.getInstance(“SHA-256”,
“SAFENET”);
byte[] digest = sha256.digest(“hello world”.getBytes());
```

SHA-384

The SHA-384 message digest algorithm produces a 384-bit digest. The algorithm is described in FIPS PUB 180-1, see http://www.itl.nist.gov/div897/pubs/fip180-1.htm. Instances of this algorithm are not cloneable.

To create a SHA-384 message digest for the message “hello world”:

```
MessageDigest sha384 = MessageDigest.getInstance(“SHA-384”,
“SAFENET”);
byte[] digest = sha384.digest(“hello world”.getBytes());
```

SHA-512

The SHA-512 message digest algorithm produces a 512-bit digest. The algorithm is described in FIPS PUB 180-1, see http://www.itl.nist.gov/div897/pubs/fip180-1.htm. Instances of this algorithm are not cloneable.

To create a SHA-512 message digest for the message “hello world”:

```
MessageDigest sha512 = MessageDigest.getInstance("SHA-512",
"SAFENET");
byte[] digest = sha512.digest("hello world".getBytes());
```

RIPEMD128

The RIPEMD128 message digest algorithm produces a 128-bit digest.Instances of this algorithm are not cloneable.

To create a RIPEMD128 message digest for the message "hello world":

```
MessageDigest rmd128 = MessageDigest.getInstance("RIPEMD128",
     "SAFENET");
byte[] digest = rmd128.digest("hello world".getBytes());
```

RIPEMD160

The RIPEMD160 message digest algorithm produces a 160-bit digest. Instances of this algorithm are not clone-able.

To create a RIPEMD160 message digest for the message "hello world":

```
MessageDigest rmd160 = MessageDigest.getInstance("RIPEMD160",
     "SAFENET");
byte[] digest = rmd160.digest("hello world".getBytes());
```

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 8
KEY GENERATION

ProtectToolkit J can generate random keys for each of the cipher algorithms it supports. These keys are Cryptoki session keys, that is, they are not stored permanently on the adapter. Session keys are not thread-safe and so may only be used by a single `Cipher` instance and a single `Signature` (or `MAC`) instance at any time. Thus it is allowable to use a DES key for encryption in a `Cipher` instance and a single `MAC` instance but not two `Cipher` instances. Keys fetched from the ProtectToolkit J KeyStore do not have this restriction (see section 8.1).

When generating a random key the size of the key will be as follows:

Key Name	Default Key Size	Valid Key Sizes
DES	56	56
DESede	196	128,196
AES	128	(128,196,256)
IDEA	128	128
CAST128	128	8-128
RC2	64	0-1024
RC4	64	8-2048
RSA	1024	512-4096
DSA	1024	512-4096
DH	1024	512-4096

Secret Keys

The secret key Ciphers will simply generate the appropriate number of random bytes for the key (there are no checks for *weak* keys).

The following example will generate a random double length DESede key. Generation of a key for a different algorithm is as simple as changing the algorithm name and choosing an appropriate key length.

```
KeyGenerator keyGen = KeyGenerator.getInstance("DESede", "SAFENET");
keyGen.init(128);
SecretKey key = keyGen.generateKey();
```

Public Keys

RSA Keys

The RSA key pair generator will generate a RSA key pair based on an algorithm determined by key size. If the size is some multiple of 256 bits greater then 1024 then the algorithm specified in ANSI X 9.31 will be used else the one specified in PKCS#1 is used. The key pair will be compatible with PKCS#1 RSA, ISO/IEC 9796 RSA and X.509 (raw) RSA standards. ANSI X 9.31 keys have a random 16 bit exponent while PKCS#1 public exponent is fixed to the Fermat-4 value (hex 0x1001).

The following example will generate a 2048 bit RSA key pair.

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA",

"SAFENET");
keyPairGen.initialise(2048);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

DSA Keys

The DSA key pair generator will generate a DSA key pair based on the algorithm specified in the Digital Signature Standard (FIPS PUB 186-1). DSA key generation requires a number of parameters, these parameters are generally fixed in a given application but they are also usually randomly generated for a particular application. At present ProtectToolkit J does not include any mechanism to generate these parameters, however the DSA key pair generator can accept these parameters (via a `java.security.spec.DSAParameterSpec`) or has configured defaults for 512 or 1024 bit keys (these defaults are listed in the JCE specification).

The following example will generate a 1024 bit DSA key pair using the default DSA parameters.

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA",

"SAFENET");
keyPairGen.initialise(1024);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

This example will use the provided DSA parameters rather than the built in defaults.

```
BigInteger p, q, g;  // These are the parameter values
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA",

"SAFENET");
DSAParameterSpec keyParamSpec = new DSAParamterSpec(p, q, g);
keyPairGen.initialise(keyParamSpec);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

Diffie-Hellman Keys

The DH KeyPairGenerator will generate Diffie-Hellman keys suitable for the Diffie-Hellman key agreement protocol. Diffie-Hellman key generation requires a number of parameters, these parameters are generally fixed in a given application but they are also usually randomly generated for a particular application. At present ProtectToolkit J does not include any mechanism to generate these parameters, however the DH key pair generator can accept these parameters (via a `java.security.spec.DHParameterSpec`) or has configured defaults for 512 or 1024 bit keys (these defaults are listed in the JCE specification).

The following example will generate a 1024 bit DH key pair using the default DH parameters.

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DH",

"SAFENET");
keyPairGen.initialise(1024);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

This example will use the provided DH parameters rather than the built in defaults.

```
BigInteger p, g;  // These are the parameter values
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DH",

"SAFENET");
DSAParameterSpec keyParamSpec = new DHParamterSpec(p, g);
keyPairGen.initialise(keyParamSpec);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

KeyAgreement Protocols

ProtectToolkit J also includes mechanisms which allow for the creation of keys based on other keys.

Diffie-Hellman KeyAgreement

The DH KeyAgreement algorithm can be used to perform a 2 phase key Diffie-Hellman key agreement.

Xor Key Derive

This algorithm may be used to derive a new key from an existing key and a known data pattern. The key value and the data pattern will be combined on the adapter using the XOR function. For example if the initial key has the value 0x12,0x34 and the data pattern has the value 0x89,0xAB then the resultant key will have the value 0x88,0x88.

The actual key values will be combined within the adapter to ensure their values are never compromised. In addition the newly created key will inherit the attributes of the two keys such that the derived key will be as protected as the two original keys. Further, this mechanism may not be used to change the key type of the base key. Therefore if the base key is a DES key the derived key must also be a DES key.

This mechanism can only be used on keys with the `CKA_DERIVE` attribute set to true. This will the case for keys generated with any of the ProtectToolkit J mechanisms (i.e. `KeyGenerator` classes) however if the key is generated with the Browser application be sure to check the 'Derive' checkbox.

Do not create an instance of this class directly, rather use the `KeyAgreement.getInstance()` factory method:

```
KeyAgreement ka = KeyAgreement.getInstance("XorBaseAndKey",
"SAFENET");
```

Once created the instance should be initialised using the base key. Then to combine with the data pattern call the `doPhase()` method with a SecretKeySpec instance created with the data pattern and true for the lastPhase parameter.

Finally to obtain the newly created instance call the `generateSecret()` method with the appropriate key name.

For example:

```
byte[] data = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};
ka.init(baseKey);
ka.doPhase(new SecretKeySpec(data), true);
Key newKey = ka.generateSecret("DES");
```

Note, the key material generated must be compatible with the key type requested in the `generateSecret()` method call. Specifically the length of the new key will be the minimum of the length's of the two components.

CHAPTER 9
KEY MANAGEMENT

Key Storage

The encryption adapter has the facility to store public, private and secret keys. These keys will be stored in the non-volatile storage on the card. As well as key storage it is also possible to store X.509 Certificates (which contain a public key). ProtectToolkit J provides access to this storage mechanism via the JCE `KeyStore` API. The JCE name for this `KeyStore` is `CRYPTOKI`.

The JCE `KeyStore` API allows storage of a `Key` and an associated alias. This alias is simply a unique string which may be used to access the key. To store a key in the key store, use the `setKeyEntry()`. To retrieve a key, use the `getKey()`. Keys may be removed from the KeyStore using the `deleteEntry()` method.

Currently only two types of keys may be stored in the ProtectToolkit J KeyStore, either ProtectToolkit J keys or `javax.crypto.spec.SecretKeySpec` keys. Other key types need to be converted to their ProtectToolkit J equivalents before storage.

Currently the `Certificate` support is based on Sun's Certificate implementation which is only available on the Sun Java2 JVM.

Per Key password protection is not supported so a `null` password may be supplied to the methods use to store and retrieve keys from the `KeyStore`. The password provided to the `load()` method will be used login to the token, and so to access private objects on the token it is necessary to provide the PIN. If a PIN is not supplied all objects will be stored as public objects, when a PIN is supplied only `PublicKey` and `Certificate` objects will be stored as public objects all others will be private. In either case the InputStream passed to the `store()` and `load()` methods will not change the contents of the key store.

Keys stored in the KeyStore are the only thread-safe ProtectToolkit J keys. A key instance obtained from the KeyStore.getKeyEntry() method will return a key that may be used in multiple `Cipher`, `Mac` and `Signature` instances.

The following example will create a new random DES key, and then store that key in the keystore. Note that even though we first create the key and then store it, it actual key value will not leave the hardware and therefore remains secure.

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES", "SAFENET");
Key key = keyGen.generateKey();
KeyStore keyStore = KeyStore.getInstance("CRYPTOKI", "SAFENET");
keyStore.load(null, null);
keyStore.setKeyEntry("des key", key, null, null);
```

The following example can be used to access the previously stored key.

```
KeyStore keyStore = KeyStore.getInstance("CRYPTOKI",
"SAFENET");
keyStore.load(null, null);
Key key = keyStore.getKey("des key", null);
```

Key Wrapping

The CRYPTOKI `KeyStore` also provides a key wrapping mechanism. Key wrapping is a technique where one key value is encrypted using another key. With ProtectToolkit J since the keys values are stored securely on the hardware we can use this technique to encrypt the key on the hardware and then extract the encrypted key.

For example, using this mechanism, a session key may be generated on the hardware and then exported from the hardware in an encrypted (i.e. wrapped) form. The key will generally be encrypted using a Public/Private key encryption cipher and can then be safely exported from the board. It is also possible to use secret keys for key wrapping, however in this case the same secret key must exist on both the source (performing the wrapping function) and the destination adapters.

The `WrappingKeyStore` API is an extension to the standard JCE that is used to provide access to key wrapping services. This class is identical to the standard `KeyStore` API except that it provides `wrapKey()` and `unwrapKey` methods. The wrapping key store can be instantiated using the following code:

```
import au.com. safenet.crypto.WrappingKeyStore;

...

WrappingKeyStore wks = WrappingKeyStore.getInstance("CRYPTOKI",
                                                    "SAFENET");
...
```

The `wrapKey()` method has the following signature:

```
public byte[] wrapKey(Key wrapKey, String transformation, Key key)
throws GeneralSecurityException
```

The `wrapKey` parameter specifies the `Key` used to encrypt the `key` parameter. The `transformation` parameter specifies the encryption transformation that is to be used to encrypt the key. With the `CRYPTOKI` KeyStore you can transform the following:

- `RSA/ECB/PKCS1Padding`
- `RSA/ECB/NoPadding`
- `DES/ECB/NoPadding`
- `DES/ECB/PKCS5Padding`
- `DESede/ECB/NoPadding`
- `DESede/ECB/PKCS5Padding`
- `IDEA/ECB/NoPadding`
- `IDEA/ECB/PKCS5Padding`
- `CAST128/ECB/NoPadding`
- `CAST128/ECB/PKCS5Padding`
- `RC2/ECB/NoPadding`
- `RC2/ECB/PKCS5Padding`
- `RC4`

A `GeneralSecurityException` will be thrown if the `transformation` parameter is invalid.

The value returned is a byte array containing the encrypted key. This value may be passed to the `unwrapKey()` method to extract the original key. The `unwrapKey()` method has the following signature:

```
public Key unwrapKey(Key unwrapKey, String transformation,
                     byte[] wrappedKey, String keyAlgorithm)
throws GeneralSecurityException
```

This method will "unwrap" or decrypt the encrypted key using the provided decryption key and transformation. The `Key` returned will be of the type specified by the keyAlgorithm parameter, however this parameter must match the actual key type that was originally wrapped.

The `unwrapKey` parameter should be the either the same secret key as was used to wrap the key or the private key corresponding to the public key used to wrap the key. The `transformation` parameter specifies the decryption transformation that is to be used to decrypt the key. This value should be the same as that used to wrap the key. The `wrappedKey` parameter should contain the encrypted key. The `keyAlgorithm` should specify the algorithm that the decrypted key is for.

A `GeneralSecurityException` will be thrown if the `transformation` parameter is invalid.

The following example will create a new random RC4 key, wrap that key with a RSA public key and then finally unwrap it with the associated RSA private key.

```
KeyGenerator keyGen = KeyGenerator.getInstance("RC4", "SAFENET");
Key rc4Key = keyGen.generateKey();

WrappingKeyStore wks = WrappingKeyStore.getInstance("CRYPTOKI");
wks.load(null, null);  // initialise the KeyStore
Key publicKey = wks.getKey("RSA_pub", null);
byte[] encKey = Wks.wrapKey(publicKey,"RSA/ECB/PKCS1Padding",rc4Key);

// give the encrypted key to the recipient, and unwrap it
Key privateKey = wks.getKey("RSA_priv", null);
Key recoveredKey = wks.unwrapKey(privateKey, "RSA/ECB/PKCS1Padding',
                                                              encKey);
```

Key Specifications

As well as supporting the relevant JCA/JCE defined `KeySpec` classes ProtectToolkit J includes a number of custom provider-independent key classes for use with its `KeyFactory` classes. These classes all live in the `au.com. safenet.crypto.spec` package:

`AsciiEncodedKeySpec`

Used to encode RSA, DSA or Diffie-Hellman public and private keys as ASCII strings. These strings contain the key's integer components as hexadecimal strings separated by a full stop. For example an RSA private key:

```
public_exponent.modulus.private_exponent.p.q
```

A public key will contain only the first two elements and a private key will contain all five. The RSA `KeyFactory` can convert from this `KeySpec` into the provider-based key.

For DSA keys the format is:

```
y.p.q.g (private keys) x.p.q.g (public keys)
```

For Diffie-Hellman keys the format is:

`y.p.g` (private keys) `x.p.g` (public keys)

`CASTKeySpec`

Used to encode keys for the CAST algorithm. This class takes a byte array which it will use directly as the CAST key. The array must be less than or equal to 16 bytes which is the maximum key size for a CAST key.

`IDEAKeySpec`

Used to encode keys for the IDEA algorithm. This class takes a byte array and uses the first 16 bytes of the array as the IDEA key.

`RC2KeySpec`

Used to encode keys for the RC2 algorithm. This class takes a byte array which it will use directly as the RC2 key. The array must be less than or equal to 128 bytes which is the maximum key size for a RC2 key.

`RC4KeySpec`

Used to encode keys for the RC4 algorithm. This class takes a byte array which it will use directly as the RC4 key. The array must be less than or equal to 256 bytes which is the maximum key size for a RC4 key.

`AESKeySpec`

Used to encode keys for the AES algorithm. This class takes a byte array which it will use directly as the AES key. The array must be either be 16, 24 or 32 bytes.

CHAPTER 10
RANDOM NUMBER GENERATION

The Safenet provider (named "`safenet`") implements a `java.security.SecureRandom` class for generating random data. This implementation is known as "CRYPTOKI". Besides using a hardware based entropy generator, one of the major benefits of this implementation is that it does not suffer from the slow initialization problem that the Sun provided (and most other software implementations) do.

This interface is only available under Java2.

This implementation allows access to the encryption adapter random source for both seeding and random number generation. The SafeNet ProtectServer PCIe HSM uses hardware based random number generation.

Serialization of an instance of this class will not save the state of the random number generator as it is contained within the hardware.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 11
BEST PRACTICE GUIDELINES

Introduction

The purpose of this section is to outline some of the "Best Practices" application developers can implement when developing their ProtectToolkit J based applications.

The following guidelines do not attempt to replace the vast body of literature regarding building secure systems or implementing cryptography for security. Rather it focuses on some of the specific aspects of the ProtectToolkit J product that are particularly relevant to building applications in a timely and reliable way.

ProtectToolkit J Provider

The ProtectToolkit J JCA/JCE Provider provides access to the many cryptographic features of the ProtectServer Blue and ProtectServer range of hardware.

As the provider is hardware based there are a number of differences between it and other software based implementations. Mostly these stem from the different methods used to protect the key store where hardware can effectively provide some level of physical protection.

Key Value Protection

Normally keys protected by the hardware will not allow their values to be revealed outside the adapter. Thus the `Key.getEncoded()` interface will generally return a null value.

Key Usage Protection

Each key has an associated set of usage flags that indicate which cryptographic operations may be performed with the key. For example, specific flags may be set to enable encryption or signature generation. Keys in the ProtectToolkit J provider will adhere to these rules.

General ProtectToolkit J Usage Guidelines

1. Create persistent keys with the Key Management Utility (KMU) and specify their key usage attributes appropriately.
    - secret and private keys should always be sensitive
    - each key should be usable for only one purpose
    - use the KMU for key backups with the exportable attribute
2. Persistent key instances from the ProtectToolkit J KeyStore implementation are shareable. This means a key lookup only needs to be performed once rather than every time a key is required.

3. Initialise the token correctly. Different applications should use different tokens.

4. Install the ProtectToolkit J provider as the highest priority or use Security.insertProvider(`SAFENET`Provider(), early on in your application. This will ensure that the SAFENET hardware SecureRandom will become the system default providing improved quality random data and avoiding the startup performance penalty of the sun implementation.

5. Fully specify Cipher transformations. E.g. use "`DES/ECB/NoPadding`" instead of "`DES`".

APPENDIX A
REFERENCES

FIPS PUB 42-2

Data Encryption Standard. See http://www.itl.nist.gov/div897/pubs/fip42-2.htm.

FIPS PUB 81

DES Modes of Operation. See http://www.itl.nist.gov/div897/pubs/fip81.htm.

FIPS PUB 113

Computer Data Authentication. See http://www.itl.nist.gov/div897/pubs/fip113.htm.

FIPS PUB 180-1

Secure Hash Standard. See http://www.itl.nist.gov/div897/pubs/fip180-1.htm.

FIPS PUB 186-1

Digital Signature Standard (DSS). See http://www.itl.nist.gov/div897/pubs/fip186-1.htm.

PKCS#1

RSA Encryption Standard. See http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-1.html.

PKCS#5

Block cipher padding. See http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-5.html.

END OF DOCUMENT