

nShield Security World

nCipherKM JCA JCE CSP v13.9.6 API Guide

03 September 2026

© Entrust Corporation. All rights reserved.

Table of Contents

1. Introduction . . . 1
2. Installing the nCipherKM JCA/JCE CSP . . . 2
    2.1. Prerequisites . . . 2
    2.2. Installing the nCipherKM JCA/JCE CSP . . . 3
    2.3. Testing the nCipherKM JCA/JCE CSP installation . . . 5
    2.4. Named Modules in Java 17, 21, and 25 . . . 7
3. System Properties . . . 9
    3.1. JCECSP_DEBUG property values . . . 10
4. Compatibility . . . 12
5. Architecture . . . 13
6. Available Functions . . . 15
    6.1. Available functions . . . 15
        6.1.1. ML-DSA . . . 20
        6.1.2. SHAKE128/SHAKE256 . . . 20
        6.1.3. nCNistKDF . . . 21
7. The KeyStore API . . . 22
8. Initialization . . . 23
9. Loading and Storing Keys . . . 24
10. keytool . . . 25
11. Using Keys . . . 27

1. Introduction

The nCipherKM JCA/JCE CSP (Cryptographic Service Provider) allows Java applications and services to access the secure cryptographic operations and key management provided by Entrust nShield hardware. This provider is used with the standard JCE (Java Cryptographic Extension) programming interface.

Before reading this document and using the nCipher JCA/JCE CSP toolkit supplied by Entrust, familiarize yourself with the information contained in API introductory guide.

2. Installing the nCipherKM JCA/JCE CSP

2.1. Prerequisites

To use the nCipherKM JCA/JCE CSP, you must install the nShield Java package which includes the nShield Java jars.

For more information about the bundles and components supplied on your Security World Software installation media, see Software packages on the installation media.

The following versions of Java have been tested to work with, and are supported by, your nShield Security World Software:

- Java 8 (or Java 1.8x)
- Java 17
- Java 21
- Java 25

We recommend that you ensure Java is installed before you install the Security World Software. The Java executable must be on your system path.

If you can do so, please use the latest Java version currently supported by Entrust that is compatible with your requirements. Java versions before those shown are no longer supported. If you are maintaining older Java versions for legacy reasons, and need compatibility with current nShield software, please contact Entrust nShield Support: https://trustedcare.entrust.com/.

To install Java you may need installation packages specific to your operating system, which may depend on other pre-installed packages to be able to work.

Suggested links from which you may download Java software as appropriate for your operating system:

- http://www.oracle.com/technetwork/java/index.html
- http://www.oracle.com/technetwork/java/all-142825.html



Detailed documentation for the JCE interface can be found on the Oracle Technology web page here.



Softcards are not supported for use with the nCipherKM JCA/JCE CSP in Security Worlds that are compliant with FIPS 140 Level 3.

2.2. Installing the nCipherKM JCA/JCE CSP

To install the nCipherKM JCA/JCE CSP:

1. For Java 8 only (Java 17 and later supports UNIX domain sockets). In the hardserver configuration file, ensure that:
   - `priv_port` (the port on which the hardserver listens for local privileged TCP connections) is set to `9001`.
   - `nonpriv_port` (the port on which the hardserver listens for local nonprivileged TCP connections) is set to `9000`.

   If you need to change either or both of these port settings, restart the hardserver before continuing the nCipherKM JCA/JCE CSP installation process. For more information, see Client software and module configuration: network-attached HSMs or Client software and module configuration: PCIe and USB HSMs.

2. For Java 8 only. Copy the `nCipherKM.jar` file to the extensions folder of your local Java Virtual Machine installation from the following directory:
   - `%NFAST_HOME%\java\classes` (Windows)
   - `/opt/nfast/java/classes` (Linux)

   The location of the extensions folder depends on the type of your local Java Virtual Machine (JVM) installation:

JVM type	Extensions folder (Windows)	Extensions folder (Linux)
Java Developer Kit (JDK)	`%JAVA_HOME%\jre\lib\ext`	`$JAVA_HOME/jre/lib/ext`
Java Runtime Environment (JRE)	`%JAVA_HOME%\lib\ext`	`$JAVA_HOME_/lib/ext`

   In these paths, `%JAVA_HOME%` (Windows) and `$JAVA_HOME` (Linux) are the home directory of the Java installation (commonly specified in the `JAVA_HOME` environment variable).

   If you are using Java 17, 21, or 25, you do not need to copy the jar file.

3. Add `%JAVA_HOME%\bin` (Windows) or `$JAVA_HOME/bin` (Linux) to your `PATH` system variable.
4. For Java 8 only. Install the unlimited strength JCE jurisdiction policy files that are appropriate to your version of Java. JDK 9 and later ship with, and use by default, the unlimited policy files.

   The Java Virtual Machine imposes limits on the cryptographic strength that may be used by default with JCE providers. Replace the default policy configuration files with

unlimited strength policy files.

To install the unlimited strength JCE jurisdiction policy files:

a. If necessary, download the archive containing the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from your Java Virtual Machine vendor's Web site. Be sure to download a file appropriate for your version of Java.



The Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files are covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. We recommend that you take legal advice before downloading these files from your Java Vir tual Machine vendor.

b. Extract the files `local_policy.jar` and `US_export_policy.jar` from Java Virtual Machine vendor's Java Cryptography Extension (JCE) Unlimited Strength Jurisdic tion Policy File archive.

c. Copy the extracted files `local_policy.jar` and `US_export_policy.jar` into the security directory for your local Java Virtual Machine (JVM) installation:

JVM type	Extensions folder (Windows)	Extensions folder (Linux)
Java Developer Kit (JDK)	`%JAVA_HOME%\jre\lib\security`	`$JAVA_HOME/jre/lib/security`
Java Runtime Envi ronment (JRE)	`%JAVA_HOME%\lib\security`	`$JAVA_HOME_/lib/security`

In these paths, `%JAVA_HOME%` (Windows) and `$JAVA_HOME` (Linux) are the home direc tory of the Java installation (commonly specified in the `JAVA_HOME` environment variable).



Copying the files `local_policy.jar` and `US_export_policy.jar` into the appropriate folder must overwrite any existing files with the same names.

5. Add the nCipherKM provider to the `java.security` file located in the security directory for your local Java Virtual Machine (JVM) installation: `security.provider.<n>=com.nci- pher.provider.km.nCipherKM`, where `<n>` is the position in the list of providers, for exam ple:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
```

```
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=com.ncipher.provider.km.nCipherKM
```

For Java 17, 21, and 25, you do not need to specify the fully qualified class name for the provider. Instead you can just use the provider name: `security.provider.<n>=nCipherKM`.

The JVM uses this file to select the provider from which to request a mechanism instance. If your JCE application does not request the nCipherKM provider by name, or if it fails to load keys, you might need to move the nCipherKM provider to the top of the list: `security.provider.1=com.ncipher.provider.km.nCipherKM`. Do not change the relative order of the other providers in the list.



Ensure you do not list multiple providers with the same number (for example, ensure your list of providers does not include two instances of `security.provider.1`, both `com.ncipher.provider.km.nCipherKM` and another provider). If you add the nCipherKM provider as `security.provider.1`, ensure that the subsequent providers are re-numbered correctly.

6. Save your updates to the file `java.security`.

When you have installed the nCipherKM JCA/JCE CSP, you must have created a Security World before you can test or use it. For more information about creating a Security World, see Create a new Security World.



If you have a Java Enterprise Edition Application Server running, you must restart it before the installed nCipherKM provider is loaded into the Application Server virtual machine and ready for use.

2.3. Testing the nCipherKM JCA/JCE CSP installation

After installation, you can test that the nCipherKM JCA/JCE CSP is functioning correctly by running the command.

For Java 8:

```
java com.ncipher.provider.InstallationTest
```

For Java 17, 21, and 25 (Windows):

```
java --module-path %NFAST_HOME%\java\classes com.ncipher.provider.InstallationTest
```

For Java 17, 21, and 25 (Linux):

```
java --module-path /opt/nfast/java/classes com.ncipher.provider.InstallationTest
```



For these commands to work, you must have added `%JAVA_HOME%` (Windows) or `$JAVA_HOME` (Linux) to your `PATH` system variable.

If the nCipherKM JCA/JCE CSP is functioning correctly, output from this command has the following form:

```
Installed providers:
1: nCipherKM
2: SUN
3: SunRsaSign
4: SunJSSE
5: SunJCE
6: SunJGSS
7: SunSASL
Unlimited strength jurisdiction files are installed.
The nCipher provider is correctly installed.
nCipher JCE services:
Alg.Alias.Cipher.1.2.840.113549.1.1.1
Alg.Alias.Cipher.1.2.840.113549.3.4
Alg.Alias.Cipher.AES
Alg.Alias.Cipher.DES3
```

If the `nCipherKM` provider is installed but is not registered at the top of the providers list in the `java.security` file, the `InstallationTest` command produces output that includes the message:

```
The nCipher provider is installed, but is not registered at the top of the providers list in the java.security
file.
See the user guide for more information about the recommended system configuration.
```

In such a case, edit the `java.security` file (located in the security directory for your local JVM installation) so that the nCipherKM provider is registered in the first position in that file's list of providers. For more information about the `java.security` file, see Installing the nCipherKM JCA/JCE CSP.

If the nCipherKM provider is not installed at all, or you have not created a Security World, or if you have not configured ports correctly in the hardserver configuration file, the `InstallationTest` command produces output that includes the message:

```
The nCipher provider is not correctly installed.
```

In such case:

- Check that you have configured ports correctly, as described in Installing the nCipherKM JCA/JCE CSP. For more information about hardserver configuration file settings, see nShield HSM configuration files.
- Check that you have created a Security World. If you have not created a Security World, create a Security World. For more information, see Create a new Security World
- If you have already created a Security World, repeat the nCipherKM JCA/JCE CSP installation process as described in Installing the nCipherKM JCA/JCE CSP.

After making any changes to the nCipherKM JCA/JCE CSP installation, run the `InstallationTest` command again and check the output.

Whether or not the nCipherKM provider is correctly installed, if the unlimited strength jurisdiction files are not installed or (not correctly installed), the `InstallationTest` command produces output that includes the message:

```
Unlimited strength jurisdiction files are NOT installed.
```



The `InstallationTest` command can only detect this situation if you are using JRE/JDK version 1.6 or later.

This message means that, because the Java Virtual Machine imposes limits on the cryptographic strength that you can use by default with JCE providers, you must replace the default policy configuration files with the unlimited strength policy files. For information about how to install the unlimited strength jurisdiction files, see Installing the nCipherKM JCA/JCE CSP.

2.4. Named Modules in Java 17, 21, and 25

The nCipherKM Provider has been implemented as a named module. For Java 17, 21, and 25, if you add the provider to the `java.security` file, you can run your application with `nCipherKM.jar` on the module path. The Java ServiceLoader class then automatically finds it. For example:

Linux

```
java --module-path /opt/nfast/java/classes com.ncipher.provider.InstallationTest
```

Windows

```
java --module-path %NFAST_HOME%\java\classes com.ncipher.provider.InstallationTest
```

Alternatively, you can specify the location of the nCipherKM jar on the classpath:

Linux

```
java --class-path /opt/nfast/java/classes/nCipherKM.jar com.ncipher.provider.InstallationTest
```

Windows

```
java --class-path %NFAST_HOME%\java\classes\nCipherKM.jar com.ncipher.provider.InstallationTest
```

3. System Properties

You can use system properties to control the provider. You set system properties when starting the Java Virtual Machine using a command such as:

```
java -D<property>=<value> <MyJavaApplication>
```

In this example command, `<property>` represents any system property, `<value>` represents the value set for that property, and `<MyJavaApplication>` is the name of the Java application you are starting. You can set multiple system properties in a single command, for example:

```
java -Dprotect=module -DignorePassphrase=true MyJavaApplication
```

The available system properties and their functions as controlled by setting different values for a property are described in the following table:

Property	Function for different values
`JCECSP_DEBUG`	This property is a bit mask for which different values specify different debugging functions; the default value is `0`. For details about the effects of setting different values for this property, see JCECSP_DEBUG property values.
`JCECSP_DEBUGFILE`	This property specifies a path to the file to which logging output is to be written. Set this property if the `JCECSP_DEBUG` property is set to a value other than the default of `0`. For details about the effects of setting different values for this property, see JCECSP_DEBUG property values. In a production environment, we recommend that you disable debug logging to prevent sensitive information being made available to an attacker.
`CKNFAST_JCE_COMPATIBILITY`	This property is included to allow the saving of objects when using Java PKCS#11 providers.
`protect`	This property specifies the type of protection to be used for key generation and nCipherKM KeyStore instances. You can set the value of this property to one of `module`, `softcard:`IDENT or `cardset`. OCS protection (`cardset`) uses the card from the first slot of the first usable hardware security module. To find the logical token hash *IDENT* of a softcard, run the command `nfkminfo --softcard-list`.
`module`	This property lets you override the default module and select a specific module to use for module and OCS protection. Set the value of this property as the ESN of the module you want to use.
`slot`	This property lets you override the default slot for OCS-protection and select a specific slot to use. Set this the value of this property as the number of the slot you want to use.

Property	Function for different values
ignorePassphrase	If the value of this property is set to true, the nCipherKM provider ignores the passphrase provided in its KeyStore implementation. This feature is included to allow the Oracle or IBM keytool utilities to be used with module-protected keys. The keytool utilities require a passphrase be provided; setting this property allows a dummy passphrase to be used.
seeintegname	Setting the value of this property to the name of an SEE integrity key causes the provider to generate SEE application keys. These keys may only be used by an SEE application signed with the named key.
com.ncipher.provider.announce-mode	The default value for this property is auto, which uses firmware auto-detection to disable algorithms in the provider that cannot be supported across all installed modules. Setting the value of this property to on forces the provider to advertise all mechanisms at start-up. Setting the value of this property to off forces the provider to advertise no mechanisms at start-up.
com.ncipher.provider.enable	For the value of this property, you supply a comma-separated list of mechanism names that are to be forced on, regardless of the announce mode selected.
com.ncipher.provider.disable	For the value of this property, you supply a comma-separated list of mechanism names that are to be forced off, regardless of the announce mode selected. Any mechanism supplied in the value for the com.ncipher.provider.disable property overrides the same mechanism if it is supplied in the value for the com.ncipher.provider.enable property.

3.1. JCECSP_DEBUG property values

The JCECSP_DEBUG system property is a bit mask for which you can set different values to control the debugging functions. The following table describes the effects of different values that you can set for this property:

JCECSP_DEBUG value	Function
0	If this property has no bits set, no debugging information is reported. This is the default setting.
1	If this property has the bit 1 set, minimal debugging information (for example, version information and critical errors) is reported.
2	If this property has the bit 2 set, comprehensive debugging information is reported.
4	If this property has the bit 3 set, debugging information relating to creation and destruction of memory and module resources is reported.
8	If this property has the bit 4 set, debugFunc and debugFuncEnd generate debugging information for functions that call them.

JCECSP_DEBUG value	Function
16	If this property has the bit 5 set, debugFunc and debugFuncEnd display the values for all the arguments that are passed in to them.
32	If this property has the bit 6 set, context information is reported with each debugging message (for example, the ThreadID and the current time.
64	If this property has the bit 7 set, the time elapsed during each logged function is calculated, and information on the number of times a function is called and by which function it was called is reported.
128	If this property has the bit 8 set, debugging information for NFJAVA is reported in the debugging file.
256	If this property has the bit 9 set, the call stack is printed for every debug message.

To set multiple logging functions, add up the JCECSP_DEBUG values for the debugging functions you want to set, and specify the total as the value for JCECSP_DEBUG. For example, if you want to set the debugging to use both function tracing (bit 4) and function tracing with parameters (bit 5), add the JCECSP_DEBUG values shown in the table for these debugging functions (8 + 16 = 24) and specify this total (24) as the value to use for JCECSP_DEBUG.

4. Compatibility

The nCipherKM JCA/JCE CSP supports both module-protected keys and OCS-protected keys. The CSP currently supports 1/*N* OCSs and a single protection type for each nCipherKM JCE KeyStore.

You can use the nCipherKM JCA/JCE CSP with Security Worlds that comply with FIPS 140 at either Level 2 or Level 3.



In a Security World that complies with FIPS 140 Level 3, it is not possible to import keys generated by other JCE providers.

The nCipherKM JCA/JCE CSP supports load-sharing for keys that are stored in the nCipherKM KeyStore. This feature allows a server to spread the load of cryptographic operations across multiple connected modules, providing greater scalability.



We recommend that you use load-sharing unless you have existing code that is designed to run with multiple modules. To share keys with load-sharing, you must create a 1/*N* OCS with at least as many cards as you have modules. All the cards in the OCS must have the same passphrase.



The nCipherKM JCA/JCE CSP does not support HSM Pool mode. If you want to use HSM Pool mode with a Java application that only uses module protected keys, one option may be to use the Sun PKCS #11 provider to access the nShield PKCS #11 library instead of using nCipherKM JCA/JCE CSP.

Keys generated or imported by the nCipherKM JCA/JCE CSP are not recorded into the Security World until:

1. The key is added to an nCipherKM KeyStore (by using a call to `setKeyEntry()` or `setCertificateEntry()`).
2. That nCipherKM KeyStore is then stored (by using a call to `store()`).

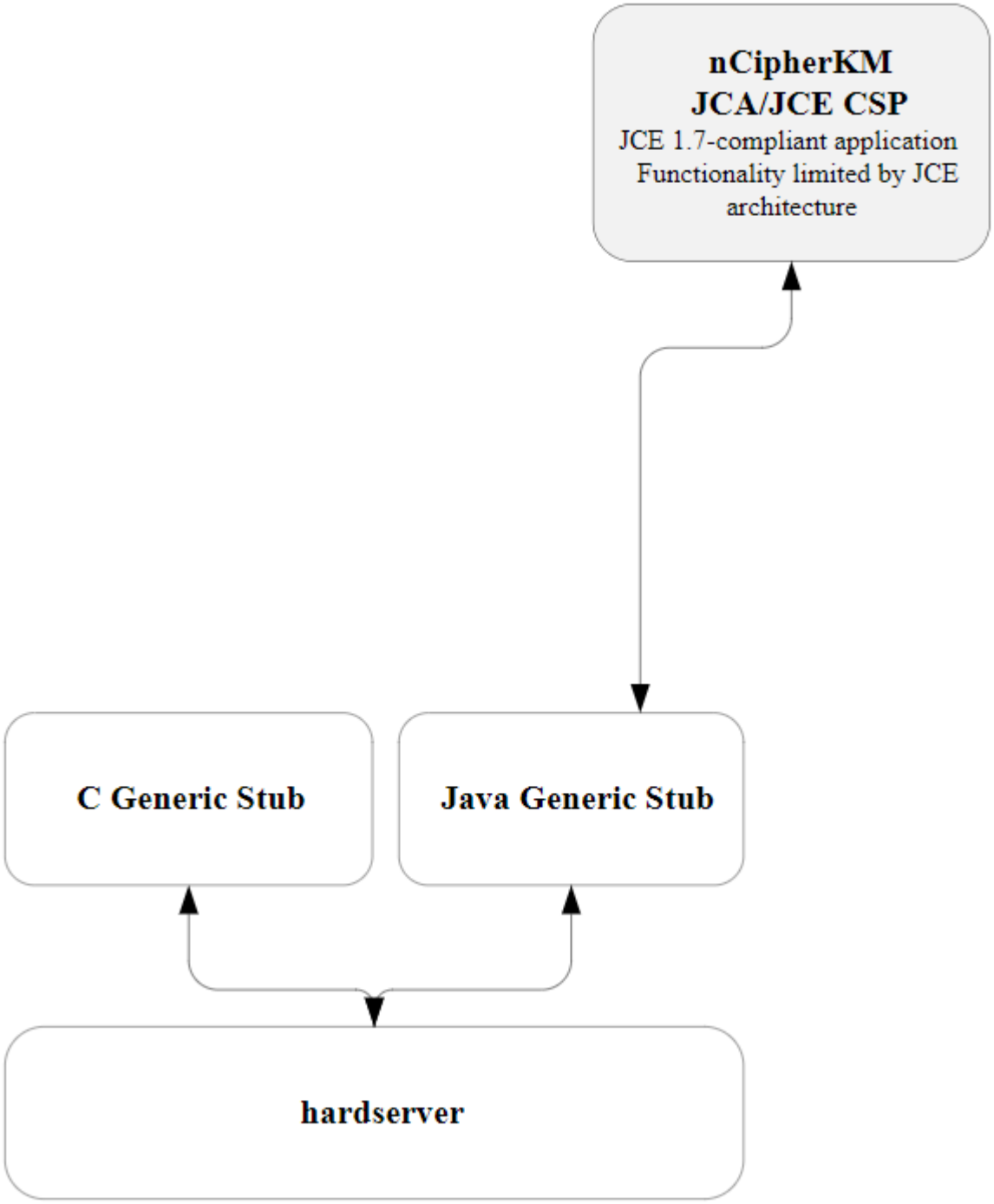
The passphrase used with the KeyStore must be the passphrase of the card from the OCS that protects the keys in the KeyStore.

5. Architecture

The nCipherKM JCA/JCE CSP implements its functionality using two underlying nShield APIs:

- the KM Java library (`kmjava`)
- the Java Generic Stub (`nfjava`).

These libraries relay commands generated by the JCE provider to the underlying hardserver and modules.


nCipherKM
JCA/JCE CSP
JCE 1.7-compliant application
Functionality limited by JCE
architecture
C Generic Stub
Java Generic Stub
hardserver

6. Available Functions

6.1. Available functions

The module firmware automatically detects which algorithms it can support. These algorithms are advertised when the provider first starts up. The provider conservatively advertises only those mechanisms that are supported by all installed modules in the system.



Certain algorithms are not supported by older versions of firmware. We recommend that you ensure that your module is upgraded to the most recent version of firmware appropriate for your environment.

The following table indicates the cipher modes available for each cipher.

Cipher	CBC	CFB	CTR	ECB	OFB	GCM
AESWrap				X		
ArcFour						
CAST256	X	X	X	X	X	
DES2	X	X	X	X	X	
DES	X	X	X	X	X	
DESede	X	X	X	X	X	
DESedeWrap	X					
ECIES[1]						
Rijndael	X	X	X	X	X	X
RSA				X		

In the table above, annotations with the following numbers indicate:

[1] These ciphers support key wrap and unwrap only.

The following table indicates the padding types available for each cipher.

Cipher	ANSI X9.23	ISO 10126	ISO 7816	None	OAEP	PKCS #1	PKCS #5	Zero byte
AESWrap				X				
ArcFour								
CAST256	X	X	X	X			X	X

Cipher	ANSI X9.23	ISO 10126	ISO 7816	None	OAEP	PKCS #1	PKCS #5	Zero byte
DES2	X	X	X	X			X	X
DES	X	X	X	X			X	X
DESede	X	X	X	X			X	X
DESedeWrap				X				
ECIES[1]								
Rijndael	X	X	X	X			X	X
RSA				X	X			

In the table above, annotations with the following numbers indicate:

[1] These ciphers support key wrap and unwrap only.

Key length is in bits for generation or signing:

Algorithm	Key length	KeyGen-erator	KeyPairGen erator	Signa-ture	Cip her	KeyA-gree-ment	KeyFac tory	MA C	Mes-sageDi-gest	Secure Random	KD F
AESWrap					Y						
Arcfour	8, 16 to 2048	Y[1]			Y[1]						
CAST256	128, 192, 256	Y[1]			Y[1]						
DES	64	Y[1]			Y[1]						
DESede	192	Y			Y						
DES2	128	Y			Y						
DESedeWrap					Y						
DH			Y			Y	Y				
DSA	1024		Y				Y				
ECDH			Y			Y	Y				
ECD-HwithSHA1K DF						Y					

Algorithm	Key length	KeyGen-erator	KeyPairGen erator	Signa-ture	Cip her	KeyA-gree-ment	KeyFac tory	MA C	Mes-sageDi-gest	Secure Random	KD F
ECD-HwithSHA224KDF						Y					
ECD-HwithSHA256KDF						Y					
ECD-HwithSHA384KDF						Y					
ECD-HwithSHA512KDF						Y					
ECDSA			Y				Y				
EdDSA	256		Y				Y				
Ed25519	256		Y	Y							
Ed25519ph				Y							
Ed448	456		Y	Y							
Ed448ph				Y							
HmacMD5		Y[1]						Y[1]			
HmacRIPEMD160	8, 16 to 2048	Y[1]						Y[1]			
HmacSHA1	8, 16 to 2048	Y						Y			
Hmac-SHA224	8, 16 to 2048	Y						Y			
Hmac-SHA256	8, 16 to 2048	Y						Y			
Hmac-SHA384	8, 16 to 2048	Y						Y			

Algorithm	Key length	KeyGen-erator	KeyPairGen erator	Signa-ture	Cip her	KeyA-gree-ment	KeyFac tory	MA C	Mes-sageDi-gest	Secure Random	KD F
HmacSHA512	8, 16 to 2048	Y						Y			
HmacTiger	8, 16 to 2048	Y[1]						Y[1]			
MD5									Y[1]		
MD5and-SHA1withRS A				Y							
MD5withRSA				Y							
ML-DSA			Y	Y			Y				
nCi-pher.sworld											
Rijndael		Y			Y						
RawRSA				Y							
RIPEMD160									Y[1]		
RIPEMD160wi thRSA				Y[1]							
RIPEMD160wi thRSAand-MGF1	322+			Y[1]							
RNG										Y	
RSA	512+		Y		Y		Y				
SHA1									Y		
SHA1with-DSA				Y							
SHA1with-ECDSA				Y							
SHA1withRS A				Y							
SHA1withRS AandMGF1	322+			Y							

Algorithm	Key length	KeyGenerator	KeyPairGenerator	Signature	Cipher	KeyAgreement	KeyFactory	MAC	MessageDigest	SecureRandom	KDF
SHA224									Y		
SHA224withDSA				Y							
SHA224withECDSA				Y							
SHA224withRSA				Y							
SHA224withRSAandMGF1	450+			Y							
SHA256									Y		
SHA256withDSA				Y							
SHA256withECDSA				Y							
SHA256withRSA				Y							
SHA256withRSAandMGF1	514+			Y							
SHA384									Y		
SHA384withDSA				Y							
SHA384withECDSA				Y							
SHA384withRSA				Y							
SHA384withRSAandMGF1	770+			Y							
SHA512									Y		
SHA512withDSA				Y							
SHA512withECDSA				Y							

Algorithm	Key length	KeyGen-erator	KeyPairGen erator	Signa-ture	Cip her	KeyA-gree-ment	KeyFac tory	MA C	Mes-sageDi-gest	Secure Random	KD F
SHA512with ML-DSA				Y							
SHA512withR SA				Y							
SHA512withR SAandMGF1	1026+			Y							
SHAKE128									Y		
SHAKE256									Y		
Tiger	8, 16 to 256	Y			Y				Y[1]		
nCNistKDF											Y

In the table above, annotations with the following numbers indicate:

[1] These algorithms are not supported in FIPS 140 Level 3 Security Worlds.

6.1.1. ML-DSA

Use of ML-DSA requires a firmware version of v13.8 or greater and the `PostQuantum` feature to be enabled.

`SHA512withML-DSA` is a pre-hash signature variant. The provider performs the SHA-512 hashing as part of the JCE Signature operation.

6.1.2. SHAKE128/SHAKE256

Use of `SHAKE128` and `SHAKE256` requires a firmware version of v13.8 or greater.

The `nCipherKM` provider supports the following fixed-length SHAKE message digest algorithms through the JCE `MessageDigest` API:

- `SHAKE128` - Produces a 32-byte digest.
- `SHAKE256` - Produces a 64-byte digest.



Although SHAKE algorithms are extendable-output functions, these JCE `MessageDigest` algorithms use fixed output lengths.

6.1.3. nCNistKDF

This is an interface to the generic KDF supported by nCore, see Generic KDF Support. KDF support requires Java 25 or later. nCNistKDF supports all SHA2 PRFs, all fields, and randomness extraction. Key derivation and random data generation are also both supported It does not support kx or the AES-CMAC PRF.

For more details, see the `JCEKDFExample.java` and the Javadocs installed in your `nfast` directory. The Java examples page in the *nCore 13.9.6 Developer Tutorial* provides more information about these examples and how to use them.



HMAC key generation requires special steps to set the permissions. See the `JCEKDFExample.java` example for more information.

7. The KeyStore API

You can load and store nShield module-protected keys by using the standard KeyStore API. This interface allows access to a KeyStore data file by means of a passphrase and an `InputStream` or `OutputStream`.

nShield KeyStore data files contain only the name-space identifier of the keys stored in them; the actual keys are stored in the Security World regardless of the stream used. The name-space identifier is the hash of the root key of the individual KeyStore. The `ident` of the KeyStore keys in the Security World begins with this hash and is followed by key-specific characters. This naming hierarchy allows you to identify the relevant key in Security World tools and remove keys from a KeyStore.



To use an existing KeyStore on another machine in the same Security World, copy both its KeyStore data file and the Security World's Key Management Data directory to the other machine.

8. Initialization

You create a new KeyStore by passing a null `InputStream` to the KeyStore load method. When you create a new KeyStore, the nCipherKM provider generates a KeyStore key that is used to sign trusted public certificate entries. The relevant signature is verified when public certificates in the KeyStore are used; this functionality prevents an attacker inserting new certificates into a KeyStore without the protection token that is needed to use the Key-Store key.

By default, the KeyStore protection key is OCS-protected. Ensure that the passphrase argu ment used with the KeyStore interface matches the passphrase of that OCS. When the Key Store method is called, you must present a card with a matching passphrase from the required OCS. You can use the `protect` system property to change the protection type used for the KeyStore key; for more information about the `protect` property, see System Properties.

An existing KeyStore file is not overwritten if the KeyStore store method is called on an `Out putStream` directed at the same file path. Instead, the KeyStore at the existing path is used to store the keys in the new KeyStore. This operation fails if the passphrases for the two KeyStores do not match.

9. Loading and Storing Keys

We recommend that separate KeyStores are used for separate purposes; for example, you can use one KeyStore to hold private keys and a different KeyStore for Certifying Authorities. With this approach, you need separate OCSs to operate separate KeyStores. However, you can also use different OCSs to protect keys within the same KeyStore.

You require a certificate chain to store private keys. The Virtual Machine JCE implementation enforces this requirement, not the nCipherKM provider.

10. keytool

You can use either the Oracle `keytool` utility or the IBM `keytool` utility to read and edit an nShield KeyStore. These utilities are shipped with the Oracle and IBM JVMs. You must specify the correct `nCipher.sworld` KeyStore type when you run the `keytool` utility, and you must specify the correct package name for the Oracle or IBM `keytool` utility.

To generate a new key in an OCS-protected KeyStore with the Oracle or IBM `keytool` utility, run the appropriate command:

- Sun Microsystems `keytool` utility:

  For Java 17, 21, and 25, use the following command:

  ```
  java --module-path /opt/nfast/java/classes sun.security.tools.keytool.Main -genkey -storetype
  nCipher.sworld -keyalg RSA -sigalg SHA1withRSA -storepass <KeyStore_passphrase> -keystore <KeyStore_path>
  ```

  For Java 8, use the following command:

  ```
  java sun.security.tools.keytool.Main -genkey -storetype nCipher.sworld -keyalg RSA -sigalg SHA1withRSA
  -storepass <KeyStore_passphrase> -keystore <KeyStore_path>
  ```

- IBM keytool utility:

  ```
  java com.ibm.crypto.tools.KeyTool -genkey -storetype nCipher.sworld -keyalg RSA -
  sigalg SHA1withRSA -storepass <KeyStore_passphrase> -keystore <KeyStore_path>
  ```

In these example commands, `<KeyStore_passphrase>` is the passphrase for the OCS that protects the KeyStore and `<KeyStore_path>` is the path to that KeyStore.

To generate a new key in a module-protected KeyStore with the Oracle or IBM `keytool` utility, run the appropriate command:

- Sun Microsystems `keytool` utility:

  For Java 17, 21, and 25, use the following command:

  ```
  java --module-path /opt/nfast/java/classes -Dprotect=module -DignorePassphrase=true
  sun.security.tools.keytool.Main -genkey -storetype nCipher.sworld -keyalg RSA -sigalg SHA1withRSA -keystore
  <KeyStore_path>
  ```

  For Java 8, use the following command:

  ```
  java -Dprotect=module -DignorePassphrase=true sun.security.tools.KeyTool -genkey -storetype nCipher.sworld
  -keyalg RSA -sigalg SHA1withRSA -keystore <KeyStore_path>
  ```

- IBM `keytool` utility:

```
java -Dprotect=module -DignorePassphrase=true com.ibm.crypto.tools.KeyTool -genkey -storetype
nCipher.sworld -keyalg RSA -sigalg SHA1withRSA -keystore <KeyStore_path>
```

In these example commands, `<KeyStore_path>` is the path to the KeyStore.

By default, the `keytool` utilities use the `MD5withRSA` signature algorithm to sign certificates used with a KeyStore. This signature mechanism is unavailable on modules with firmware version 2.33.60 or later.

11. Using Keys

Only the nCipherKM provider can use keys stored in an nShield KeyStore because the under lying key material is held separately in the Security World.

You can always store nShield keys in an nShield KeyStore. You can also store keys generated by a third-party provider into an nShield KeyStore if both of the following conditions apply:

- the key type is known to the nCipherKM provider
- the Security World is not compliant with FIPS 140 Level 3.

When you generate an nShield key (or create it from imported key material), that key is associated with an ACL (Access Control List). This ACL prevents the key from being used for operations for which it is unsuited and enforces requirements that certain tokens be presented; for example, the ACL can specify that signing key cannot be used for encryption.