Oracle® Jipher

User's Guide

Release 20
G59103-01
September 2026

Oracle Jipher User's Guide, Release 20

G59103-01

Contents

Preface

This guide shows you how to install, deploy, and configure Oracle Jipher, a Java cryptography service provider that provides FIPS 140-3 validated cryptography. This guide also provides technical information about available cryptographic engine class algorithms and default parameter values.

Audience

Jipher is intended for administrators who need to deploy their organization's Java applications in FIPS 140-3 regulated environments.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

What is Oracle Jipher?

Oracle Jipher is a Java Cryptographic Service Provider (CSP) that dynamically loads a Federal Information Processing Standards (FIPS) 140-3 validated OpenSSL cryptographic module at runtime and uses it as the underlying cryptographic implementation. It enables the deployment of Java applications in FIPS-regulated environments. Jipher makes its cryptographic services available to Java developers using the Java Cryptography Architecture (JCA).

Jipher 20 supports the following runtimes:

- Oracle JDK 27
- Oracle JDK 25
- Oracle JDK 21
- Oracle JDK 17

Jipher 20 supports the following platforms:

- Oracle Linux 10, 9, and 8 on x86-64 and aarch64
- Red Hat Enterprise Linux (RHEL) 10, 9, and 8 on x86-64 and aarch64
- macOS 26 on M series Apple silicon
- Windows Server 2025

For TLS, Jipher supports the JDK JSSE provider, in particular, the SunJSSE provider for TLSv1.2 and TLSv1.3.

Jipher, also known as JipherJCE, provides FIPS 140-3 cryptographic services by using the Foreign Function and Memory (FFM) API starting with JDK 25 and the Java Native Interface (JNI) in JDK 17 and 21 to make calls into the OpenSSL FIPS module. It maps Java cryptography API calls to OpenSSL API calls, which call into the OpenSSL FIPS module. See FIPS-140 in the OpenSSL documentation for more information.

The Java Cryptographic Architecture (JCA), Engine Classes, and Providers

The JCA is a framework for working with cryptographic services, such as digital signature algorithms, message digest algorithms, and key conversion services.

The JCA defines classes that provide the functionality of these cryptographic services. These classes are called *engine classes*. An engine class provides the interface to a specific type of cryptographic service, independent of a particular algorithm or provider.

The JCA includes both *cryptographic engine classes*, which support cryptographic algorithms and *non-cryptographic engine classes*, which support non-cryptographic algorithms.

Cryptographic engine classes that support cryptographic algorithms provide one of the following:

- Cryptographic operations, for example, encryption, digital signatures, and message digests; these engine classes include `Cipher`, `Mac`, `KDF`, `KEM`, `KeyAgreement`, `MessageDigest`, and `Signature`
- Generators or converters of cryptographic material such as keys and algorithm parameters; these engine classes include `KeyGenerator`, `KeyFactory`, `SecretKeyFactory`, `SecureRandom`, `AlgorithmParameters`, and `AlgorithmParameterGenerator`

Non-cryptographic engine classes that support non-cryptographic algorithms provide one of the following:

- Objects, such as keystores and certificates, that encapsulate cryptographic data and can be used at higher layers of abstraction; these engine classes include `KeyStore`, `CertificateFactory`, `CertPathBuilder`, `CertPathValidator`, and `CertStore`
- Protocols, such as TLS, and other higher level functionality that employ cryptographic operations and the objects that encapsulate cryptographic data; these engine classes include `SSLContext`, `KeyManagerFactory`, and `TrustManagerFactory`

A CSP, which is used interchangeably with the term "provider," is a package or set of packages that implement one or more cryptographic services. To use the JCA, an application requests a particular type of object, such as a `MessageDigest`, and a particular algorithm or service, such as the SHA-256 algorithm, and gets an implementation from one of the installed providers. For example, the following statement requests a SHA-256 message digest from an installed provider:

```
md = MessageDigest.getInstance("SHA-256");
```

Alternatively, the program can request objects from a specific provider. For example, the following statement requests a SHA-256 message digest from Jipher:

```
md = MessageDigest.getInstance("SHA-256", "JipherJCE");
```

See Java Cryptography Architecture (JCA) Reference Guide in *Java Platform, Standard Edition Security Developer's Guide* for more information.

FIPS 140

The Federal Information Processing Standards (FIPS) of the United States are a set of publicly announced standards published by the National Institute of Standards and Technology (NIST). The FIPS 140 standards detail security requirements for cryptographic modules. Accredited Cryptographic and Security Testing Laboratories (CSTLs) test that cryptographic modules meet FIPS 140 security requirements. Modules that meet these security requirements are issued a Cryptographic Module Validation Program (CMVP) certificate.

Organizations that must comply with the Federal Information Security Modernization Act (FISMA) include state agencies and private sector companies with government contracts. FISMA mandates the use of FIPS 140 approved or allowed cryptography provided by cryptographic modules with CMVP certificates.

You don't have to submit Jipher for testing to a CSTL nor do you need to acquire a CMVP certificate.

> **Note**
>
> FIPS 140-3 is the latest version of this standard, which supersedes the previous version, FIPS 140-2. In September 2026, FIPS 140-2 validation certificates will be placed on the historical list. This means that Federal Agencies should not include them in new systems but can procure them for legacy systems. See FIPS 140-3 Transition Effort and Cryptographic Module Validation Program.

Jipher Artifacts

Jipher is packaged in a `.tar.gz` file named `jipher-20.0-<platform>.tar.gz`. The value of `<platform>` is one of the following, which correspond to each supported platform:

- `linux-aarch64`
- `linux-x64`
- `macos-aarch64`
- `windows-x64`

This `.tar.gz` file contains the following:

- The Jipher JAR file, which is named `jipher-jce-20.0.jar`.
- A `.tar.gz` file named `jipher-native-20.0-<platform>.tar.gz`. This archive contains:
  - The Jipher native library, which facilitates loading the OpenSSL cryptography library through the JNI on JDK 17 and 21
  - The OpenSSL cryptography library, which facilitates loading and communicating with the OpenSSL FIPS provider
  - Two OpenSSL FIPS modules:
    - **Certified OpenSSL module**: A version of the OpenSSL FIPS module built from source code that has been tested by a CSTL, validated by the CMVP, and issued a FIPS 140-3 validation certificate
    - **Patched OpenSSL module**: A version of the OpenSSL FIPS module built from the FIPS 140-3 compliant baseline source code with additional security patches applied, which is used by default

Independence from and Coexistence with Other Instances of OpenSSL

Jipher's native runtime dependencies, which include an instance of the OpenSSL cryptography library, are distributed in the jipher-native package and should be installed in the target filesystem. By default, Jipher loads and uses this bundled OpenSSL instance at runtime. Jipher therefore does not rely on the OpenSSL version provided by the operating system or by another installed package.

The OpenSSL cryptography library packaged in jipher-native is built so that references to symbols defined within the library resolve to that library's own definitions. In addition, Jipher loads the library without exporting its public symbols to the process-wide dynamic-linker namespace.

Together, these measures reduce the risk of symbol interposition when Jipher runs in a process that also loads another OpenSSL instance.

2

Downloading and Deploying Jipher

Downloading Jipher

Download Jipher from Java Verified Portfolio.

> **Note**
>
> On macOS, downloaded files are usually marked with a quarantine attribute. After verifying the SHA-256 hash of the downloaded package remove the quarantine attribute before installing the the package contents on macOS.
> `xattr -d com.apple.quarantine jipher-20.0-macos-aarch64.tar.gz`

Jipher Deployment

1. Extract Jipher's native library dependency archive, `jipher-native-20.0-<platform>.tar.gz`, from `jipher-20.0-<platform>.tar.gz`.
2. Install Jipher's native library dependencies by extracting them from the native library dependency archive to a location in the file system.

   Jipher's native library dependency archive contains the following:

   - **Certified OpenSSL module**: A version of the OpenSSL FIPS module built from source code that has been tested by an accredited Cryptographic and Security Testing Laboratory (CSTL), validated by the Cryptographic Module Validation Program (CMVP), and issued a FIPS 140-3 validation certificate
   - **Patched OpenSSL module**: A version of the OpenSSL FIPS module built from the FIPS 140-3 compliant baseline source code with additional security patches applied, which is used by default

   The following command extracts the patched OpenSSL module for a `PLATFORM` to a `TARGETPATH`:

   ```
   tar -xf jipher-native-20.0-${PLATFORM}.tar.gz -C "${TARGETPATH}" --strip-components=2 "patched/${PLATFORM}"
   ```

   The following command extracts the certified OpenSSL module for a `PLATFORM` to a `TARGETPATH`:

   ```
   tar -xf jipher-native-20.0-${PLATFORM}.tar.gz-C "${TARGETPATH}" --strip-components=2 "certified/${PLATFORM}"
   ```

> **Note**
>
> By default Jipher loads its native library dependencies from a platform-specific location:
>
> - Linux and macOS: `/opt/jipher`
> - Windows: `C:\Program Files\jipher`
>
> If you extract the native library dependencies to this location then no further configuration is required. Otherwise, you must specify the location of Jipher's native library dependencies through system properties. See Configuring Jipher Through System and Security Properties.

> **Caution**
>
> Ensure that Jipher's native library dependencies and the directory in which these files are stored are owned by a highly privileged operating system account and write permissions are restricted to that account (or an equivalent administrative group). Standard users should not have permission to modify, replace, or delete these files. Restricting write access in this manner reduces the risk of an attacker replacing the native library dependency libraries with malicious versions.

3. Extract the Jipher JAR file, `jipher-jce-20.0.jar`, from `jipher-20.0-<platform>.tar.gz`. Add the Jipher JAR file to your class path or your module path. When placed on the module path, it will be observable as the module `com.oracle.jipher`.

> **Note**
>
> The JCA authenticates a security provider by verifying the JAR file's signature at runtime. See Registering the SUN Provider for JAR File Signature Verification. Consequently, you cannot extract the contents of the Jipher JAR file and incorporate them into a JAR with dependencies (a JAR file that contains not only a Java application but includes its dependencies) to simplify deployment.

Permit Native Access

Starting in JDK 25, Jipher uses the Foreign Function and Memory (FFM) API to access OpenSSL. Some of the FFM API methods used by Jipher are restricted. See Restricted Methods in *Java Platform, Standard Edition Core Libraries*. To enable Jipher to access these methods, add one of the following command-line options when you run your Java application depending on where you specified the Jipher JAR file:

- On the class path:

  ```
  --enable-native-access=ALL-UNNAMED
  ```

- On the module path:

  ```
  --enable-native-access=com.oracle.jipher
  ```

Check the Version of Your Jipher Deployment

Jipher is distributed as a Java module. Launching the module invokes its main class, which prints version information and then exits.

Running the module is a simple way to verify that the Jipher native libraries can be loaded successfully. If running the module prints version information without any errors, then the native libraries have been loaded correctly.

Run the Jipher module as follows:

```
java --enable-native-access=com.oracle.jipher \
     --module-path jipher-jce-20.0.jar \
     --module com.oracle.jipher
```

This command should print output similar to the following:

```
JipherJCE Provider 20.0 [OpenSSL 3.6.3+oracle.jipher 9 Jun 2026 with OpenSSL FIPS
Provider version 3.5.7] (implements AES, DESede, Diffie-Hellman, DSA, ML-KEM, ML-
DSA, ECDSA, ECDH, HMAC, PBKDF2, RSA, SHA-1, SHA-2, SHA-3)
```

See Reporting Misconfiguration if this command reports any errors.

3

Configuring an Application to Achieve FIPS 140-3 Compliance with Jipher

To achieve FIPS 140-3 compliance, your applications must use only FIPS-approved or NIST-recommended cryptography provided by CMVP-certified cryptographic modules. However, none of the providers included in the JDK are CMVP-certified.

Configure your application to achieve FIPS 140-3 compliance with Jipher by following these steps:

> **Note**
>
> If you haven't already done so, download and deploy Jipher as described in Downloading and Deploying Jipher.

1. Register Jipher and other security providers required by the JCA and your application as described in Provider Registration:
   a. Registering Jipher as the Most Preferred Provider: It's recommended that you register Jipher as the most preferred provider, in position 1 in the list of security providers.
   b. Registering the SUN Provider for JAR File Signature Verification: The JCA requires the SUN provider for JAR file signature verification. The JDK requires this when a provider is first used to perform an encryption operation.
   c. Registering Other Security Providers: This section lists non-cryptographic and cryptographic engine class algorithms not supported by Jipher, which your application might require, and some guidance about registering the security providers that provide these algorithms.

   You can register a provider included in the JDK as long as it's not used to provide a cryptographic engine class algorithm. (See The Java Cryptographic Architecture (JCA), Engine Classes, and Providers.) However, it can be difficult to ensure that registering one will not cause it to be used to provide a cryptographic engine class algorithm. For instance, third-party dependencies and providers themselves can use other registered providers to provide cryptographic engine class algorithms. It's therefore good practice to only register providers required by your application. One option is to register a provider and then unregister it once your application no longer needs it. See Registering the SUN Provider for JAR File Signature Verification for an example.
2. If you're using the SunJSSE provider, which enables secure Internet communications by providing implementations of the SSL, TLS, and DTLS protocols, then follow the steps described in Configure the SunJSSE Provider to Select Jipher to Provide Cryptography. The SunJSSE provider doesn’t provide its own cryptographic engine class algorithms for these protocols. Instead, it uses the JCA to request implementations of cryptographic engine class algorithms from other providers. These steps ensure that the SunJSSE provider uses cryptographic engine class algorithms provided by Jipher and no other provider.
3. In your application code, ensure that Jipher is the only security provider used to provide cryptography. Follow the guidance in these sections:

- Explicitly Selecting Jipher to Provide Cryptography
- Implicitly Selecting Jipher to Provide Cryptography

> **Note**
>
> If your application requires a non-FIPS 140-3 allowed algorithm, then to achieve FIPS 140-3 compliance, you must change your application to no longer use that non-FIPS 140-3 allowed algorithm.

4. If required, configure some of Jipher's features through system properties. See Configuring Jipher Through System and Security Properties.

Provider Registration

To use a security provider that's not included in the JDK, such as Jipher, you must register it so that the JCA can access its security services. You can register a provider statically or dynamically.

Registering Jipher as the Most Preferred Provider

It's recommended that you register Jipher as the most preferred provider, in position 1 in the list of security providers.

When an application requests an instance of an algorithm, without specifying which provider should provide the implementation, the JCA searches the list of registered providers in preference order. The JCA returns the implementation from the first provider supplying the algorithm. Registering Jipher as the most preferred provider ensures that the JCA will select it to provide certified implementations of algorithms allowed by FIPS 140-3 standards.

> **Note**
>
> Registering Jipher as the most preferred provider does not guarantee that it will be selected to provide an implementation of an algorithm. The following are some reasons why Jipher might not be selected:
>
> - The algorithm is not approved or allowed by FIPS 140-3 standards.
> - The application wants to initialize the algorithm with a parameter not allowed by FIPS 140-3 standards because, for example, the key size is too weak or the algorithm parameters are not approved.
> - The application wants to use the algorithm for an operation not allowed by FIPS 140-3 standards, for example, using SHA-1 to digest data to be signed.
> - The algorithm is approved or allowed by FIPS 140-3 standards but Jipher doesn't provide an implementation of it.
>
>   For example, the SUN provider provides implementations of algorithms that are approved or allowed by FIPS 140-3 standards that Jipher doesn't provide. These include truncated digests such as SHA-512/224 and SHA-512/256.
>
>   As a result, the SUN provider (or another registered provider that doesn't provide FIPS 140-3 validated cryptography) could be selected to provide an implementation of an algorithm that is approved or allowed by FIPS 140-3 standards even though Jipher is registered as the most preferred provider.
>
>   See Registering Other Security Providers for a list of algorithms that the SUN provider supports that aren't supported by Jipher.

You register a provider like Jipher in the following ways:

- Statically by specifying it in the list of registered providers in the `java.security` file
- Dynamically by calling either the `addProvider` or `insertProviderAt` method in the `java.security.Security` class in your application code

See Step 8.1: Configure the Provider in How to Implement a Provider in the Java Cryptography Architecture in *Java Platform, Standard Edition Security Developer's Guide* for steps on how to do this.

> **Note**
>
> It might be more fragile to register Jipher statically than dynamically. The following describes how Jipher could be configured incorrectly and result in Jipher not being used as expected. Of greatest concern is that this consequence might not be readily apparent. Your application still works and doesn't report any errors, but it's not using Jipher or FIPS 140-3 certified cryptography.
>
> - Suppose the Jipher provider classes are not available to a JVM instance because, for example, the `jipher-jce-20.0.jar` JAR file is not in the class path. As a result, the JVM instance will silently fail to register Jipher when configured to statically register it. The JVM instance will behave as though it wasn't configured to statically register Jipher. This will in turn impact your application's FIPS 140-3 compliance.
> - If the JDK's `java.security` file has been edited to statically register Jipher, then it might need to be re-edited if the JDK was reinstalled.

The following example is an excerpt from the `java.security` file. It registers Jipher in position 1 by specifying its provider name, `JipherJCE`. It also registers the SUN provider in position 2, and SunJSSE in position 3:

```
security.provider.1=JipherJCE
security.provider.2=SUN
security.provider.3=SunJSSE
# Other registered providers follow...
```

> **Note**
>
> The list of statically registered providers is often longer. A more complete list better ensures that a requested algorithm that is not supported by Jipher will be provided by another provider.

In this example, because Jipher is registered at position 1, the JCA will retrieve cryptographic engine class algorithm implementations from it first. If the JCA can't retrieve an implementation of a requested algorithm from Jipher, it will attempt to retrieve an implementation from one of the other registered providers. For example a request for the MD5 message digest algorithm, which is not a FIPS 140-3 allowed algorithm, would result in the JCA retrieving the implementation provided by the SUN provider.

Registering the SUN Provider for JAR File Signature Verification

You must register the SUN provider because it provides an X.509 `CertificateFactory`, which Jipher doesn't provide, that is required for JAR file signature verification.

Security providers, like Jipher, that provide algorithms for engine classes from the `javax.crypto` package must be in a signed JAR file. The verification of a JAR file's signature occurs when getting an algorithm instance from a class in the `javax.crypto` package. If JAR file signature verification fails, then the provider is not selected to provide the implementation.

If you want to minimize the number of registered providers in your application, you can statically register the SUN provider, which is required for JAR file signature verification, then dynamically unregister the SUN provider once Jipher's JAR file signature is verified. The following example demonstrates this:

```
static {

    // Dynamically register the "JipherJCE" provider if it has not already
been statically registered.
    if (Security.getProvider("JipherJCE") == null) {
        Provider jipherJCE = ServiceLoader.load(Provider.class).stream()
            .filter(provider ->
provider.type().getName().equals("com.oracle.jipher.provider.JipherJCE"))
            .map(ServiceLoader.Provider::get).findFirst()
            .orElseThrow(() -> new IllegalStateException("JipherJCE provider
not found"));
        Security.insertProviderAt(jipherJCE, 1);
    }

    // Trigger the javax.crypto.JarVerifier's certificate verification self-
test.
```

```
    // This assumes that the "SUN" provider has been statically registered.
    try {
        Cipher.getInstance("AES", "JipherJCE");
    } catch (NoSuchPaddingException | NoSuchAlgorithmException |
NoSuchProviderException e ) {
        throw new RuntimeException(e);
    }

    // Dynamically unregister the "SUN" provider
    Security.removeProvider("SUN");
}
```

Registering Other Security Providers

As described in The Java Cryptographic Architecture (JCA), Engine Classes, and Providers, some providers included in the JDK provide both cryptographic engine class algorithms and non-cryptographic engine class algorithms. However, Jipher doesn't provide any non-cryptographic engine class algorithms.

In addition, the SUN provider is needed for provider JAR file signature verification. The JDK requires this when a provider is first used to perform an encryption operation. See Registering the SUN Provider for JAR File Signature Verification.

However, the fewer security providers that are registered, the lower the chance that a provider other than Jipher will be selected to provide a cryptographic engine class algorithm that Jipher doesn't support (due to FIPS 140-3 restrictions) or, in the case of delayed provider selection (see Delayed Provider Selection), a cryptographic engine class algorithm that Jipher does support initialized with a `Key` it does not support (due to FIPS 140-3 restrictions).

The SUN provider is optionally required to provide support for the following non-cryptographic engine class algorithms:

- Loading certificates and building/validating certificate paths:
  - `X.509` `CertificateFactory` algorithm
  - `PKIX` `CertPathBuilder` and `CertPathValidator` algorithms
  - `Collection` `CertStore` algorithm
- Loading keystores and truststores:
  - `PKCS12` `KeyStore` algorithm

The SunJSSE provider is optionally required to provide TLS support through these non-cryptographic engine class algorithms:

- `PKIX` `KeyManagerFactory` and `TrustManagerFactory` algorithms
- All the `SSLContext` algorithms

Note that the SUN provider supports cryptographic engine class algorithms that are supported by Jipher. Consequently, if you register the SUN provider, it's important that Jipher is registered with a higher preference than it as described in Registering Jipher as the Most Preferred Provider.

The SUN provider supports the following cryptographic engine class algorithms that are not supported by Jipher:

- `KeyFactory`:

  - HSS/LMS
- `MessageDigest`:
  - MD2
  - MD5
  - SHA-512/224
  - SHA-512/256
- `Signature`:
  - HSS/LMS
  - NONEwithDSAinP1363Format
  - SHA1withDSAinP1363Format
  - SHA224withDSAinP1363Format
  - SHA256withDSAinP1363Format
  - SHA384withDSAinP1363Format
  - SHA512withDSAinP1363Format
  - SHA3-224withDSA
  - SHA3-256withDSA
  - SHA3-384withDSA
  - SHA3-512withDSA
  - SHA3-224withDSAinP1363Format
  - SHA3-256withDSAinP1363Format
  - SHA3-384withDSAinP1363Format
  - SHA3-512withDSAinP1363Format

The SunJSSE provider does not support any cryptographic engine class algorithms not supported by Jipher.

Delayed Provider Selection

Java cryptographic API classes that generate, import, or can be initialized with a key (`KeyGenerator`, `KeyPairGenerator`, `KeyFactory`, `SecretKeyFactory`, `Cipher`, `KeyAgreement`, `Mac`, and `Signature`) support delayed provider selection. A `getInstance(String algorithm)` method call compiles a preference ordered list of providers that support the algorithm. Later, when the `init` method is called, the first provider in the compiled preference ordered list of providers that also supports the key (with regards to size and format) is selected.

Configure the SunJSSE Provider to Select Jipher to Provide Cryptography

The SunJSSE provider enables secure Internet communications by providing implementations of the SSL, TLS, and DTLS protocols. It doesn't provide its own cryptographic engine class algorithms for these protocols. Instead, it uses the JCA to request implementations of cryptographic engine class algorithms from other providers. Follow these configuration steps to

ensure that the SunJSSE provider uses cryptographic engine class algorithms provided by Jipher and no other provider:

1. Permit Access to Internal JDK API Classes: This ensures that Jipher is allowed to provide cryptography requested by the SunJSSE provider.
2. Registering Jipher as the Most Preferred Provider: This ensures that Jipher will be chosen ahead of any other providers to provide an implementation of any algorithm it supports. Also, register the SUN provider as described in Registering the SUN Provider for JAR File Signature Verification.
3. Limit Protocol Versions, Cipher Suites, Key Material, and Named Groups: This ensures that the SunJSSE provider never requests cryptography that Jipher does not support.

These steps combined ensure that the SunJSSE provider never uses any cryptography provided by a non-FIPS 140-3 certified registered provider.

> **Note**
>
> If the following is true about your environment, then you only have to follow the steps Permit Access to Internal JDK API Classes and Configure SSLContext with Key Material That Provides at Least 112-Bits of Security:
>
> - You're using JDK 21 or JDK 25.
> - It uses a standard JDK security property configuration where:
>   - The `jdk.certpath.disabledAlgorithms` security property specifies `MD2` and `MD5`.
>   - The `jdk.tls.disabledAlgorithms` security property specifies `SSLv3`, `TLSv1`, `TLSv1.1`, and `MD5withRSA`.
> - Jipher is registered as the most preferred provider and the only other registered providers are SUN and SunJSSE. See Registering Jipher as the Most Preferred Provider.

Permit Access to Internal JDK API Classes

When a security provider provides the cryptography required by the SunJSSE to provide TLSv1.2, it must access internal JDK API classes.

If Jipher attempts to access them, an `IllegalAccessException` is thrown. To prevent this from happening, add one of the following series of command-line options when you run your Java application depending on where you specified the Jipher JAR file

- On the class path:

  ```
  --add-exports=java.base/sun.security.internal.spec=ALL-UNNAMED
  ```

- On the module path:

  ```
  --add-exports=java.base/sun.security.internal.spec=com.oracle.jipher
  ```

When a security provider provides the cryptography required by the SunJSSE to provide Hybrid TLS 1.3 named group support, it might need to access an internal JDK utility class - `sun.security.util.RawKeySpec`.

If Jipher attempts to access the `RawKeySpec` class, an `IllegalAccessException` is thrown. To prevent this from happening, add one of the following series of command-line options when you run your Java application depending on where you specified the Jipher JAR file

- On the class path:

  ```
  --add-exports=java.base/sun.security.util=ALL-UNNAMED
  ```

- On the module path:

  ```
  --add-exports=java.base/sun.security.util=com.oracle.jipher
  ```

Limit Protocol Versions, Cipher Suites, Key Material, and Named Groups

This section describes which SunJSSE parameters and properties to limit to the same levels and values as Jipher. This ensures that the SunJSSE provider never requests cryptography that Jipher does not support.

Ensure That Clients and Servers Only Use TLS 1.2 and TLS 1.3

Ensure that the security property `jdk.tls.disabledAlgorithms` includes the values `SSLv3`, `TLSv1`, and `TLSv1.1`.

In addition, ensure that the system properties `jdk.tls.server.protocols` and `jdk.tls.client.protocols` contain the values `TLSv1.2` and `TLSv1.3`. See Customizing JSSE in *Java Platform, Standard Edition Security Developer's Guide* for more information about these system properties and other properties you can specify to customize JSSE.

In your code, call `SSLContext.getInstance("TLS")` instead of supplying specific TLS protocol versions.

> **Note**
>
> Calling `SSLContext.getInstance("TLS1.3")` sets the ceiling for protocol versions. As a result, this statement returns a context that allows TLS 1.0 and TLS 1.1 protocol versions.

Ensure TLS Cipher Suites Use FIPS 140-3 Algorithms from Jipher

Depending on which TLS protocol version you want to support (TLS 1.2 or TLS 1.3), specify the cipher suites that only require FIPS 140-3 allowed cryptography provided by Jipher as a comma-separated list for the system properties `jdk.tls.client.cipherSuites` and `jdk.tls.server.cipherSuites`. See Specifying Default Enabled Cipher Suites in *Java Platform, Standard Edition Security Developer's Guide* for more information about these system properties.

See The SunJSSE Provider in *Java Platform, Standard Edition Security Developer's Guide* for a complete list of supported cipher suites.

TLS 1.2 cipher suites are expressed as follows:

`TLS_<key exchange algorithm>_<authentication algorithm>_WITH_<cipher algorithm>_<cipher strength>_<cipher mode>_<HASH or MAC>`

TLS 1.3 cipher suites are expressed as follows:

`TLS_<cipher algorithm>_<cipher strength>_<cipher mode>_<HASH or MAC>`

The following table lists the values within a cipher suite's name that are supported by Jipher:

Table 3-1 Values Within a Cipher Suite's Name Supported by Jipher

Component	Supported	Supported but Not Recommended
Key exchange algorithm	ECDHE, DHE	DH
Authentication algorithm	ECDSA, RSA	DSS
Cipher algorithm	AES	—
Cipher strength	256, 128	—
Cipher mode	GCM	CBC
HASH or MAC	SHA384, SHA256	SHA

> **Note**
>
> Cipher suites that use RSA for key exchange and authentication are not supported. They are expressed as follows:
> `TLS_RSA_WITH_<cipher algorithm>_<cipher strength>_<cipher mode>_<HASH or MAC>`
>
> See "Appendix D—RSA Key Transport" in NIST SP 800-52 Rev. 2: Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations for the reasons why these cipher suites are not supported. In short, it is because RSA key transport as used in TLS versions 1.0 through 1.2 is implemented using the PKCS #1 v1.5 padding scheme.
>
> See Disabling TLS_RSA Cipher Suites to disable these cipher suits in JDK releases prior to JDK 24.

Configure SSLContext with Key Material That Provides at Least 112-Bits of Security

Key managers are responsible for managing the key material which is used to authenticate the local TLS endpoint to its peer. Configure your key manager with key material with the following parameters so that the key material provides at least 112-bits of security:

- RSA: Modulus >= 2048
- EC: secp224r1, secp256r1, secp384r1, or secp521r1

See the section KeyManagerFactory Class in *Java Platform, Standard Edition Security Developer's Guide*.

> **Note**
>
> The system property `jdk.tls.disabledAlgorithms` has no impact on key size used by the local TLS endpoint to generate a digital signature to authenticate itself to its peer.
> While `jdk.tls.disabledAlgorithms=DSA`, disables DSA (disables DSS cipher suites), `jdk.tls.disabledAlgorithms=DSA keySize < 2048` neither disables DSS cipher suites nor does it prevent the local TLS endpoint from using a DSA private key of less than 2048 bits when generating a signature to authenticate itself to its peer.

Configure the Size of Ephemeral Diffie-Hellman Keys

Set the system property `jdk.tls.ephemeralDHKeySize` to `2048`.

Ensure the JSSE Only Uses FIPS 140-3 Approved Named Groups

FIPS 140-3 compliance requires the use of approved elliptic curves and safe-prime groups for key establishment as documented in "Appendix D: Approved ECC Curves and FFC Safe-prime Groups" in NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography.

Set the system property `jdk.tls.namedGroups` to only allow the following:

- FIPS 140-3 approved named curves:
  - `secp256r1`
  - `secp384r1`
  - `secp521r1`
- FIPS 140-3 approved FFC safe-prime Groups:
  - `ffdhe2048`
  - `ffdhe3072`
  - `ffdhe4096`
- Hybrid TLS 1.3 key-exchange groups supported in JDK 27:
  - `X25519MLKEM768`— X25519 + ML-KEM-768
  - `SecP256r1MLKEM768`— secp256r1 + ML-KEM-768
  - `SecP384r1MLKEM1024`— secp384r1 + ML-KEM-1024

Explicitly Selecting Jipher to Provide Cryptography

When an application calls `getInstance` from an engine class such as `MessageDigest`, `Signature`, `KeyFactory`, `KeyPairGenerator`, or `Cipher` to acquire an instance of a cryptographic engine class algorithm, it can explicitly specify which provider should provide the algorithm by calling one of the following:

- `getInstance(String algorithm, String provider)`
- `getInstance(String algorithm, Provider provider)`

One way to ensure that Jipher is the only security provider used to provide cryptography is to explicitly specify that Jipher provide the instance of the algorithm in all `getInstance` method calls used to acquire an instance of a cryptographic engine class algorithm, for example:

```
Cipher.getInstance("AES", "JipherJCE");
```

In practice, this is often infeasible because of the following:

- Applications often need to use other security providers to provide higher level functionality, and these providers will internally make `getInstance(String algorithm)` calls to acquire the cryptographic engine class algorithms they need to deliver the higher level functionality they provide. For example the PKCS12 `KeyStore` algorithm provided by the SUN provider internally uses other security providers to provide the cryptographic engine class algorithms it uses to secure the key store.
- Applications often use third-party libraries that internally make `getInstance(String algorithm)` calls to acquire cryptographic engine class algorithms they need to deliver the functionality they provide.

Consequently, if your application, or one of its dependencies, calls `getInstance(String algorithm)` from an engine class, then configure the JVM so that Jipher is implicitly selected to provide the algorithm. See Implicitly Selecting Jipher to Provide Cryptography.

Implicitly Selecting Jipher to Provide Cryptography

To implicitly select Jipher whenever an application or one of its dependencies calls an engine class's `getInstance(String)` method, follow these steps:

1. Register Jipher as the most preferred provider. See Provider Registration.
2. Configure your application and its dependencies to only request cryptographic engine class algorithms that Jipher supports. For example, if you're using the SunJSSE provider to provide secure Internet communications, then follow the steps described in Configure the SunJSSE Provider to Select Jipher to Provide Cryptography. If your application requires a non-FIPS 140-3 allowed algorithm, then to achieve FIPS 140-3 compliance, you must change your application to no longer use that non-FIPS 140-3 allowed algorithm.

Configuring Jipher Through System and Security Properties

You can configure some of the features of Jipher through the following properties. Unless otherwise specified, these are system properties:

Table 3-2 Jipher System Properties

System Property	Description	Default Value
`java.security.debug`	Standard Java system property. If the value includes `jipher` or `all`, then debug logging through `System.err` is enabled within Jipher. This prints debugging information, including the library loading steps that are performed when Jipher is first used.	*None*

Table 3-2 (Cont.) Jipher System Properties

System Property	Description	Default Value
`jipher.fips.deactivateSecurityPatches`	Specifies whether Jipher prioritizes security (`false`) or compliance (`true`). The jipher-native package provides two copies of the OpenSSL FIPS module for each supported platform: • **Certified OpenSSL module**: A version of the OpenSSL FIPS module built from source code that has been tested by a CSTL, validated by the CMVP, and issued a FIPS 140-3 validation certificate • **Patched OpenSSL module**: A version of the OpenSSL FIPS module built from the FIPS 140-3 compliant baseline source code with additional security patches applied, which is used by default The system property `jipher.fips.deactivateSecurityPatches` can have one of the following values: • `false`: Jipher prioritizes security over compliance. At runtime, it verifies that it's using the patched OpenSSL module with the additional security patches. • `true`: Jipher prioritizes compliance over security. At runtime, it verifies that it's using the certified OpenSSL module without the additional security patches. **⚠ Caution** Setting `jipher.fips.deactivate.security.patches` to `true` incurs the serious risk that security vulnerabilities, published as Common Vulnerabilities and Exposures (CVE) in the public domain, will become exploitable in your deployment. However, any change to the OpenSSL FIPS module following FIPS 140-3 validation renders it non-compliant, so Jipher's default operation, setting `jipher.fips.deactivate.security.patches` to `false`, is not strictly FIPS 140-3 compliant with FISMA. Some users prefer to be strictly FIPS 140-3 compliant and use the CSTL-tested and CMVP-validated OpenSSL FIPS module that has been issued a FIPS 140-3 validation certificate. Others, like Oracle, use a recent baseline of the OpenSSL FIPS module that is FIPS 140-3 compliant, add security patches, and disclose this choice to customers and auditors. Regardless of whether you choose to be strictly FIPS 140-3 compliant or use the OpenSSL FIPS module with the additional security patches, Oracle recommends disclosing this choice to customers and auditors.	`false`

Table 3-2 (Cont.) Jipher System Properties

System Property	Description	Default Value
`jipher.fips.enforcement`	Specifies the FIPS 140-3 enforcement policy to be applied at the Java security provider layer. It can have one of the following values: • `FIPS`: Approved algorithms and key lengths are permitted in accordance with how they are used. Legacy use is permitted. • `FIPS_STRICT`: Only algorithms and key lengths with Acceptable approval status are permitted. • `NONE`: No additional enforcement is performed; any enforcement implemented in OpenSSL is still applied. This policy is intended to be used during preproduction debugging. You may encounter behavior that differs from the other policies such as: – A different exception being thrown – A delay in an exception being thrown; for example, an exception may be thrown when a signature object is used instead of when it is created or initialized *Acceptable* is used by NIST to mean that the algorithm and key length is safe to use. *Legacy use* means that the algorithm or key length may be used only to process already protected information, for example, to decrypt ciphertext data or to verify a digital signature. See NIST SP 800-131A Rev. 2: Transitioning the Use of Cryptographic Algorithms and Key Lengths and FIPS 140-3 Enforcement Policy for more information.	`FIPS`
`jipher.jniadapter.jipherffi.dir`	This system property is active only when Jipher runs on a JDK release earlier than 25. Specifies the path location that Jipher uses to load its JNI adapter library (`libjipherffi.dylib`, `libjipherffi.so`, or `jipherffi.dll`). If this property is unspecified, then Jipher will attempt to load its JNI adapter library from a platform specific location: • Linux and macOS: `/opt/jipher/lib` • Windows: `C:\Program Files\jipher\lib`	*None*
`jipher.nonfips.supportPkcs12Kdf`	Configures Jipher to support the HmacPBE* MAC algorithms as defined in Appendix B.4: Keys for Password Integrity Mode in RFC 7292. These MAC algorithms use PKCS #12 KDF, which is not a FIPS-allowed algorithm. This opt-in, non-FIPS-allowed algorithm support is provided to enable the JipherJCE provider to provide all the cryptography required by the SUN provider's PKCS12 `KeyStore` service. On JDK 25 and earlier, the SUN provider's PKCS12 `KeyStore` service employs HmacPBE* MAC algorithms for the integrity check when reading or writing `KeyStore` instances from or to persistent storage.	`false`

Table 3-2 (Cont.) Jipher System Properties

System Property	Description	Default Value
`jipher.openssl.dir`	This system property is only active if `jipher.openssl.useOsInstance` is **not** set to `true`. The directory must contain the following files: • Linux: – `lib/libcrypto.so.3` – `lib/ossl-modules/fips.so` • macOS: – `lib/libcrypto.3.dylib` – `lib/ossl-modules/fips.dylib` • Windows: – `bin\libcrypto-3-x64.dll` – `lib\ossl-modules\fips.dll` If `lib64` exists, then it's preferred over `lib`. If neither `jipher.openssl.useOsInstance` or `jipher.openssl.dir` are specified, then Jipher will attempt to load the OpenSSL cryptographic library and FIPS provider from a platform-specific location: • Linux and macOS: `/opt/jipher/openssl` • Windows: `C:\Program Files\jipher\openssl`	*None*
`jipher.openssl.sanctioned.osProvided.cryptoLibraryVersions`	This system property is only active if `jipher.openssl.useOsInstance` is set to `true`. Jipher will fail at startup unless the version number of the OpenSSL cryptography library it loads matches a version sanctioned by this system property value. See Versioning Sanctioning Syntax for more information.	`[,]`
`jipher.openssl.sanctioned.osProvided.fipsProviderVersions`	This system property is only active if `jipher.openssl.useOsInstance` is set to `true`. Jipher will fail at startup unless the version number of the OpenSSL FIPS provider it loads matches a version sanctioned by this system property value. See Versioning Sanctioning Syntax for more information.	`[,]`
`jipher.openssl.useOsInstance`	**Only** supported on Oracle Linux version 9.4 or later. Configures Jipher to load the instance of the OpenSSL cryptographic library and FIPS provider managed by the operating system.	`false`
`jipher.pbkdf2.maximumIterationCount`	Specifies the maximum iteration count supported for pbkdf2 operations.	10,000,000
`jipher.pbkdf2.minimumPasswordLength`	Specifies the minimum password length enforced for pbkdf2 operations	8

Table 3-2 (Cont.) Jipher System Properties

System Property	Description	Default Value
`securerandom.strongAlgorithms` (security property)	Defines the list of known strong `SecureRandom` implementations. It is used by `SecureRandom.getInstanceStrong()`. If you use the default value of this security property, then `SecureRandom` will not be provided by the JipherJCE provider. If your application uses `getInstanceStrong()` and the JipherJCE provider, then set this security property to `JipherJCE:DRBG`. **Note** Jipher defers to OpenSSL's FIPS Random Bit Generator (RBG) for Jipher cryptographic operations. A `SecureRandom` supplied to `init`, `initialize`, or `newEncapsulator` is accepted for Java API compatibility but ignored as a source of random bytes. See Sourcing Cryptographically Secure Random Bits for more information.	Refer to your security properties file, which is, by default `$JAVA_HOME/conf/security/java.security`. See The Security Properties File in *Java Platform, Standard Edition Security Developer's Guide*.

Versioning Sanctioning Syntax

The values of the system properties `jipher.openssl.sanctioned.osProvided.cryptoLibraryVersions` and `jipher.openssl.sanctioned.osProvided.fipsProviderVersions` use this syntax to specify sanctioned versions.

Single Version

A single version can be sanctioned by specifying its version.

For example, only version 3.5.4 is sanctioned:

```
3.5.4
```

Version Interval

Version ranges can be sanctioned by defining an optionally unbounded interval. This is defined by its optional boundaries and the type of bracket used to indicate whether those boundaries are included or excluded:

```
<open-bracket> <from-version> <comma> <to-version> <close-bracket>
```

Omitting `<from-version>` means unbounded below. Omitting `<to-version>` means unbounded above.

Square brackets include the exact version:

- [: Inclusive lower bound

-]: Inclusive upper bound

Round brackets exclude the exact version:

- (: Exclusive lower bound
-): Exclusive upper bound

Examples:

- Versions after 3.5.1 and before 3.5.5 are sanctioned:

  ```
  (3.5.1,3.5.5)
  ```

- Versions from 3.5.1 onward are sanctioned:

  ```
  [3.5.1,]
  ```

- Versions before 3.6 are sanctioned:

  ```
  (,3.6)
  ```

- Versions until 3.6 are sanctioned:

  ```
  (,3.6]
  ```

Combining Multiple Individual Versions and Version Intervals

You can combine multiple individual versions and version intervals using a comma as a separator, which acts as a logical OR (union). The union comma separates "items," where an item is either a single version or a bracketed interval.

Examples:

- Only versions 3.5.1 and 3.5.4 are sanctioned:

  ```
  3.5.1,3.5.4
  ```

- Version 3.5.1 and versions after 3.5.4 are sanctioned:

  ```
  3.5.1,(3.5.4,)
  ```

FIPS 140-3 Enforcement Policy

Jipher can perform enforcement of FIPS 140-3 algorithm usage and key sizes according to how the algorithm and key is being used.

You can specify a FIPS 140-3 enforcement policy with the `jipher.fips.enforcement` system property. See Configuring Jipher Through System and Security Properties. The default policy, `FIPS`, permits legacy use. This means that an algorithm or key length with "legacy use" status may be used, but only to process already protected information, for example, to decrypt ciphertext data or to verify a digital signature. The `FIPS_STRICT` policy only permits algorithms and key lengths with Acceptable approval status. This is more restrictive though likely to be more stable over time. See NIST SP 800-131A Rev. 2: Transitioning the Use of Cryptographic Algorithms and Key Lengths for more information..

In general, the default `FIPS` policy is similar to the `FIPS_STRICT` policy except that the `FIPS_STRICT` policy requires longer key lengths for digital signature verification and MACs for various algorithms. The following table highlights these differences in **bold**.

Table 3-3 Minimum Key Lengths of the Default FIPS Policy and the FIPS_STRICT Policy

Rule	FIPS Policy	FIPS_STRICT Policy
AES key bits	>= 128 for symmetric encryption and decryption	>= 128 for symmetric encryption and decryption
DESede key bits	Not supported for symmetric encryption = 192 for symmetric decryption	Not supported for symmetric encryption **and decryption**
DH key bits	>= 2048 for key generation and key agreement	>= 2048 for key generation and key agreement
DSA parameter bits	Not supported for key generation and signature generation prime size > 512 bits for signature verification	Not supported for key generation, signature generation and signature verification
EC curve bits	>= 224 for key generation, signature generation and key agreement	>= 224 for key generation, signature generation, **verification**, and key agreement
EC curve bits	>= 160 for signature verification	*See the previous table cell in this column*
HMAC key bits	>= 0 for MAC	>= **112** for MAC
Message digest algorithm	!= SHA-1 for signature generation	!= SHA-1 for signature generation **and verification**
RSA key bits	>= 2048 for asymmetric encryption and decryption key generation and signature generation	>= 2048 for asymmetric encryption and decryption key generation and signature generation **and verification**
RSA key bits	>= 1024 for signature verification.	*See the previous table cell in this column*

4

Applying NIST Guidelines for TLS

NIST SP 800-52 Rev. 2: Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations specify that TLS servers should use minimum key sizes when using certain algorithms and must perform revocation checking of the client certificate when client authentication is used.

Configure Minimum Key Sizes for Certificate Path Validation

Update the security property `jdk.certpath.disabledAlgorithms` to add the following restrictions:

- `RSA keySize < 2048`
- `EC keySize < 256`

See Disabled and Restricted Cryptographic Algorithms in *Java Platform, Standard Edition Security Developer's Guide*.

Enable Revocation Checking

A certificate is a digitally signed statement, typically issued by a Certificate Authority (CA), vouching for the identity and public key of an entity. Certificates used in TLS can be revoked by the issuing CA if there is reason to believe that a certificate is compromised. NIST guidelines specify that servers must perform revocation checking of the client certificate when client authentication is used. In addition, the server must retrieve revocation information though the Online Certificate Status Protocol (OCSP) or Certificate Revocation Lists (CRLs).

See PKIX TrustManager Support and Client-Driven OCSP and OCSP Stapling in *Java Platform, Standard Edition Security Developer's Guide* for more information.

Follow these steps to enable revocation checking and client-driven OCSP.

1. Set the system property `com.sun.net.ssl.checkRevocation` to `true`.
2. Set the system property `ocsp.enable` to `true`.
3. Set the system property `com.sun.security.enableCRLDP` to `true`.

5

Jipher Performance Optimizations

KeyManager Selection

Only select NewSunX509, an `X509KeyManager` implementation in the SunJSSE provider, if your application explicitly requires support for multiple and dynamic keystores, for example, a server that provides Server Name Indication (SNI) support. See Multiple and Dynamic Keystores in *Java Platform, Standard Edition Security Developer's Guide* for more information.

Elliptic Curve Key Pair Generation

The performance of the elliptic curve key pair generation algorithm provided by Jipher varies depending on the elliptic curve.

Among the curves supported by Jipher, the National Security Agency (NSA) recommends the use of the secp384r1 curve; refer to the Commercial National Security Algorithm (CSNA) Suite 1.0. However, the secp256r1 curve also supported by Jipher provides better performance. See Supported Elliptic Curve Names.

6

Secure Coding Guidance

Avoiding Unnecessary In-Memory Buffering of Plaintext

The `Cipher` methods `update` and `doFinal` support data streaming. However, cipher transformations that use an AES KeyWrap algorithm (defined in RFC 3394: Advanced Encryption Standard (AES) Key Wrap Algorithm) such as AESWrap, AESWrapPad, AES/KW/NoPadding, and AES/KWP/NoPadding don't lend themselves to data streaming because all input data must be available before any of the input data can be fully processed. Consequently, if an AESWrap transform `Cipher` object is initialized with the `ENCRYPT_MODE` operation, any plaintext passed to an `update` method is copied into an internal buffer so that it may be later processed during a subsequent `doFinal` method call. The `Cipher` object's internal plaintext buffer is zeroed and freed when `doFinal` is invoked or when the `Cipher` object is garbage collected. Applications that want to avoid plaintext being buffered by an AESWrap transform `Cipher` object should avoid calling `update`. For example, consider the following code:

```
Cipher wrapper = Cipher.getInstance("AESWrap");
wrapper.init(Cipher.ENCRYPT_MODE, secretKey);
wrapper.update(plaintext);
byte[] cipherText = wrapper.doFinal();
```

You can replace it with the following:

```
Cipher wrapper = Cipher.getInstance("AESWrap");
wrapper.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] cipherText = wrapper.doFinal(plaintext);
```

7

Jipher Diagnostics

Confirming That a Java Application Is Using Jipher

If you set the system property `java.security.debug` to `jipher`, then Jipher will print additional information. The output is similar to the following:

```
jipher: FIPS Provider detected OpenSSL FIPS Provider
jipher: FIPS Provider version 3.5.7
jipher: FIPS Provider major version 3
jipher: FIPS Provider minor version 5
jipher: Setting FIPS enforcement policy = FIPS
jipher: Checking KEYGEN RSA size against FIPS Policy (FIPS)...
```

See The java.security.debug System Property in *Java Platform, Standard Edition Security Developer's Guide* for more information.

Keeping Track of Security Provider Usage with the jdk.SecurityProviderService Java Flight Recorder (JFR) Event

In JDK 20 and later, the Java Flight Recorder (JFR) event `jdk.SecurityProviderService` records the details of `java.security.Provider.getService(String type, String algorithm)` calls. This event contains the following fields:

Table 7-1 JFR Event jdk.SecurityProviderService Fields

Field Name	Description
type	Type of service
algorithm	Algorithm name
provider	Security provider

You can use the JFR event `jdk.SecurityProviderService` to confirm that a Java application is using Jipher. This JFR event is disabled by default. You can enable it through JFR configuration files or standard JFR options.

Reporting the Enforcement of FIPS 140-3 Restrictions

When enforcing FIPS 140-3 restrictions, Jipher throws an `InvalidParameterException` if directed to generate the following:

- A `SecretKey` or `KeyPair` with a security strength of less than 112 bits
- A Diffie-Hellman `KeyPair` using domain parameters that are not a FIPS 140-3 approved safe prime group (see "Appendix D: Approved ECC Curves and FFC Safe-prime Groups"

in NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography)

- An elliptic curve `KeyPair` using a curve that is not a FIPS 140-3 approved `secp` curve (see again "Appendix D: Approved ECC Curves and FFC Safe-prime Groups").

Similarly, Jipher throws a `ProviderException` if directed to use the following:

- A `KeyPair` with a security strength of less than 80 bits to process secured data, for example, to verify a signature or decrypt cipher text
- A `SecretKey` or `KeyPair` with a security strength of less than 112 bits to secure data, for example, to generate a digital signature or encrypt plaintext
- SHA-1 to generate a signature
- A DSA `KeyPair` that does not use domain parameters allowed by FIPS 140-3 (listed previously)

See "Table 2: Comparable security strengths of symmetric block cipher and asymmetric-key algorithms" in NIST SP 800-57 Part 1 Rev. 5: Recommendation for Key Management: Part 1 – General for the estimated security strengths of specific algorithms and key lengths.

Reporting Misconfiguration

If Jipher is unable to load its native library dependencies at run time, it throws the following:

```
java.security.ProviderException: OpenSSL is not available
```

Determine the failure's root cause from the final `Throwable` in the chain of exceptions:

Throwable	Message	Meaning
`java.lang.UnsatisfiedLinkError`	Can't load library: `<path to>/(lib)jipherffi.<ext>`	Jipher is unable to load its JNI native library. This happens if the file is missing. See `jipher.jniadapter.jipherffi.dir` in Table 3-2.
	`<path to>/(lib)jipherffi.<ext>`: Access is denied	Jipher is unable to load its JNI native library. This occurs if the file is not readable or cannot be run by the operating system user running the JVM process. See `jipher.jniadapter.jipherffi.dir` in Table 3-2.
`com.oracle.jipher.internal.openssl.OpenSslException`	OpenSSL crypto library path not found	Jipher is unable to load the OpenSSL cryptography library. This occurs if the file is missing, unreadable, or is not executable by the operating system user running the JVM process. See `jipher.openssl.dir` in Table 3-2.

Throwable	Message	Meaning
	FIPS provider is not available	Jipher is unable to load the OpenSSL FIPS provider. This occurs if the file is missing, unreadable, or is not executable by the operating system user running the JVM process. It also occurs if the instance of the OpenSSL FIPS module that Jipher attempts to load is inconsistent with the `jipher.fips.deactivateSecurityPatches` system property setting. If `jipher.fips.deactivateSecurityPatches` is not set to `true` and Jipher attempts to load the certified OpenSSL module, then an `OpenSslException` is thrown. If `jipher.fips.deactivateSecurityPatches` is set to `true` and Jipher attempts to load the patched OpenSSL module, then an `OpenSslException` is thrown. See `jipher.openssl.dir` and `jipher.fips.deactivateSecurityPatches` in Table 3-2.

Reporting Abnormal Operation in OpenSSL Native Code

An error condition that arises in OpenSSL native code is reported to the application through the following:

- A `java.lang.Error`, such as `java.lang.OutOfMemoryError`
- A `java.lang.RuntimeException`, either `java.lang.ArrayIndexOutOfBoundsException` or `java.lang.IllegalArgumentException`, indicating a programming error
- A `ProviderException` whose chained cause is an internal `OpenSslException` whose detail message describes the OpenSSL error stack for use in debugging and troubleshooting

8

Jipher Reference Information

Supported Algorithm Strings

The following table lists the algorithm strings and their aliases supported by Jipher. These strings are grouped by their associated engine class.

Table 8-1 Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
AlgorithmParameters	AES (2.16.840.1.101.3.4.1, OID.2.16.840.1.101.3.4.1, 2.16.840.1.101.3.4.1.2, OID.2.16.840.1.101.3.4.1.2, 2.16.840.1.101.3.4.1.3, OID.2.16.840.1.101.3.4.1.3, 2.16.840.1.101.3.4.1.4, OID.2.16.840.1.101.3.4.1.4, 2.16.840.1.101.3.4.1.6, OID.2.16.840.1.101.3.4.1.6, 2.16.840.1.101.3.4.1.22, OID.2.16.840.1.101.3.4.1.22, 2.16.840.1.101.3.4.1.23, OID.2.16.840.1.101.3.4.1.23, 2.16.840.1.101.3.4.1.24, OID.2.16.840.1.101.3.4.1.24, 2.16.840.1.101.3.4.1.26, OID.2.16.840.1.101.3.4.1.26, 2.16.840.1.101.3.4.1.42, OID.2.16.840.1.101.3.4.1.42, 2.16.840.1.101.3.4.1.43, OID.2.16.840.1.101.3.4.1.43, 2.16.840.1.101.3.4.1.44, OID.2.16.840.1.101.3.4.1.44, 2.16.840.1.101.3.4.1.46 OID.2.16.840.1.101.3.4.1.46) DESede (OID.1.2.840.113549.3.7, 1.2.840.113549.3.7)[1] GCM OAEP (1.2.840.113549.1.1.7, OID.1.2.840.113549.1.1.7)	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
	DH (DiffieHellman, 1.2.840.113549.1.3.1, OID.1.2.840.113549.1.3.1) EC (1.2.840.10045.2.1, OID.1.2.840.10045.2.1) RSASSA-PSS (1.2.840.113549.1.1.10, OID.1.2.840.113549.1.1.10)	—
	PBE PBES2 (1.2.840.113549.1.5.13, OID.1.2.840.113549.1.5.13)	—
	PBEWithHmacSHA1AndAES_128 PBEWithHmacSHA1AndAES_256 PBEWithHmacSHA224AndAES_128 PBEWithHmacSHA224AndAES_256 PBEWithHmacSHA256AndAES_128 PBEWithHmacSHA256AndAES_256 PBEWithHmacSHA384AndAES_128 PBEWithHmacSHA384AndAES_256 PBEWithHmacSHA512AndAES_128 PBEWithHmacSHA512AndAES_256	—
Cipher	AES (Rijndael, 2.16.840.1.101.3.4.1, OID.2.16.840.1.101.3.4.1) AES/CTR/NoPadding	—
	AES/KW/NoPadding (AESWrap, AES-KW) AES_128/KW/NoPadding (AESWrap_128, 2.16.840.1.101.3.4.1.5, OID.2.16.840.1.101.3.4.1.5) AES_192/KW/NoPadding (AESWrap_192, 2.16.840.1.101.3.4.1.25, OID.2.16.840.1.101.3.4.1.25) AES_256/KW/NoPadding (AESWrap_256, 2.16.840.1.101.3.4.1.45, OID.2.16.840.1.101.3.4.1.45)	RFC 3394
	AES/KWP/NoPadding (AESWrapPad, AES-KWP) AES_128/KWP/NoPadding (AESWrapPad_128, 2.16.840.1.101.3.4.1.8, OID.2.16.840.1.101.3.4.1.8) AES_192/KWP/NoPadding, (AESWrapPad_192, 2.16.840.1.101.3.4.1.28, OID.2.16.840.1.101.3.4.1.28) AES_256/KWP/NoPadding (AESWrapPad_256, 2.16.840.1.101.3.4.1.48, OID.2.16.840.1.101.3.4.1.48)	RFC 5649

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
	AES/GCM/NoPadding AES_128/CBC/PKCS5Padding (AES_128/CBC/PKCS7Padding, 2.16.840.1.101.3.4.1.2, OID.2.16.840.1.101.3.4.1.2) AES_128/CFB/NoPadding (2.16.840.1.101.3.4.1.4, OID.2.16.840.1.101.3.4.1.4) AES_128/ECB/NoPadding (2.16.840.1.101.3.4.1.1, OID.2.16.840.1.101.3.4.1.1) AES_128/GCM/NoPadding (2.16.840.1.101.3.4.1.6, OID.2.16.840.1.101.3.4.1.6) AES_128/OFB/NoPadding (2.16.840.1.101.3.4.1.3, OID.2.16.840.1.101.3.4.1.3) AES_192/CBC/PKCS5Padding (AES_192/CBC/PKCS7Padding, 2.16.840.1.101.3.4.1.22, OID.2.16.840.1.101.3.4.1.22) AES_192/CFB/NoPadding (2.16.840.1.101.3.4.1.24, OID.2.16.840.1.101.3.4.1.24) AES_192/ECB/NoPadding (2.16.840.1.101.3.4.1.21, OID.2.16.840.1.101.3.4.1.21) AES_192/GCM/NoPadding (2.16.840.1.101.3.4.1.26, OID.2.16.840.1.101.3.4.1.26) AES_192/OFB/NoPadding (2.16.840.1.101.3.4.1.23, OID.2.16.840.1.101.3.4.1.23) AES_256/CBC/PKCS5Padding (AES_256/CBC/PKCS7Padding, 2.16.840.1.101.3.4.1.42, OID.2.16.840.1.101.3.4.1.42) AES_256/CFB/NoPadding (2.16.840.1.101.3.4.1.44, OID.2.16.840.1.101.3.4.1.44) AES_256/ECB/NoPadding (2.16.840.1.101.3.4.1.41, OID.2.16.840.1.101.3.4.1.41) AES_256/GCM/NoPadding (2.16.840.1.101.3.4.1.46, OID.2.16.840.1.101.3.4.1.46) AES_256/OFB/NoPadding (2.16.840.1.101.3.4.1.43, OID.2.16.840.1.101.3.4.1.43)	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
	DESede (TripleDES)[1] DESede/CBC/PKCS5Padding (DESede/CBC/PKCS7Padding, OID.1.2.840.113549.3.7, 1.2.840.113549.3.7)[1]	—
	PBEWithHmacSHA1AndAES_128 PBEWithHmacSHA1AndAES_256 PBEWithHmacSHA224AndAES_128 PBEWithHmacSHA224AndAES_256 PBEWithHmacSHA256AndAES_128 PBEWithHmacSHA256AndAES_256 PBEWithHmacSHA384AndAES_128 PBEWithHmacSHA384AndAES_256 PBEWithHmacSHA512AndAES_128 PBEWithHmacSHA512AndAES_256	PBES2 password-based cipher
	RSA/ECB/OAEPPadding RSA/ECB/OAEPWithSHA-1andMGF1Padding (RSA/ECB/OAEPWithSHA1andMGF1Padding) RSA/ECB/OAEPWithSHA-224andMGF1Padding (RSA/ECB/OAEPWithSHA224andMGF1Padding) RSA/ECB/OAEPWithSHA-256andMGF1Padding (RSA/ECB/OAEPWithSHA256andMGF1Padding) RSA/ECB/OAEPWithSHA-384andMGF1Padding (RSA/ECB/OAEPWithSHA384andMGF1Padding) RSA/ECB/OAEPWithSHA-512andMGF1Padding (RSA/ECB/OAEPWithSHA512andMGF1Padding)	—
KDF[2]	HKDF-SHA256 HKDF-SHA384 HKDF-SHA512	HMAC-based KDF as defined in RFC 5869.[2]
KEM	ML-KEM ML-KEM-512 ML-KEM-768 ML-KEM-1024	The Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) as defined in FIPS 203. This algorithm supports keys with ML-KEM-512, ML-KEM-768, and ML-KEM-1024 parameter sets.
KeyAgreement	DH (DiffieHellman, 1.2.840.113549.1.3.1, OID.1.2.840.113549.1.3.1) ECDH	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
KeyFactory	DH (DiffieHellman, 1.2.840.113549.1.3.1, OID.1.2.840.113549.1.3.1) DSA (1.2.840.10040.4.1, OID.1.2.840.10040.4.1, 1.3.14.3.2.12)[1] EC (EllipticCurve, 1.2.840.10045.2.1, OID.1.2.840.10045.2.1) RSA (1.2.840.113549.1.1, OID.1.2.840.113549.1.1, 1.2.840.113549.1.1.1, OID.1.2.840.113549.1.1.1) RSASSA-PSS (PSS, 1.2.840.113549.1.1.10, OID.1.2.840.113549.1.1.10)	—
	ML-DSA ML-DSA-44 ML-DSA-65 ML-DSA-87	—
	ML-KEM ML-KEM-512 ML-KEM-768 ML-KEM-1024	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
KeyGenerator	AES (Rijndael, 2.16.840.1.101.3.4.1, OID.2.16.840.1.101.3.4.1) AES_128/CBC/PKCS5Padding (AES_128/CBC/PKCS7Padding, OID.2.16.840.1.101.3.4.1.2, 2.16.840.1.101.3.4.1.2) AES_128/CFB/NoPadding (OID.2.16.840.1.101.3.4.1.4, 2.16.840.1.101.3.4.1.4) AES_128/ECB/NoPadding (OID.2.16.840.1.101.3.4.1.1, 2.16.840.1.101.3.4.1.1) AES_128/GCM/NoPadding (OID.2.16.840.1.101.3.4.1.6, 2.16.840.1.101.3.4.1.6) AES_128/OFB/NoPadding (OID.2.16.840.1.101.3.4.1.3, 2.16.840.1.101.3.4.1.3) AES_192/CBC/PKCS5Padding (AES_192/CBC/PKCS7Padding, OID.2.16.840.1.101.3.4.1.22, 2.16.840.1.101.3.4.1.22) AES_192/CFB/NoPadding (OID.2.16.840.1.101.3.4.1.24, 2.16.840.1.101.3.4.1.24) AES_192/ECB/NoPadding (OID.2.16.840.1.101.3.4.1.21, 2.16.840.1.101.3.4.1.21) AES_192/GCM/NoPadding (OID.2.16.840.1.101.3.4.1.26, 2.16.840.1.101.3.4.1.26) AES_192/OFB/NoPadding (OID.2.16.840.1.101.3.4.1.23, 2.16.840.1.101.3.4.1.23) AES_256/CBC/PKCS5Padding (AES_256/CBC/PKCS7Padding, OID.2.16.840.1.101.3.4.1.42, 2.16.840.1.101.3.4.1.42) AES_256/CFB/NoPadding (OID.2.16.840.1.101.3.4.1.44, 2.16.840.1.101.3.4.1.44) AES_256/ECB/NoPadding (OID.2.16.840.1.101.3.4.1.41, 2.16.840.1.101.3.4.1.41) AES_256/GCM/NoPadding (OID.2.16.840.1.101.3.4.1.46, 2.16.840.1.101.3.4.1.46)	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
	AES_256/OFB/NoPadding (OID.2.16.840.1.101.3.4.1.43, 2.16.840.1.101.3.4.1.43)	
	HmacSHA1 (1.2.840.113549.2.7, OID.1.2.840.113549.2.7) HmacSHA224 (1.2.840.113549.2.8, OID.1.2.840.113549.2.8) HmacSHA256 (1.2.840.113549.2.9, OID.1.2.840.113549.2.9) HmacSHA384 (1.2.840.113549.2.10, OID.1.2.840.113549.2.10) HmacSHA512 (1.2.840.113549.2.11, OID.1.2.840.113549.2.11)	—
	SunTls12Prf SunTlsExtendedMasterSecret SunTlsKeyMaterial (SunTls12KeyMaterial) SunTlsRsaPremasterSecret (SunTls12RsaPremasterSecret)	These non-standard KeyGenerator algorithms are needed to provide the cryptography required by the SunJSSE provider to support TLSv1.2.
KeyPairGenerator	DH (DiffieHellman, 1.2.840.113549.1.3.1, OID.1.2.840.113549.1.3.1) EC (EllipticCurve, 1.2.840.10045.2.1, OID.1.2.840.10045.2.1) RSA (1.2.840.113549.1.1, OID.1.2.840.113549.1.1, 1.2.840.113549.1.1.1, OID.1.2.840.113549.1.1.1) RSASSA-PSS (PSS, 1.2.840.113549.1.1.10, OID.1.2.840.113549.1.1.10)	—
	ML-DSA ML-DSA-44 ML-DSA-65 ML-DSA-87	—
	ML-KEM ML-KEM-512 ML-KEM-768 ML-KEM-1024	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
Mac	HmacSHA1 (1.2.840.113549.2.7, OID.1.2.840.113549.2.7) HmacSHA224 (1.2.840.113549.2.8, OID.1.2.840.113549.2.8) HmacSHA256 (1.2.840.113549.2.9, OID.1.2.840.113549.2.9) HmacSHA384 (1.2.840.113549.2.10, OID.1.2.840.113549.2.10) HmacSHA512 (1.2.840.113549.2.11, OID.1.2.840.113549.2.11)	—
MessageDigest	SHA-1 (SHA, SHA1, 1.3.14.3.2.26, OID.1.3.14.3.2.26)	—
	SHA-224 (SHA224, 2.16.840.1.101.3.4.2.4, OID.2.16.840.1.101.3.4.2.4) SHA-256 (SHA256, 2.16.840.1.101.3.4.2.1, OID.2.16.840.1.101.3.4.2.1) SHA-384 (SHA384, 2.16.840.1.101.3.4.2.2, OID.2.16.840.1.101.3.4.2.2) SHA-512 (SHA512, 2.16.840.1.101.3.4.2.3, OID.2.16.840.1.101.3.4.2.3)	—
	SHA3-224 (2.16.840.1.101.3.4.2.7, OID.2.16.840.1.101.3.4.2.7) SHA3-256 (2.16.840.1.101.3.4.2.8, OID.2.16.840.1.101.3.4.2.8) SHA3-384 (2.16.840.1.101.3.4.2.9, OID.2.16.840.1.101.3.4.2.9) SHA3-512 (2.16.840.1.101.3.4.2.10, OID.2.16.840.1.101.3.4.2.10)	—
SecretKeyFactory	AES DESede (TripleDES)[1]	—
	PBEWithHmacSHA1AndAES_128 PBEWithHmacSHA1AndAES_256 PBEWithHmacSHA224AndAES_128 PBEWithHmacSHA224AndAES_256 PBEWithHmacSHA256AndAES_128 PBEWithHmacSHA256AndAES_256 PBEWithHmacSHA384AndAES_128 PBEWithHmacSHA384AndAES_256 PBEWithHmacSHA512AndAES_128 PBEWithHmacSHA512AndAES_256	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
	PBKDF2WithHmacSHA1 (PBKDF2WithSHA1, 1.2.840.113549.1.5.12, OID.1.2.840.113549.1.5.12) PBKDF2WithHmacSHA224 (PBKDF2WithSHA224) PBKDF2WithHmacSHA256 (PBKDF2WithSHA256) PBKDF2WithHmacSHA384 (PBKDF2WithSHA384) PBKDF2WithHmacSHA512 (PBKDF2WithSHA512)	—
SecureRandom	DRBG (SHA1PRNG, CTRDRBG, CTRDRBG128, NativePRNG, NativePRNGNonBlocking)	All aliases use the same underlying DRBG algorithm from OpenSSL
Signature	ML-DSA ML-DSA-44 ML-DSA-65 ML-DSA-87	The Module-Lattice-Based Digital Signature Algorithm (ML-DSA) as defined in FIPS 204. This algorithm supports keys with ML-DSA-44, ML-DSA-65, and ML-DSA-87 parameter sets.
	RSASSA-PSS (1.2.840.113549.1.1.10, OID.1.2.840.113549.1.1.10)	—
	NONEwithDSA (RawDSA)[13] SHA1withDSA (DSA, DSS, SHA/DSA, SHA-1/DSA, SHA1/DSA, SHAwithDSA, DSAWithSHA1, 1.2.840.10040.4.3, OID.1.2.840.10040.4.3, 1.3.14.3.2.13, OID.1.3.14.3.2.13, 1.3.14.3.2.27, OID.1.3.14.3.2.27)[13] SHA224withDSA (2.16.840.1.101.3.4.3.1, OID.2.16.840.1.101.3.4.3.1)[13] SHA256withDSA (2.16.840.1.101.3.4.3.2, OID.2.16.840.1.101.3.4.3.2)[13] SHA384withDSA (2.16.840.1.101.3.4.3.3, OID.2.16.840.1.101.3.4.3.3)[13] SHA512withDSA (2.16.840.1.101.3.4.3.4, OID.2.16.840.1.101.3.4.3.4)[13]	—
	NONEwithECDSA SHA1withECDSA (1.2.840.10045.4.1, OID.1.2.840.10045.4.1)[13] SHA224withECDSA (1.2.840.10045.4.3.1, OID.1.2.840.10045.4.3.1) SHA256withECDSA (1.2.840.10045.4.3.2, OID.1.2.840.10045.4.3.2) SHA384withECDSA (1.2.840.10045.4.3.3, OID.1.2.840.10045.4.3.3) SHA512withECDSA (1.2.840.10045.4.3.4, OID.1.2.840.10045.4.4.4)	—

Table 8-1 (Cont.) Algorithm Strings Supported by Jipher

Engine	Supported Algorithm Strings and Their Aliases	Notes
	NONEwithRSA SHA1withRSA (1.2.840.113549.1.1.5, OID.1.2.840.113549.1.1.5, 1.3.14.3.2.29, OID.1.3.14.3.2.29)[13] SHA224withRSA (1.2.840.113549.1.1.14, OID.1.2.840.113549.1.1.14) SHA256withRSA (1.2.840.113549.1.1.11, OID.1.2.840.113549.1.1.11) SHA384withRSA (1.2.840.113549.1.1.12, OID.1.2.840.113549.1.1.12) SHA512withRSA (1.2.840.113549.1.1.13, OID.1.2.840.113549.1.1.13)	RSA with PKCS1
	SHA1withRSAandMGF1[13] SHA224withRSAandMGF1 SHA256withRSAandMGF1 SHA384withRSAandMGF1 SHA512withRSAandMGF1	—

1 This algorithm is not supported if the FIPS 140 Enforcement Policy is set to FIPS_STRICT.
2 KDF service algorithms are only supported when Jipher is run on a Java version that supports the KDF API. See JEP 510
3 Only signature verification is supported if the FIPS 140 Enforcement Policy is set to FIPS.

Optionally Supported Non-FIPS 140-3 Allowed Algorithms

The algorithms corresponding to the strings in the following table use the PKCS #12 Key Derivation Function (KDF) as described in Appendix B: Deriving Keys and IVs from Passwords and Salt in *RFC 7292: PKCS #12: Personal Information Exchange Syntax v1.1*. The PKCS #12 KDF is not allowed by FIPS 140-3. Jipher provides support for these algorithms only if the system property `jipher.nonfips.supportPkcs12Kdf` is set to `true`. Support for these algorithms will be removed in a future release of Jipher.

Table 8-2 Algorithm Strings of Optionally Supported Non-FIPS 140-3 Allowed Algorithms

Engine	Algorithm Strings (and their Aliases) of Optionally Supported Non-FIPS 140-3 Allowed Algorithms
AlgorithmParameters	PBEWithSHA1AndDESede (OID.1.2.840.113549.1.12.1.3, 1.2.840.113549.1.12.1.3)[1]
Cipher	PBEWithSHA1AndDESede (1.2.840.113549.1.12.1.3, OID.1.2.840.113549.1.12.1.3)[1]
Mac	HmacPBESHA1 HmacPBESHA224 HmacPBESHA256 HmacPBESHA384 HmacPBESHA512

Table 8-2 (Cont.) Algorithm Strings of Optionally Supported Non-FIPS 140-3 Allowed Algorithms

Engine	Algorithm Strings (and their Aliases) of Optionally Supported Non-FIPS 140-3 Allowed Algorithms
SecretKeyFactory	`PBEWithSHA1AndDESede` (`OID.1.2.840.113549.1.12.1.3`, `1.2.840.113549.1.12.1.3`)[1]

[1] This algorithm is not supported if the FIPS 140 Enforcement Policy is set to `FIPS_STRICT`.

On JDK 25 and earlier, the SUN provider's PKCS12 `KeyStore` service employs HmacPBE* Mac algorithms for the integrity check when reading `KeyStore` instances from or writing them to persistent storage. If your application absolutely must load a PKCS12 `KeyStore` from or store one to persistent storage, then you can set the system property `jipher.nonfips.supportPkcs12Kdf` to `true` to configure Jipher to provide support for HmacPBE* Mac algorithms. This setting deviates from the FIPS 140-3 standard.

Keysize Restrictions

Jipher uses the following default key sizes (in bits) and enforces the following restrictions for `KeyGenerator` and `KeyPairGenerator`.

KeyGenerator

Jipher honors the system property `jdk.security.defaultKeySize`, which enables users to configure the default key size used by `KeyGenerator`. The value of this property is a list of comma-separated entries. Each entry consists of a case-insensitive algorithm name and the corresponding default key size (in decimal) separated by a colon.

Table 8-3 KeyGenerator Algorithms and Default Key Sizes

Algorithm Name	Default Key Size	Restrictions and Comments
AES	256 if permitted by the cryptographic policy (see Import Limits on Cryptographic Algorithms), 128 otherwise.	Key size must be equal to 128, 192, or 256.
AES_128/<*mode*>/<*padding*>	128	Key size must be equal to 128.
AES_192/<*mode*>/<*padding*>	192	Key size must be equal to 192.
AES_256/<*mode*>/<*padding*>	256	Key size must be equal to 256.
HmacSHA1	160	Key size must be at least 40 bits. Key sizes that are not a multiple of 8 are increased to the next multiple of 8.
HmacSHA224	224	Key size must be at least 40 bits. Key sizes that are not a multiple of 8 are increased to the next multiple of 8.
HmacSHA256	256	Key size must be at least 40 bits. Key sizes that are not a multiple of 8 are increased to the next multiple of 8.

Table 8-3 (Cont.) KeyGenerator Algorithms and Default Key Sizes

Algorithm Name	Default Key Size	Restrictions and Comments
HmacSHA384	384	Key size must be at least 40 bits. Key sizes that are not a multiple of 8 are increased to the next multiple of 8.
HmacSHA512	512	Key size must be at least 40 bits. Key sizes that are not a multiple of 8 are increased to the next multiple of 8.

KeyPairGenerator

Jipher honors the system property `jdk.security.defaultKeySize`, which enables users to configure the default key size used by `KeyPairGenerator`. The value of this property is a list of comma-separated entries. Each entry consists of a case-insensitive algorithm name and the corresponding default key size (in decimal) separated by a colon.

Table 8-4 KeyPairGenerator Algorithms and Default Key Sizes

Algorithm Name	Default Key Size	Restrictions and Comments
DiffieHellman	3072	Key size must be equal to 2048, 3072 or 4096. Algorithm parameter specification must specify an approved FFC Safe-prime group defined in SP 800-56A Rev. 3, "Appendix D: Approved ECC Curves and FFC Safe-prime Groups."
EC	256	Key size must be equal to 224, 256, 384, 521. Algorithm parameter specification must specify one the four approved ECC named curves listed in Approved ECC Named Curves and SP 800-56A Rev. 3, "Appendix D: Approved ECC Curves and FFC Safe-prime Groups" defined in RFC 8422: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier.
RSA and RSASSA-PSS	3072	Key size must be between 2,048 and 15,360 bits. The public exponent length must exceed 16 bits and cannot exceed 256 bits. If the key size exceeds 3072, then the public exponent length cannot exceed 64 bits.
ML-KEM	ML-KEM-768	Parameter set must be one of ML-KEM-512, ML-KEM-768 or ML-KEM-1024.
ML-DSA	ML-DSA-65	Parameter set must be one of ML-DSA-44, ML-DSA-65 or ML-DSA-87.

Approved ECC Named Curves

Standard for Efficient Cryptography Group (SECG) Name	NIST	OID
secp224r1	P-224	1.3.132.0.33

Standard for Efficient Cryptography Group (SECG) Name	NIST	OID
secp256r1	P-256	1.2.840.10045.3.1.7
secp384r1	P-384	1.3.132.0.34
secp521r1	P-521	1.3.132.0.35

Supported Elliptic Curve Names

Jipher supports only a fixed set of named (published) elliptic curves. These are NIST-recommended curves based on prime fields.

The following table lists the elliptic curves that are provided by Jipher.

Table 8-5 Supported Elliptic Curve Names

Elliptic Curve	Object Identifier and Aliases	Aliases
secp224r1	`1.3.132.0.33`	`P-224`, `P224`
secp256r1	`1.2.840.10045.3.1.7`	`P-256`, `P256`, `prime256v1`
secp384r1	`1.3.132.0.34`	`P-384`, `P384`
secp521r1	`1.3.132.0.35`	`P-521`, `P521`

Default Diffie-Hellman Parameters

When generating Diffie-Hellman (DH) key pairs, default DH parameters are selected based on key size. Supported key sizes are 2048, 3072, and 4096.

The default parameters are from RFC 7919: Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security.

Table 8-6 Default DH Parameters

Key Size	Default Parameter
2048	`ffdhe2048`
3072	`ffdhe3072`
4096	`ffdhe4096`

Sourcing Cryptographically Secure Random Bits

This section describes how the `com.oracle.jipher` module sources cryptographically secure random bits in accordance with NIST SP 800-90A.

In short, Jipher defers to OpenSSL's FIPS Random Bit Generator (RBG) for Jipher cryptographic operations. A `SecureRandom` supplied to `init`, `initialize`, or `newEncapsulator` is accepted for Java API compatibility but ignored as a source of random bytes.

`SecureRandom.getInstance("DRBG", "JipherJCE")` also obtains bytes from the OpenSSL FIPS RBG.

What Happens to a Supplied SecureRandom

For any of the following Java Cryptography API calls to the JipherJCE provider, the `SecureRandom` argument is ignored as a source of random bytes:

- `Cipher.init(..., SecureRandom)`
- `KeyPairGenerator.initialize(..., SecureRandom)`
- `Signature.initSign(..., SecureRandom)`
- `KeyGenerator.init(..., SecureRandom)`
- `AlgorithmParameterGenerator.init(..., SecureRandom)`
- `KeyAgreement.init(..., SecureRandom)`
- `KEM.newEncapsulator(..., SecureRandom)`

In the following example, Jipher does not use the argument `testRandom`:

```
Signature signature = Signature.getInstance("RSASSA-PSS", "JipherJCE");
...
signature.initSign(privateKey, testRandom);
```

The argument `testRandom` is ignored. Any random bytes required by the RSA signing operation come from OpenSSL. The same applies whether `testRandom` is a Jipher `SecureRandom`, a standard JDK implementation, or a deterministic test double. Do not rely on using a `SecureRandom` that produces a repeatable deterministic series of bits for application testing. The Java Cryptography API, specifies that API calls that do not accept a `SecureRandom` argument, but nonetheless require a source of random bytes, obtain randomness

> … using the `SecureRandom` implementation of the highest-priority installed provider as the source of randomness. (If none of the installed providers supply an implementation of `SecureRandom`, a system-provided source of randomness will be used.)

The JipherJCE provider deviates from this specified behavior. When an API call implemented by JipherJCE requires random bytes and no `SecureRandom` instance is supplied, it uses an internal OpenSSL Deterministic Random Bit Generator (DRBG) instead of a `SecureRandom` provided by the highest-priority installed provider.

Motivation for Using OpenSSL's FIPS Random Bit Generator

There are two reasons for Jipher using an OpenSSL DRBG instead of a `SecureRandom` provided by the highest-priority installed provider:

- FIPS compliance issues:
  - Approved random bit generators (RBGs) are defined in SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators.
  - The required security strength for RBGs is defined in several NIST documents, including NIST SP 800-133 Rev. 2: Recommendation for Cryptographic Key Generation.

  Jipher must guarantee that it only uses a FIPS-approved RBG that provides sufficient security strength.

- Performance issues:
  - Cryptographic functionality provided by JipherJCE classes is, behind the scenes, delivered by the OpenSSL FIPS module.
  - A performance benefit is derived by having OpenSSL use the FIPS module's own native DRBG for all internal operations.

  Configuring an OpenSSL algorithm instance to use a specific Java `SecureRandom` instance as its source of randomness is technically possible. However, we have chosen not to support it at this time, pending a better understanding of the runtime performance implications of developing and maintaining the required bridge.

Where Jipher SecureRandom Bytes Come From

Jipher registers one `SecureRandom` algorithm: `DRBG`. Its SPI delegates `nextBytes` and `generateSeed` to Jipher's `Rand.generate`, which calls OpenSSL's `randBytes` at a requested strength of 256 bits. This means that the following code fills `bytes` using the OpenSSL random subsystem in Jipher's library context:

```
SecureRandom random = SecureRandom.getInstance("DRBG", "JipherJCE");
random.nextBytes(bytes);
```

Jipher's `DRBG` algorithm is not a Java-side DRBG with Java-visible state.

Calling `random.setSeed(...)` is accepted but has no effect. Jipher's `SecureRandomSpi.engineSetSeed` is intentionally a no-op. In particular, seeding a Jipher `SecureRandom` cannot make either that instance or an OpenSSL-backed Jipher operation reproducible.

OpenSSL DRBG

OpenSSL internally manages a DRBG hierarchy used by operations requiring a source of randomness. The configured DRBG algorithm will be one of the following, configured with 256-bit security strength:

- CTR-DRBG
- HASH-DRBG
- HMAC-DRBG

OpenSSL uses a per-thread DRBG seeded from a global DRBG. Entropy for the global DRBG is obtained from the operating system and both global and per-thread DRBGs are automatically reseeded according to configured call-count and time intervals. These are OpenSSL configuration settings, not per-instance Java `SecureRandom` settings.

See the OpenSSL RAND and OSSL_PROVIDER-FIPS man pages for more information.

Unsupported Java DRBG Operations

Jipher does not bridge Java's post-JDK-8 parameterized `SecureRandom` APIs to OpenSSL.

The following table lists Java DRBG operations that aren't supported by Jipher and describes how Jipher behaves when they are called.

Table 8-7 Java DRBG Operations Not Supported by Jipher

Operation	Jipher Behavior
`SecureRandom.getInstance("DRBG", params, "JipherJCE")`	Throws `NoSuchAlgorithmException`. Jipher's SPI cannot be constructed with parameters.
`random.getParameters()`	Returns `null`.
`random.nextBytes(bytes, params)`	Throws `UnsupportedOperationException`.
`random.reseed()`	Throws `UnsupportedOperationException`.
`random.reseed(params)`	Throws `UnsupportedOperationException`.

Calling `SecureRandom.getInstance(String algorithm, SecureRandomParameters params)` without naming Jipher may select another installed provider that supports the requested parameters. That instance remains a Java-side random source. Passing it to a Jipher service still does not alter the OpenSSL source used by that service.

Jipher Development and Test Guidance with RBGs

- Use Jipher's `SecureRandom` when an application directly needs random bytes produced by Jipher's OpenSSL-backed random subsystem.
- It is preferable to use JCA initialization APIs that do not specify a `SecureRandom` when possible instead of passing `null` or a subsequently ignored object.
- Do not use a deterministic `SecureRandom` injection to make a Jipher operation repeatable. Assert externally observable properties instead, or use a dedicated Jipher/OpenSSL test facility if one is provided.