Java Cryptography: Tools and Techniques

David Hook
Jon Eaves

Java Cryptography: Tools and Techniques

David Hook and Jon Eaves

This book is available at https://leanpub.com/javacryptotoolsandtech

This version was published on 2025-07-16

Contents

Acknowledgements

The authors wish to acknowledge Peter Grant, Stephen Farrugia, and Megan Woods for reviewing the text.

The authors also wish to express their gratitude and acknowledge Christoffer Sawicki, Sree Harsha Totakura, and Alexander Railean for reporting errors and ambiguities in the published text. Such feedback has helped improve the book for everyone.

David also wishes to acknowledge support of his wife, Janine, and son, Lewis, as well as the continued interest of Hamlet the cat (the original little cryptographer in a fur coat) and that of Ophelia the dog (perhaps not a cryptographer, but certainly very enthusiastic about whatever it is she actually thinks she is doing). You have all been a constant source of inspiration and support, thanks!

Jon would like to acknowledge the support of his wife Jo for supporting him through not only this book, but in all things. Additionally Jon wants to acknowledge the amazing friendship, support and mentorship of David, for without his diligent stewardship of Bouncy Castle, the crypto community would be significantly poorer.

Introduction

"It's not wise to violate the rules until you know how to observe them."

– T. S. Eliot

Our previous effort at writing a book about Java, Bouncy Castle, and how it touches on the world of Cryptography is now well over ten years old.

A lot has changed since then, versions of the Bouncy Castle APIs for Java have gone through two FIPS 140-2 certifications, seeing the project transferring from a part time pursuit to something which now has a small group working on it and supporting it full time. The API set has expanded with the addition of things like Time Stamp Protocol, TLS, and even, more recently, a JSSE provider. Java 1.5, which was the bleeding edge at the time, has reached end-of-life, and workhorse algorithms, like SHA-1 for example, have seen both research and technology overtake them. Even many of the algorithms we currently hold as secure now have seen their key sizes dramatically increase and even with that may still end up being rendered obsolete by the arrival of Quantum Computers.

So there has been a bit of catching up to do, and this book is an attempt to do just that. In addition to dealing with updates and enhancements in the Bouncy Castle APIs, we have also tried to provide some of the original source material and standards these APIs are based on. We have done this because it is important to have a grasp of the background material to the implementations as well. The biggest problem most of us face "in the field" is producing messages and communicating with other already deployed systems and, in the light of that, having some understanding of what is supposed to be implemented and what the options, or points of difference might be, can be a big help when you are faced with the situation of working out what some black box system you have to deal with actually expects.

A Word about the Book

If you are really new at this, and starting out with this book, take some time to read Appendix A first. It describes a data description language called ASN.1 which has its origins back in the 1980s - an interesting time to be in computing, not the least because we used to fly into a panic if a message was a byte longer than it needed to be. ASN.1 reflects this "panic" as well, and is used to define encodings for a wide variety of structures that are used in cryptographic protocols. As a result of this, there are references to ASN.1 throughout the text and having some idea what it is actually about will make things a lot easier to follow.

A Word About the Software Itself

The Bouncy Castle project has been going since 1999 with formal releases appearing in the year 2000. While the setting up of the Bouncy Castle charity and Crypto Workshop has meant that people can finally work on the APIs on a full-time basis, even with that, the one thing our experiences have taught us is that errors are unavoidable. These can be errors in what we do, errors in the documents describing the algorithms, errors in the specification of protocols, and errors in interpretation, by us, or other users. In some cases the "error" might be totally unforeseen as the discovery of an attack may reduce what was previously seen as a safe algorithm, or protocol, to something more akin to the level of security you would expect from spray painting your secrets on the side of a Zeppelin and then flying it over a national sporting event. This kind of thing requires constant vigilance.

All these things can affect both the security and the compatibility of the libraries we provide. If you are going to work in this space please do not deploy things without at least some of idea of how you will upgrade, because chances are you will need to! While it can provide some short term relief to blame NIST, the IETF, some corporate, or a bunch of "long hairs on the Internet" for a security issue, it will not actually help you, your users, or their data, if you cannot deploy a patch for a vulnerability. The need to patch is the rule, not the exception.

Finally, consider getting a support contract through Keyfactor (https://www.keyfactor.com/platform/bouncy-castle-support/). It pays to remember where you expect your patches to be coming from so support us and we will always be here to support you! It costs real money and real resources to keep this project going now, and as much as we can try and make this work accessible, the reality is there will still be times when it is cheaper to have the direct connection a support contract gives you with the Bouncy Castle developers than to find yourself typing an email starting with "My team has spent the last month trying to…".

And Finally

We hope this book is not interpreted as a reason not to think for yourself, you still need to do so.

We do hope that it is at least some help in finding out how to solve your problems though. If it is also of any use, the example code used in the book is also available in:
https://www.bouncycastle.org/java-crypto-tools-src.zip.

Lastly, one of the issues with trying to explain material like this, especially when you have been doing it for a while, is that it is easy to dwell on obvious things thinking other people will not understand them, and take for granted things that others may find complex believing the facts of the case are obvious. To that end please feel free to get in touch with us if you have any questions or suggestions about this book. While we cannot promise to act on all suggestions we will try to take them into account, and as for questions, there are no stupid ones, just a couple of writers trying to find a way to fix badly worded answers.

Enjoy!

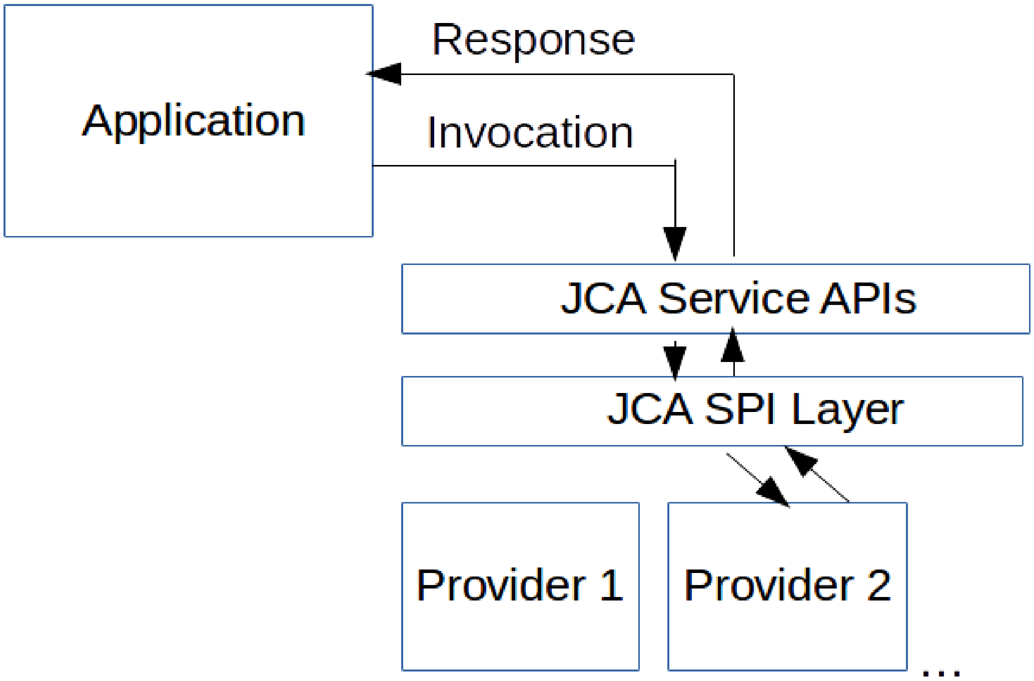
Chapter 1: Getting Started, an Overview

This chapter introduces the architecture of the cryptography APIs used in Java as well as the architecture of the Bouncy Castle APIs for CMS/SMIME/OpenPGP that are built on top of the cryptography APIs. Basic installation instructions are given, as well as a discussion of the differences between the BC FIPS certified APIs and the regular distribution. Some cautionary advice about the use of random numbers that applies across all the APIs will also be given and we will look at what "bits of security" means for applications and algorithms.

The Java Provider Architecture

The Provider Architecture associated with the Java Cryptography Architecture (JCA) provides the foundation for a range of security services in the JVM. These now include the Java Secure Socket Extension (JSSE) which provides access to Secure Socket Layer (SSL) and Transport Layer Security (TLS) implementations, the Java Generic Security Services (JGSS) APIs, and the Simple Authentication and Security Layer (SASL) APIs. You will also see references to the Java Cryptography Extension (JCE), as originally (pre-Java 1.4) it was bundled separately from the JCA, however with a few exceptions it is generally now better, for the most part, to think of it as part of the JCA. One of the situations where this distinction between the JCA and the JCE is meaningful is in the case of provider signing, which we will look at further on.

The foundation layer is specified in the provider architecture by having a set of service classes which are exposed to developers in order to give a uniform interface to interact with installed providers of services. The services classes provide their functionality by invoking methods defined in various service provider interface (SPI) which are implemented by objects defined in the actual providers that either come with the JVM or are made available by other vendors such as the Legion of the Bouncy Castle.



A block view of a Provider invoked through the Java Cryptography Architecture

The first thing the provider architecture provides is a separation between what an application developer will normally call, for example methods belonging to `java.security.Signature` or `javax.crypto.Cipher`, and what is actually implemented by a provider of cryptographic services, such as Bouncy Castle. For example, from an developer's point of view creating a cipher will look like:

```
Cipher c = Cipher.getInstance("Blowfish/CBC/PKCS5Padding");
```

But an implementor of the Blowfish cipher service will actually extend the `javax.crypto.CipherSpi` class and it will be this Service Provider Interface (SPI) object that is wrapped inside the `Cipher` object and provides the actual working parts of the cipher concerned.

The second thing the provider architecture provides is the ability to allow the deployer of the application to determine which vendor implementation of a particular algorithm is used. This can be done by relying on provider precedence which is the case when the `getInstance()` method for a particular service is invoked as in the example above. You can also specify that the SPI contained by the cryptographic services object you create using `getInstance()` must come from a particular provider by either specifying the provider's name:

```
Cipher c = Cipher.getInstance("Blowfish/CBC/PKCS5Padding", "BC");
```

or, from Java 1.5 and onward, by passing in an actual provider object:

```
Cipher c = Cipher.getInstance(
                "Blowfish/CBC/PKCS5Padding", new BouncyCastleProvider());
```

How Provider Precedence Works

In the situation where no provider is explicitly specified, the JCA relies on provider precedence to determine which SPI to create.

You can see an example of this by using the following example:

```
/**
 * A simple application demonstrating the effect of provider precedence on
 * what is returned by a JVM.
 */
public class PrecedenceDemo
{
    public static void main(String[] args)
        throws Exception
    {
        // adds BC to the end of the precedence list
        Security.addProvider(new BouncyCastleProvider());

        System.out.println(MessageDigest.getInstance("SHA1")
                                        .getProvider().getName());

        System.out.println(MessageDigest.getInstance("SHA1", "BC")
                                        .getProvider().getName());
    }
}
```

If you compile and run the example, assuming a regular Java installation you should see the following output:

```
SUN
BC
```

As the comment in the code suggests the `addProvider()` method adds the provider to the end of the precedence list, so the default SHA-1 implementation is taken from the `SUN` provider, rather than from Bouncy Castle.

When a JVM starts up the precedence of the installed providers is determined by a precedence table in the `java.security` file that comes with your JVM. If you are looking at Java 1.8 and earlier you can find the `java.security` file in `$JAVA_HOME/jre/lib/security`, for Java 1.9 onwards the file is in `$JAVA_HOME/conf/security`.

What Does Provider Signing Mean?

You will probably have also seen references to signed providers. Many JVMs require that any provider of encryption or key exchange services is signed by a JCE signing certificate. Some, such as those produced by the OpenJDK project do not, but those produced by organisations like Oracle and IBM still do.

In the event a provider has not been signed correctly the JCE in a JVM which requires provider signing will reject any use of the provider for JCE functions (note this does not include signatures and message digests). For the most part this isn't a concern as it is unusual to be building your own provider, but in the event that you are, you will need to apply for a JCE signing certificate. In our case, at Bouncy Castle, we have already applied for a signing certificate and use one issued to us by Oracle, which enables both the regular Bouncy Castle provider and the Bouncy Castle FIPS provider to work with a standard JVM.

Note: the JCE signing certificate is completely different from a regular code signing certificate. Where one is required, you need to follow the procedure described in "Get a Code-Signing Certificate" in the "How to Implement a Provider in the Java Cryptography Architecture" document that comes with the documentation included with your Java distribution to obtain one.

The Jurisdiction Policy Files

As with the need for a signed provider, some JVMs, such as those from Oracle and IBM, also have their JCE implementation constrained in terms of what algorithms and key sizes are allowed. These restrictions are enforced by the Jurisdiction Policy files. Assuming you need greater key sizes than allowed by these restrictions, you will probably need to install the unrestricted policy files if you are running in a JVM that is pre-Java 1.9 (in Java 1.9 the default for the jurisdiction policy files became unlimited).

You will normally find the unrestricted policy files at the same place from where you downloaded the JDK/JRE. Usually it is on a link entitled "Unlimited Strength Jurisdiction Policy Files". The download is usually a ZIP file, and providing it is legal for you to do so, you should download the ZIP and extract it. You will find a couple of jar files and a README in the ZIP. Follow the instructions and you should be ready to proceed. If you installed your JDK/JRE using a package manager, such as apt, you will normally find there is also a package containing the unrestricted policy files that you can install as well.

It is important that you install the unrestricted policy files both in your development JVM and any JVM associated with what you are doing in production. On Linux, or some other Unix variant, this usually means you will require root access, or access to someone who has root access to install the policy files. This is especially important on Windows where the regular JDK install usually installs two separate JVMs - one for the JDK and one for use as a general Java runtime. In this situation, you need to make sure the policy files have been installed in both.

Post Java 1.5 the simplest way to test for the presence of the policy files is using the Java utility `$JAVA_HOME/bin/jrunscript`:

```
jrunscript -e 'print (javax.crypto.Cipher.getMaxAllowedKeyLength("AES") >= 256);'
```

If you are trying to do this on Java 1.5 or earlier you need to write a small application. The following should generally do the job:

```
/**
 * A simple application to check for unrestricted policy files.
 */
public class PolicyFileCheck
{
    public static void main(String[] args)
        throws NoSuchAlgorithmException
    {
        try
        {
            Cipher cipher = Cipher.getInstance("Blowfish/ECB/NoPadding", "BC");

            cipher.init(Cipher.ENCRYPT_MODE,
                    new SecretKeySpec(new byte[32], "Blowfish"));

            System.out.print("true");
        }
        catch (NoSuchAlgorithmException e)
        {
            throw e;
        }
        catch (Exception e)
        {
            System.out.print("false");
        }
    }
}
```

Both the above methods are designed to display a single “true” or “false” depending on whether or not the unrestricted policy files are present, although the second one will also throw a NoSuchAlgorithmException if the Blowfish algorithm is not present. In the unlikely event Blowfish is not present, you will need to change the algorithm choice to something else which is available and representative of an algorithm you actually plan to use.

The Standard Provider Set

Java 1.4 had five providers by default: the standard JCA one, a JSSE provider, an RSA signature provider, a provider of JCE SPIs (so ciphers, MACs, and key agreement algorithms), and a provider for the JGSS.

As of Java 1.9 there are now twelve. All the previous ones, plus providers for Elliptic Curve, XML signatures, LDAP, Simple Authentication and Security Layer (SASL - there are two of these), a PC/SC provider for use with Smart Cards, and a PKCS11 provider for use with Hardware Security Modules (HSMs).

What you are likely to see will depend on both which version of the JVM you are looking at and which distribution. We will see in the next section that it is possible to get a clearer picture of what is available by default with a little bit of coding.

Examining Providers and their Capabilities

The available providers are also accessible using the standard Java API. The following program displays a list of the providers installed in the Java runtime giving their names and "info" strings as output.

```java
/**
 * Simple application to list installed providers and their available info.
 */
public class ListProviders
{
    public static void main(String[] args)
    {
        Provider[] installedProvs = Security.getProviders();

        for (int i = 0; i != installedProvs.length; i++)
        {
            System.out.print(installedProvs[i].getName());
            System.out.print(": ");
            System.out.print(installedProvs[i].getInfo());
            System.out.println();
        }
    }
}
```

It is also possible to get a "human readable" list of what algorithms a provider implements by displaying the lookup table that each provider carries within it. The following program prints out the available algorithms, as well as any aliases that might be available and which algorithms they resolve to.

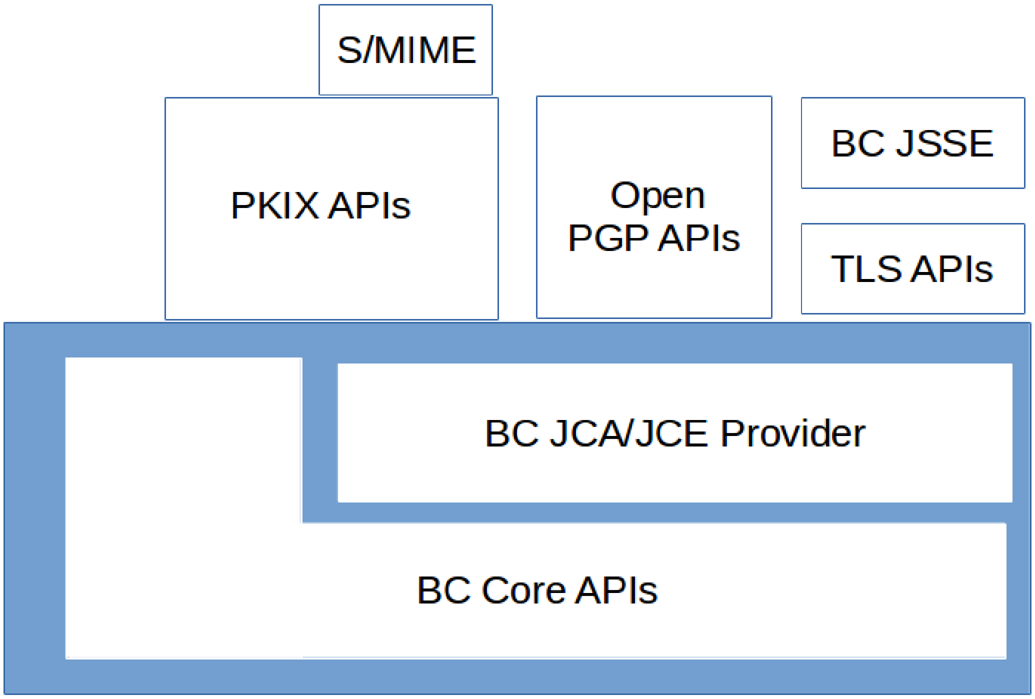
```java
/**
 * Simple application to list the capabilities (including alias names)
 * of a provider.
 * <pre>
 *     usage: chapter1.ListProviderCapabilites provider_name
 * </pre>
 */
public class ListProviderCapabilities
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println(
                "usage: chapter1.ListProviderCapabilites provider_name");
            System.exit(1);
        }

        Provider provider = Security.getProvider(args[0]);

        if (provider != null)
        {
            for (Iterator it = provider.keySet().iterator(); it.hasNext();)
            {
                String entry = (String)it.next();
                boolean isAlias = false;

                // an alias entry refers to another entry
                if (entry.startsWith("Alg.Alias"))
                {
                    isAlias = true;
                    entry = entry.substring("Alg.Alias".length() + 1);
                }

                String serviceName = entry.substring(
                                            0, entry.indexOf('.'));
                String name = entry.substring(serviceName.length() + 1);

                if (isAlias)
                {
                    System.out.print(serviceName + ": " + name);
                    System.out.println(" (alias for "
                              + provider.get("Alg.Alias." + entry) + ")");
                }
                else
                {
                    System.out.println(serviceName + ": " + name);
                }
            }
        }
        else
        {
            System.err.println("provider " + args[0] + " not found");
            System.exit(1);
        }
    }
}
```

Architecture of the Bouncy Castle APIs

Bouncy Castle was originally developed as a project to support encryption on the J2ME platform. After putting together a basic API for doing this, it became obvious that it was possible to use the components to construct a higher level API that would support the provider architecture described in the JCA and the JCE.

As things progressed, it was also obvious that still higher level APIs such as CMS, S/MIME, and X.509 certificate generation needed to be supported.

There were two attempts at the higher level APIs. The first attempt was closely coupled to the BC provider and the JCA. The second attempt, seen from BC 1.46 onward, introduced interfaces for providing the “gross operations” required for the different APIs. Together with the basic provider/low-level split, this resulted in the arrangement of the different APIs looking something like the following.



The high-level layout of the Bouncy Castle APIs (FIPS and General)

The layered approach allowed for two things to become possible. Programs could be written using the light-weight API so the BC provider was no longer mandatory and, while the diagram does not suggest it, other JCA/JCE providers from different vendors could be used as well, including multiple providers if necessary. Consequently in many cases, the BC APIs should work equally well with a HSM as they do with the BC Provider.

The BC Core API

The main part of the BC core API is in the package `org.bouncycastle.crypto` and traditionally this has been described as the light-weight API. While it may seem strange to describe it as a light-weight library these days (frankly, after 17 years, while it is not over-weight, it has become rather large) the library was originally conceived as something that could be easily sub-setted and run in small devices. In the early days, for the most part, even method arguments were not validated. On most devices the space was not there to make such "luxuries" possible.

The light-weight library is designed to provide a kind of "algebra" which allows different ciphers and digests to be mixed and matched with different encodings, modes, MACs, and signature algorithms. It should be pointed out that the manner in which things can be mixed is more flexible than is required to prevent combinations that are not so secure. If you are going to use the light-weight API it is a good idea to either have a very good idea of what you are doing, or to be implementing an algorithm/mode combination which has been sourced from someone reliable.

The BC core APIs also include an ASN.1 library which provide a range of low-level classes for supporting different protocol elements, covering everything from key encoding, to time-stamps and digital signatures. The classes in the ASN.1 library are set up to be usable on any platform the BC light-weight cryptography API can be used on.

The BC Provider

One thing the BC light-weight API does provide is enough building blocks to construct a JCA/JCE provider, as a result the BC Provider is constructed from the light-weight classes. This has had the effect of skewing some parts of the light-weight API towards a JCA view of the world, but it has meant that the actual algorithm support for Bouncy Castle remains concentrated in a few packages, with the majority of the BC Provider really just providing infrastructure to allow the light-weight classes to be presented via the Service Provider Interface (SPI) interface that is defined in the core JCA and the JCE with the BC ASN.1 library being used for describing any protocol components, such as key encodings and algorithm parameters that need to be generated for the JCA.

The Supporting APIs

The BC cryptography APIs include support for dealing with CMS, PKCS#10, PKCS#12, S/MIME, OpenPGP, DTLS, TLS, OCSP, TSP, CMP, CRMF, DVCS, DANE, EST, PEM/OpenSSL, X.509 certificates, and X.509 attribute certificates. Other than the API for OpenPGP these APIs are for protocols built on ASN.1 and they provide the higher level functionality that is not met by the packet-level objects that the BC core ASN.1 library provides.

The APIs are built on a set of operator interfaces and are designed to be JCA/JCE agnostic, in the sense that it is possible to get them working with any provider and that they can also be used with Java cryptography APIs that do not use the JCE or the JCE. The APIs come with operators implemented for the JCA/JCE and also the BC lightweight API. The different operator packages can be distinguished by the package names associated with them, the JCA/JCE operators are all in packages that have names that end in ".jcajce" and the BC lightweight packages are all in packages that have names that end in ".bc".

The supporting APIs are divided into 4 jars files: * the "bcpkix" set, which contains all things related to certificates, CMS, and TSP * the "bcpg" set which contains the OpenPGP APIs * the "bcmail" set which contains the S/MIME APIs. * the "bctls" set which contains the TLS APIs and the BCJSSE provider.

The Bouncy Castle FIPS distribution

The Bouncy Castle FIPS distributions have all been built on code which has been certified to FIPS 140-2 level 1 and are provided to allow people to develop applications for an environment either requiring the use of FIPS compliance or actual FIPS compliance. They have also been used as the basis for other FIPS certifications as well as providing a basis to work on certifications such as Common Criteria.

⚠ Just using the FIPS API is not in itself enough to ensure FIPS compliance. The FIPS API must be used in accordance with the FIPS security policy document that is associated with the distribution. NIST make copies of security policies they have accepted publicly available on their website if you need a definitive source for the document.

How the FIPS distribution is different

There are two major differences between the FIPS distribution and the regular BC distribution.

The first one is the equivalent to the light-weight API in the regular BC library is hidden. The reason for this is that the FIPS standard requires the low-level support to be locked down so that access by a developer is only along pathways known to be safe, so the general BC approach of providing the light-weight API that can be assembled at will to perform higher-level functions is too open. There is still a low-level API which has a similar relationship to the BC FIPS provider, but it is considerably more rigid than the one provided in the general BC APIs.

The second difference is that the distribution is capable of running in two modes: an approved-only mode where only FIPS approved algorithms are available and a non-approved mode where the full algorithm set implemented by the provider is available.

There is also a third, lesser difference: not everything implemented in the general distribution of Bouncy Castle is available in the FIPS distribution. Partly this is due to an original design decision to only try supporting algorithms which were referred to in at least one current IETF standard in the FIPS distribution. The other consideration is that it is substantially harder and more expensive to add anything to the FIPS distribution due to the certification requirements. At any rate, if you find anything missing which you think should be included, by all means let us know, but please understand patience will be required!

The BC FIPS Providers

There are 3 variations of the BCFIPS provider, two of them due to Android. The primary distribution is BCFIPS which is under the `org.bouncycastle` package and is generally certified for particular versions of the JVM. The next distribution is under the `org.stripycastle` package and is normally pre-installed on an Android device which itself has been certified - this one is accessed using the provider name SCFIPS. Finally there is an Android version which can be included in a DEX file as part of an application. This last version follows Roberto Tyley's well established `org.spongycastle` convention. In this case the provider is referred to by the name SCFDEX.

At the current time, where the use of the SCFDEX provider is made, a DEX application would need to be explicitly certified for a particular platform to be FIPS compliant. Primarily SCFDEX has been used by developers who need to be able to say they are running off a FIPS certified code base, rather than having to be able to say the resulting code is FIPS certified.

Bouncy Castle FIPS provider names and provider classes

Name	Provider Class	Platform
BCFIPS	org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider	General JVM
SCFIPS	org.stripycastle.jcajce.provider.StripyCastleFipsProvider	Android
SCFDEX	org.spongycastle.jcajce.provider.SpongyCastleFipsProvider	Android

The BC FIPS Supporting APIs

The usual support APIs are available for the FIPS distribution, although at the time of writing for the most part they rely exclusively on the use of the provider. While the usual convention of "jcajce" packages applies for implementations of the different operator interfaces in the APIs there are no equivalent "bc" package implementations using the low-level APIs as there are in the regular API set.

For BCFIPS the 4 jars start with bcpkix-fips (the PKIX API set), bcmail-fips (the S/MIME API set built on PKIX), bcpg-fips (the OpenPGP API set), and bctls-fips (the TLS API set).

For SCFIPS the 4 jars start with scpkix-fips (the PKIX API set), scmail-fips (the S/MIME API set built on PKIX), scpg-fips (the OpenPGP API set), and sctls-fips (the TLS API set).

For SCFDEX the 4 jars start with scpkix-fipsdx (the PKIX API set), scmail-fipsdx (the S/MIME API set built on PKIX), scpg-fips (the OpenPGP API set), and sctls-fipsdx (the TLS API set).

It should be noted that another difference from the general APIs in this case is that where the FIPS provider has been configured in approved-only mode, not all the algorithms that are otherwise supported by the supporting APIs are available for use.

Installing Bouncy Castle

There are two ways to install a cryptography provider into a JVM. You can add it yourself at runtime, either by actually adding the provider, or by invoking it on an "as needed" basis by passing the provider object to the `getInstance()` methods of the services you are trying to create. Which method you choose to use really depends on your circumstances, but you are less likely to have issues if a provider is installed by configuring the Java runtime.

Note that while the following instructions, where they are provider specific, are given in the context of the BC provider, the same instructions, other than those around provider configuration, apply to the BC FIPS providers, and most other providers, as well.

Static Installation

How you install a provider statically depends on whether you are dealing with a JVM that is pre-Java 1.9 or not.

For Java 1.4 to Java 1.8 you can add the provider by inserting its class name into the list in the `java.security` file in `$JAVA_HOME/jre/lib/security`. The provider list is a succession of entries of the form “security.provider.n” where n is the precedence number for the provider, with 1 giving the provider the highest priority. To add Bouncy Castle in this case you need to add a line of the form:

```
security.provider.N=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Where N represents the precedence you want the BC provider to have. Make sure you adjust the values for the other providers if you do not add the BC provider to the end of the list. After that you need to make sure the provider is on the class path, preferably in the `$JAVA_HOME/jre/lib/ext` directory so that it will always be loaded by a class loader the JCA trusts.

For Java 1.9 onwards the list is in the file `$JAVA_HOME/conf/security`. At the moment the BC providers do not support the `META-INF/service` feature of jar files, so the provider needs to be added in a similar way to before, but by using the name of the provider, either “BC” or “BCFIPS”, to the list of providers.

Java 1.9 no longer supports `$JAVA_HOME/jre/lib/ext` either. In the case of Java 1.9 you just need to make sure BC is included on the class path.

Runtime Installation

There are two ways of making use of a provider at runtime.

The first method is to use `java.security.Security` - for the BC provider this looks like:

```
java.security.Security.addProvider(
            new org.bouncycastle.jce.provider.BouncyCastleProvider());
```

After the above statement executes the situation in the JVM is almost equivalent to what will happen if the provider is statically installed. Services such as message digests can be accessed by using `getInstance()` methods using the provider’s name.

With the second method, the provider object is passed into the service being requested instead, for example, to create a Cipher for Blowfish using the BC provider, you can write:

```
Cipher c = Cipher.getInstance("Blowfish/ECB/NoPadding",
                                          new BouncyCastleProvider());
```

Provider configuration

There are three ways of configuring providers:

- using system properties
- passing a string to the provider’s constructor
- calling some configuration object

The different Bouncy Castle providers, BC, BCFIPS, and BCJSSE make use of all of these methods.

Configuring the BC Provider

The BC Provider supports three system properties:

org.bouncycastle.asn1.allow_unsafe_integer
Which allows for extended ASN.1 integers to be accepted. Extended in this case means encodings where the sign bit has been propagated through entire bytes.

org.bouncycastle.ec.disable_mqv
Which removes the EC MQV implementation from the provider and light-weight API.

org.bouncycastle.dsa.FIPS186-2for1024bits
This is a legacy property to allow 1024 bit DSA keys to be generated using the algorithm in FIPS 186-2.

Configuring the BCFIPS Provider

The BCFIPS provider supports configuration via a string constructor argument as well as system properties.

The provider can be configured to create a specific form of DRBG and/or to provide an entropy pool for DRBGs on construction. For example to configure the entropy pool using the `java.security` file entry for BCFIPS you would write the entry as follows:

```
org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider C:HYBRID;ENABLE{All};
```

The `HYBRID` command is the keyword causing the entropy pool construction to take place. This configuration string can also be passed at runtime by calling the `BouncyCastleFipsProvider` constructor with the configuration string as an argument, as in:

```
new BouncyCastleFipsProvider("C:HYBRID;ENABLE{All};")
```

The BCFIPS provider also supports a number of system properties, including org.bouncycastle.ec.disable_mqv and org.bouncycastle.dsa.FIPS186-2for1024bits. Full details on these and on other configuration can be found in the document “BC-FJA (Bouncy Castle FIPS Java API) User Guide” [38].

Configuring the BCJSSE Provider

Like the BCFIPS provider the BCJSSE provider also supports configuration using a string constructor argument and system properties. Full details on these are in the document “Java (D)TLS API and JSSE Provider User Guide” [39], however as it is fairly common we will briefly look at how to put the BCJSSE into FIPS mode and use BCFIPS as the sole provider. This is also done using a string argument to the constructor, so for the static install case, the assignment in `java.security` would appear as:

```
org.bouncycastle.jsse.provider.BouncyCastleJsseProvider fips:BCFIPS
```

Note in this case there are two separate settings being used. The first, `fips`, turns on FIPS mode in the BCJSSE, meaning that only FIPS compliant cipher suites and parameters will be used. The second, `BCFIPS`, says that only the BCFIPS provider is to be used for providing cryptographic services to the BCJSSE. You would use this configuration where you are looking for FIPS compliance using the BCFIPS provider.

Basic Troubleshooting

Having configured the provider, set up the jurisdiction policy files, and selected a method of installing the provider you should be ready to use the Bouncy Castle provider. However, the following section provides some frequently identified issues with their resolution.

Provider not properly installed

If you are accessing services on a provider by name, you might see something like the following (in this case the provider is called "BC"):

```
java.security.NoSuchProviderException: No such provider: BC
        at javax.crypto.Cipher.getInstance(Cipher.java:593)
```

In this case the provider has not been properly installed. If you were expecting it to be statically installed you need to check the java.security file, otherwise you need to make sure the code performing the runtime installation you were expecting to do will execute before the attempt to invoke a service on the provider is made.

Provider not properly signed

As mentioned earlier, if the jar containing the provider is not properly signed, the JCE will reject it when an attempt is made to use JCE functionality. If you see:

```
java.lang.SecurityException: JCE cannot authenticate the provider BC
        at javax.crypto.Cipher.getInstance(Cipher.java:657)
        at javax.crypto.Cipher.getInstance(Cipher.java:596)
```

The provider jar is either not signed, or has somehow ended up on a class loader that the JCE does not trust. If this occurs you need to check the provider jar is signed and, possibly, the class path to make sure the provider jar is appearing in a location where the system class loader is likely to trust it.

Issues with the JCE policy files

If you see something like:

```
java.security.InvalidKeyException: Illegal key size or default parameters
        at javax.crypto.Cipher.checkCryptoPerm(Cipher.java:1026)
        at javax.crypto.Cipher.implInit(Cipher.java:801)
        at javax.crypto.Cipher.chooseProvider(Cipher.java:864)
        at javax.crypto.Cipher.init(Cipher.java:1249)
        at javax.crypto.Cipher.init(Cipher.java:1186)
```

or an exception stack trace starting with:

```
java.lang.SecurityException: Unsupported keysize or algorithm parameters
```

If you are running on a pre-Java 1.8 JVM, this normally means the unrestricted jurisdiction policy files for the JCE have not been installed correctly for the JVM that the application is running on and you should download and install the policy files. If you are using Java 1.8 or later the problem will be due to the value of the `crypto.policy` property in the lib/security/java.security file. The value needs to be set to “unlimited” if you wish to make full use of the BC provider. From Java 1.8 Update 161 the default for `crypto.policy` is “unlimited”, but earlier versions are “limited”. In the case of Java 1.9 and later the file to check is conf/security/java.security and the default value for `crypto.policy` is “unlimited”.

The Provider Appears to Hang

If this happens, you probably have a case of entropy exhaustion, please see the next section headed “A Word About Entropy” for details.

A Word About Entropy

Entropy generation is an issue for software-only providers such as Bouncy Castle, the main reason being the provider has to rely on an external source of entropy as it is unable to generate entropy otherwise.

With Bouncy Castle, both the BC and BCFIPS provider will use the JVM’s entropy source unless another one is provided. The data then gets filtered through a DRBG (Deterministic Random Bit Generator).

What the JVM is using as an entropy source will vary, on Linux for example, it is normally set to “/dev/random” which may block. Usually installing “rng-tools” or the nearest equivalent will deal with this as it will also expose any underlying hardware supporting RNG generation to be used for seeding “/dev/random”. With some virtual environments hardware RNG may never be available, in that case it is important to find other ways of making entropy available to your JVM. Ways of doing this will vary with the environment you are using.

Fetching a SecureRandom from the Provider

The BC and BCFIPS providers contain two SecureRandom services.

The `DEFAULT` service which can be accessed by:

```
SecureRandom random = SecureRandom.getInstance("DEFAULT", "BC");
```

The DEFAULT service is suitable for generating keys, IVs, and random nonces. Internally it is in prediction resistant mode, which reseeds before it processes each request. This ensures that the underlying DRBG (by default based on SHA-512) is producing the noisiest output it can.

The second service is the NonceAndIV service. The NonceAndIV service can be created using:

```
SecureRandom random = SecureRandom.getInstance("NonceAndIV", "BC");
```

The NonceAndIV service differs from the DEFAULT service in that it only reseeds periodically. This puts less strain on the entropy source and is currently regarded as fine for IVs and random nonces, but it does not meet the seeding requirements for generating a key.

Building a SecureRandom by Hand

It is also possible to build SecureRandom objects that conform to SP 800-90A [13] style DRBGs in the regular BC API and the BCFIPS API by hand. This allows customisation of nonces, type, personalization data, and even entropy sources.

Owing to FIPS requirements the construction details are slightly different for the BC and the BCFIPS APIs. In the case of the regular BC API you can construct a SecureRandom object based on a hash algorithm as follows:

```
SP800SecureRandom random =
        new SP800SecureRandomBuilder(new SecureRandom(), true)
                .setPersonalizationString(
                      Strings.toByteArray("My Bouncy Castle SecureRandom"))
                .buildHash(new SHA512Digest(), null, false);
```

Note in the above case the nonce is null, and prediction resistance is set to false. This last point means generally this would make the DRBG less than ideal for generating keys as it will not reseed between requests. For secret and private key generation you normally want prediction resistance to be set to true.

The equivalent code in the BC FIPS API would look like the following:

```
FipsDRBG.Builder rBuild =
       FipsDRBG.SHA512.fromEntropySource(new SecureRandom(), true)
                      .setPersonalizationString(
                           Strings.toByteArray("My Bouncy Castle SecureRandom"));

FipsSecureRandom random = rBuild.build(null, tv.predictionResistance());
```

In both cases the SecureRandom passed in for initialisation is not used directly, only its `generateSeed()` method is used.

Why this Matters

Having a good entropy source matters because, at the end of the day, it affects the security of any keys you create, as well as the real randomness of any parameters such as initialization vectors.

That said, JVMs on Unix style operating systems are normally configured with /dev/random as the default source and this will normally block if the system runs out of entropy. There is also an additional random source /dev/urandom (notice the different name) which does not block if it runs out of entropy.

There is an argument in some quarters that /dev/urandom might be good enough - the reality, at the time of writing, is if you are using FIPS, /dev/urandom is, officially at least, not. There are two reasons for this: the entropy assessments that have already been done on most systems are using /dev/random, and the simple fact that a badly generated key is actual less useful than no key at all - you are better off waiting. Yes, the threat of a server blocking while you are encrypting private information is painful, but it will seem like an eternity in Paradise compared to what is likely to happen if you start encrypting things with low quality keys and your client's data is compromised as a result.

The first thing to do is, where you can, install hardware RNG support, or if you are using a virtual environment that does expose hardware RNG, the nearest appropriate equivalent to hardware RNG support.

The second thing you can do is use an entropy pool based on a valid DRBG - the regular BC provider uses one of these by default, the BC FIPS providers actually expect the entropy source to keep up with requests for randomness, but allow an entropy pool to be configured (see the section on provider configuration for details).

⚠ If you are using an entropy pool, you must only use it for generating seed material for other DRBGs. If you do otherwise you will be effectively leaking the internal state of your seed generator.

Finally, if you are trying to pull entropy for your DRBGs from some other system, keep in mind the contents of RFC 4086 "Randomness Requirements for Security" [57], SP 800-90B "Recommendation for the Entropy Sources Used for Random Bit Generation" [14] and the "golden rule" that just because a value you might pull from somewhere has 64 bits, it may only represent a few bits of actual entropy.

Bits of Security

You will see in different chapters, and in documents elsewhere on the Internet, references to bits of security strength. At the moment the values you will see will be from the set of 80, 112, 128, 192, and 256. As you would probably assume the larger the number the better.

For most algorithms these numbers are an attempt to provide a measure of computational hardness, or for an ideal symmetric cipher the complexity of a brute-force attack. Where an algorithm is, say, rated at 2^{80}, the assumption is that 2^{80} operations should be required to perform an effective attack on it.

As computers have gotten faster and faster, numbers like 2^{80} no longer represent the challenge they used to. These days it seems to be generally regarded that algorithms with at least a strength of 112 bits of security should be used, with a higher strength, 128 or greater, being used if there is an expectation the data being processed by an application will need to be secure more than 6 years out. Section 5.6.2 of SP 800-57 Part 1 (Revision 4) [12] includes a discussion on what considerations should be made in choosing algorithms to use with applications. For a different, although similar, perspective on decision making there is also the report produced by the European Union Agency for Network and Information Security (ENISA) [119].

Summary

In this chapter we have looked at the architecture of the JCA/JCE and how provider precedence works. We have also looked at the architecture for the different Bouncy Castle providers for the JCA/JCE and how they can be installed and configured. FIPS and non-FIPS configuration for the BCJSSE, the Bouncy Castle provider for the JSSE, has also been looked at.

We also looked at a list of common problems that can occur and their indicators. Hopefully this will be enough to get you up and running so you can work through the examples in the rest of the book.

Finally, we also looked at a more subtle configuration/setup issue affecting the use of cryptography providers in Java. That of the underlying quality of the random numbers available for secret and private key generation. Although a good quality source of random numbers is a largely unseen benefit, it is an important one. In the case of the FIPS provider, where the best RNG source is a possible performance issue, do not forget to check for hardware RNG support. Where that is not available, do not forget that it is possible to configure the BCFIPS provider to use an entropy pool as well.

Chapter 2: Block and Stream Ciphers

This chapter looks at the Java APIs that are used for doing encryption with symmetric ciphers. You will see that there are two different approaches to encrypting data using symmetric ciphers, one based on “virtual code-books” and the other based on mixing a message with a key-stream from a cipher. Configuring ciphers for basic modes will also be discussed and as well as the input and generation of keys and other parameters. Finally, we will have a look at what is available to make it easier to use ciphers with input/output streams.

The Basics

Symmetric ciphers rely on the use of a secret key to encrypt data. Expose the key to someone who should not have it and the game is over. We will look at methods for exchanging secret keys later in the book but as symmetric ciphers provide us with the means we need to handle encryption of large amounts of data it is worth looking at them first before worrying about anything else.

Algorithm Security Strengths

The security strength of symmetric ciphers, where properly analyzed, is described in bits of security.

These days, if you want to be safe, it is generally considered that your cipher should have a least 112 bits of security. The two symmetric ciphers currently recommended by NIST, AES and Triple-DES, have estimated security strengths presented in Table 2 of NIST SP 800-57 [12]. You can see these below.

NIST Symmetric Cipher Security Strengths

Algorithm	Key Size	Security Strength (bits)
2-Key Triple-DES	112	<= 80
3-Key Triple-DES	168	112
AES	128	128
AES	192	192
AES	256	256

Ideally a symmetric cipher’s security strength for a given key size is reflected by the key size used. From the table above you can see that AES is such a cipher, but you can also see from Triple-DES that it is not always the case that this is true. If you are looking to use other algorithms it is worth checking the literature to ensure that the keysize/algorithm combination you have in mind accurately reflects the bits of security you think your application requires.

How not to Keep a Secret

Before doing anything involving a symmetric secret key it is important to be aware that “not telling” about the secret key is only one of the things that affects the effective life of a key. There are a minimum of three other considerations that need to be kept in mind as well.

How the secret is generated
The quality of a key relies on the entropy of the random data used to generate it. While it is great to find a random value generator that produces, say 64 bits, if those bits are either all 1s or all 0s, there is only 1 bit of entropy. It will be easy to simply guess a secret key generated in such a fashion.

The amount of data and the number of times a key is used
Other considerations not withstanding, the general rule is the maximum number of blocks is around $2^{(blockSize/2)}$[1]. At a minimum you should plan to rotate the key out before the total number of blocks represented by this number is processed, either in a single message, or as an aggregate of messages processed in total.

The parameters associated with the key and cipher mode
Most cipher modes involve the use of other parameters which provide things like nonces and initialization vectors. With some modes, such as GCM, reusing an IV can be fatal to security. Generate any parameters wisely!

What does this all mean? The answer is to make sure you use quality entropy for generating keys and do not expect symmetric keys to last forever. When you design your application make sure you have some idea of how much “stress” from block processing keys will be under and make sure they are rotated out accordingly.

The JCE APIs

The Bouncy Castle APIs offer a couple of ways of dealing with ciphers. The common one across all of them is the JCE provider.

> Before we begin, note we are assuming that you have installed the Bouncy Castle provider. If you are using the BC FIPS provider, you will need to replace the provider name “BC” with “BCFIPS”.

The SecretKeySpec

The `javax.crypto.spec.SecretKeySpec` class provides the simplest mechanism for converting byte data into a SecretKey for the JCE. Extending the `SecretKey` interface directly, it just takes a byte array representing a key and an algorithm name, as in:

[1] If you an interested in more information on this, the issue is generally called a “generic birthday attack”. That should provide enough to start investigations on.

```
byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f");

SecretKey key = new SecretKeySpec(keyBytes, "AES");
```

The resulting key can then be used with the cipher of its algorithm name.

Symmetric Key Generation

As an alternative to the `SecretKeySpec` it is also possible to generate secret keys for a specific algorithm using the `javax.crypto.KeyGenerator` class.

```
KeyGenerator kGen = KeyGenerator.getInstance("AES", "BC");

SecretKey key = kGen.generateKey();
```

The KeyGenerator class also has `init()` methods to allow the generator to be configured for specific key sizes or algorithm parameters related to key processing.

The Cipher Class

The Cipher class provides the business end of the encryption/decryption process. The Cipher class takes a definition string and optionally a provider or provider name. The syntax of the definition string is as follows:

```
definition := cipher_name "/" mode "/" padding
            | cipher_name
```

The first version is especially relevant for block ciphers. For example:

```
Cipher c = cipher.getInstance("AES/CBC/PKCS7Padding");
```

will produce the cipher that has been described (AES in CBC mode using PKCS#7 Padding).

On the other hand:

```
Cipher c = cipher.getInstance("AES");
```

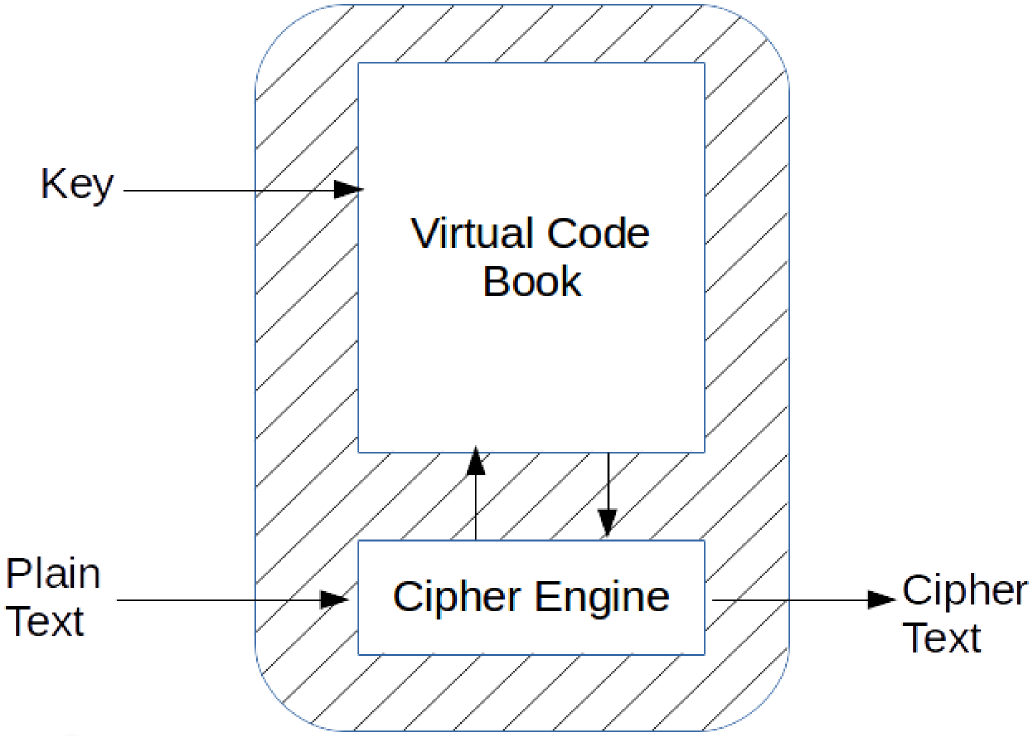
could return anything. It will be AES, but whether it will need parameters, an IV, or even be a mode that is safe for what you are trying to do is anyone's guess.

In the case of stream ciphers you will normally just see the `cipher_name` as stream ciphers do not have modes or paddings associated with them.

Block Ciphers

In the “good old days” people often used to encrypt messages using code-books, where a string of characters would be replaced with some string indicated by the code-book.

Fundamentally, block ciphers provide a mechanism for generating entries for such a code-book on the fly, with substitutions being done according to a string of bits representing the bit-size of the block cipher being used.



A Basic Block Cipher in Encryption Mode

Obviously generating random entries would not be very useful, so a block cipher uses the secret key it is initialised with to ensure that the code-book entries it generates are deterministic. By that I mean in the sense that a string of b-bits, representing a block of input to the cipher, will always map to the same, but most likely different corresponding string of b-bits in the virtual code-book. Input to the cipher is then mapped through the virtual code-book to produce the corresponding output.

There is a list of the block ciphers supported in Bouncy Castle in Appendix C.

Block Modes

Modes available for block ciphers divide into four types, modes that work with blocks, modes that can be used for streaming, authenticated modes, and modes for key wrapping. We will cover the block based modes and streaming modes in the rest of this chapter. Authenticated modes and key wrapping are built on these and we will look at those modes in Chapter 3.

ECB Mode

Electronic Code Book (ECB) mode is the symmetric cipher in its simplest form, and as the name of the mode suggests, the cipher is simply mapping one block of bits to a block of bits in the virtual code book represented by the secret key. The encryption/decryption setting of the cipher determines whether it is cipher text or plain text that is being produced by the operation.

Here is a basic example showing the use of AES in ECB mode. It is probably the simplest thing you can do in the JCE, but it shows the use of the Cipher class and the SecretKeySpec class.

```java
/**
 * A simple example of AES in ECB mode.
 */
public class ECBShortExample
{
    public static void main(String[] args)
        throws Exception
    {
        byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f");

        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

        Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding", "BC");

        byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7");

        System.out.println("input    : " + Hex.toHexString(input));

        cipher.init(Cipher.ENCRYPT_MODE, key);

        byte[] output = cipher.doFinal(input);

        System.out.println("encrypted: " + Hex.toHexString(output));

        cipher.init(Cipher.DECRYPT_MODE, key);

        System.out.println("decrypted: "
                            + Hex.toHexString(cipher.doFinal(output)));
    }
}
```

Running the example should produce the following output:

```
input    : a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7
encrypted: c600a21491fbcfec368b5adf31c50881
decrypted: a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7
```

As you can see the message is quite short, try replacing the message assigned to `input` with the longer message here:

```
    byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
                            + "a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7");
```

In this case you will see this output instead:

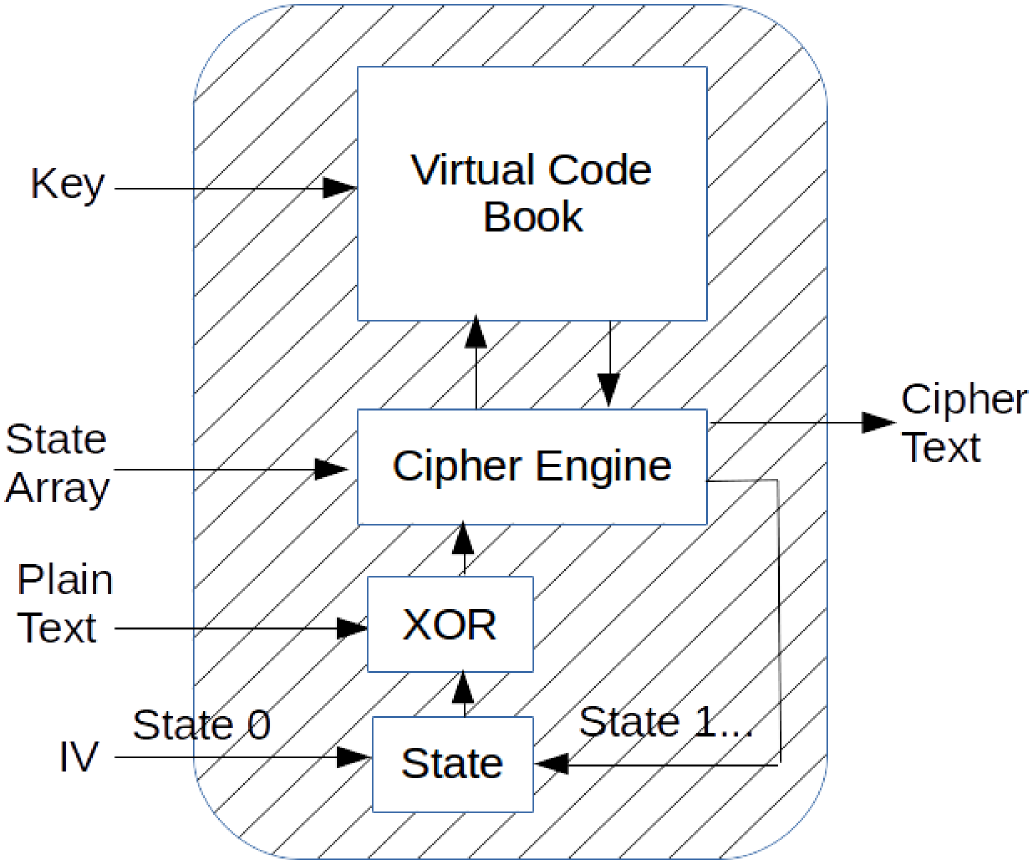
```
input    : a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7
encrypted: c600a21491fbcfec368b5adf31c50881c600a21491fbcfec368b5adf31c50881
decrypted: a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7
```

As we can see the second block in the cipher text produced is the same as the first block as the repetitions in the original message now extend to a full block. Code book mode is really just a lookup in a table, any patterns in the plain text will also appear in the cipher text if they exist on block boundaries. Generally, this is not really the effect we are looking for in encrypting data.

CBC Mode

In order to get away from these patterns, it is necessary to add some randomness to the stream and then allow that to propagate. This is where Cipher Block Chaining (CBC) mode comes in. CBC mode introduces a random initialization vector (IV) into the output stream and providing the IV is not reused with the same plain text the cipher text produced will always look different as the randomness of the IV will propagate through the blocks produced by the cipher.

CBC mode makes this happen by using the IV to initialise its internal state and then XORing the state with each block of plain text that comes in before encryption. When the XORed block is processed by the cipher engine the final result is output and also used to overwrite the internal state, with the updated internal state being used for the initial XOR step on the plain text and so on.



The basic CBC process - note the IV is used to provide initial state only

The following example uses CBC mode. Note that in addition to the string describing the cipher changing to specify "CBC" rather than "ECB", there is now a new parameter called `iv` which is being passed using an `javax.crypto.spec.IvParameterSpec`.

```
/**
 * A simple example of AES in CBC mode. Note the use of the IV.
 */
public class CBCExample
{
    public static void main(String[] args)
        throws Exception
    {
        byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f");

        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

        Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding", "BC");

        byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
                                + "a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7");

        System.out.println("input    : " + Hex.toHexString(input));

        byte[] iv = Hex.decode("9f741fdb5d8845bdb48a94394e84f8a3");

        cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(iv));

        byte[] output = cipher.doFinal(input);

        System.out.println("encrypted: " + Hex.toHexString(output));

        cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));

        System.out.println("decrypted: "
                            + Hex.toHexString(cipher.doFinal(output)));
    }
}
```

This time when you run the example you will see that the repetition in the output is no longer present.

```
input    : a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7
encrypted: efd8d07f7b4d2cd46fb0bb2c96123fb27d5345e9d20ad766f665c95783a4beee
decrypted: a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7
```

As was mentioned earlier, having a hard-coded IV, other than for the purposes of example, is not a good idea. The JCE offers three ways of creating a random one, you can either use a SecureRandom, as in:

```
SecureRandom random = new SecureRandom();
byte[] iv = new byte[cipher.getBlockSize()];

random.nextBytes(iv);
```

or you can have the `Cipher` object create one for you. The generated IV can be returned either as byte array or as an `AlgorithmParameters` object. Whether or not an IV is auto-generated is determined by whether the mode requires an IV.

This next example shows how auto-generation of an IV can be done for CBC mode. You will find that any ciphers which require the use of the IV will support something similar.

```
 1  /**
 2   * A simple example of AES in CBC mode, with the Cipher generating IV.
 3   */
 4  public class CBCGenExample
 5  {
 6      public static void main(String[] args)
 7          throws Exception
 8      {
 9          byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f");
10
11          SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
12
13          Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding", "BC");
14
15          byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
16                                  + "a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7");
17
18          System.out.println("input    : " + Hex.toHexString(input));
19
20          cipher.init(Cipher.ENCRYPT_MODE, key);
21
22          byte[] iv = cipher.getIV();
23
24          byte[] output = cipher.doFinal(input);
25
26          System.out.println("encrypted: " + Hex.toHexString(output));
27
28          cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));
29
30          System.out.println("decrypted: "
31                              + Hex.toHexString(cipher.doFinal(output)));
32      }
33  }
```

If you run this example a few times you should see the encrypted output changing randomly.

The alternative to using the generated byte array returned by `Cipher.getIV()` is to use `Cipher.getParameters()` which returns an `AlgorithmParameters` object. This can be used in the same way the `IvParameterSpec` was in the last example, by changing the example using the following fragment as a guide. This code replaces the section from line 20 to line 28 in the example above.:

```
 cipher.init(Cipher.ENCRYPT_MODE, key);

 AlgorithmParameters ivParams = cipher.getParameters();

 byte[] output = cipher.doFinal(input);

 System.out.println("encrypted: " + Hex.toHexString(output));

 cipher.init(Cipher.DECRYPT_MODE, key, ivParams);
```

In the case of the two examples of auto-generation of IVs we are using the provider's default internal `SecureRandom` for generating the IV. Where you need to, it is also possible to pass in a `SecureRandom` to a `Cipher` when calling the `init()` method so that your own source of randomness is used. For example you might write:

```
SecureRandom myRandom = ...

Cipher.init(Cipher.ENCRYPT_MODE, key, myRandom);
```

where `myRandom` is a `SecureRandom` you have built especially from a DRBG constructor.

> ⚠ Forgetting to set the correct IV for a decryption mode that requires one is a common error. If the IV is the problem with a CBC mode decryption, only the first block will be wrong in the decrypted output, if every block is wrong the key is also incorrect.

Padding

Block ciphers in their basic form, when still used in their code-book form, such as with ECB, or when underlying a CBC cipher can only process data that is aligned with their block size. Naturally it is not the case that the data you want to encrypt is always exactly a multiple of your block size, so a number of procedures have been proposed for adding padding to data.

All padding mechanisms (except occasionally zero padding) add an extra block of padding when the original plain text is block aligned. In this case the extra block the padding will be a string of bytes equal to the byte length of the block size of the cipher (for example 16 for AES, 8 for Triple-DES).

With the exception of Zero Byte Padding, which is not supported by the BC FIPS provider, the following padding mechanisms are support by the Bouncy Castle providers.

ISO7816-4 Padding

This padding is defined in ISO7816-4. The first pad byte has the value 0x80 and the remaining padding is made up of zero bytes. Set by the padding string “ISO7816-4Padding”.

ISO101026-2 Padding

This padding is defined in ISO10126-2. The last byte of the padding is the number of pad bytes and the remaining pad bytes are made up of random data. Set by the padding string "ISO10126-2Padding".

PKCS#5/PKCS#7 Padding

PKCS#5/PKCS#7 padding was originally defined in PKCS #5 [100] with a further revision in PKCS #7 [47]. The plain text is padded with a string of bytes representing the number of bytes of padding to make up a block. So for example with a cipher like AES, which a 16 byte block, plain text which is 5 bytes shorter than the next block boundary will be padded with bytes of the value '5'. Set by the padding string "PKCS5Padding" or "PKCS7Padding".

TBC Padding

Trailing bit complement (TBC) padding is added based on the value of the last bit in the plain text. If it is 0 then a string of 1s is added, if it is 1 then a string of 0s is added. Set by the padding string "TBCPadding".

X9.23 Padding

This padding is defined in X9.23. Like PKCS#5/PKCS#7 the last byte of the padding is the number of pad bytes, but unlike PKCS#5/PKCS7 the other pad bytes are either all zero, or made up of random data. Set by the padding string "X9.23Padding".

Zero Byte Padding

Don't ever use this in a new system. You may still run into it with legacy software though. Padding in this case is a string of zero bytes, just pray no one ever produces plain text that ends with a zero value. Set by the padding string "ZeroBytePadding".

As far as other providers go, PKCS#5/PKCS#7 padding is by far the most common. Occasionally it is referred to as "PKCS5Padding", sometimes "PKCS7Padding", often both (BC will recognise both). The reason for the two names is that the original definition, which appeared in PKCS #5, was for ciphers with up to 8 bytes in block size. PKCS #7 was where the padding mechanism was extended to a more general case.

The following example shows how to make use of PKCS#5/PKCS#7 padding with CBC mode. Note that in the example we do not simply use `Cipher.doFinal()`, we will discuss what is going on after the code.

```java
/**
 * A simple example of AES in CBC mode with block aligned padding.
 */
public class CBCPadExample
{
    public static void main(String[] args)
        throws Exception
    {
        byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f");

        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding", "BC");

        byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
                                + "a0a1a2a3a4a5a6a7a0");

        System.out.println("input    : " + Hex.toHexString(input));

        byte[] iv = Hex.decode("9f741fdb5d8845bdb48a94394e84f8a3");

        cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(iv));

        byte[] output = cipher.doFinal(input);

        System.out.println("encrypted: " + Hex.toHexString(output));

        cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));

        byte[] finalOutput = new byte[cipher.getOutputSize(output.length)];

        int len = cipher.update(output, 0, output.length, finalOutput, 0);

        len += cipher.doFinal(finalOutput, len);

        System.out.println("decrypted: "
                 + Hex.toHexString(Arrays.copyOfRange(finalOutput, 0, len)));
    }
}
```

The example introduces two new methods, `Cipher.getOutputSize()` and `Cipher.update()` as well as a new variation on `Cipher.doFinal()`. The first thing to note is that the return value of the `update()` method is kept and used as the offset to pass to the `doFinal()` method as the start position for the remaining output `doFinal()` will output. The second thing to note is that the return value of `getOutputSize()` is not assumed to be the actual value of the length of the decrypted output. The only guarantee that `getOutputSize()` makes is that the length of the end result will not be longer than the return value of `getOutputSize()`.

If you run the example you will see the following output:

```
input    : a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
encrypted: efd8d07f7b4d2cd46fb0bb2c96123fb2480e1ff81f1de9d67488a773b26c6258
decrypted: a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
```

You can see the presence of the padding in encrypted data as it is both longer than the original input and block aligned.

Cipher Text Stealing

If you have a requirement to deal with arbitrary length data (with one restriction), Cipher Text Stealing, or CTS, can be used. The CTS operation was first published as an RFC in RFC 2040 [45] and further expanded on in an addendum to SP 800-38A [5]. It is treated as a special case of CBC mode, and for Java purposes we will regard it as a padding mechanism. While CTS is only usable for messages that are more than one block in size, it does provide a handy way of producing cipher texts using CBC mode which are the exact length of the original plain text.

The addendum to SP 800-38A defines 3 ways of doing it, one of which CS3, is the same as the one defined in RFC 2040. The following example is for this mode, which can be referred to in the provider as either “CS3Padding”, or “CTSPadding”.

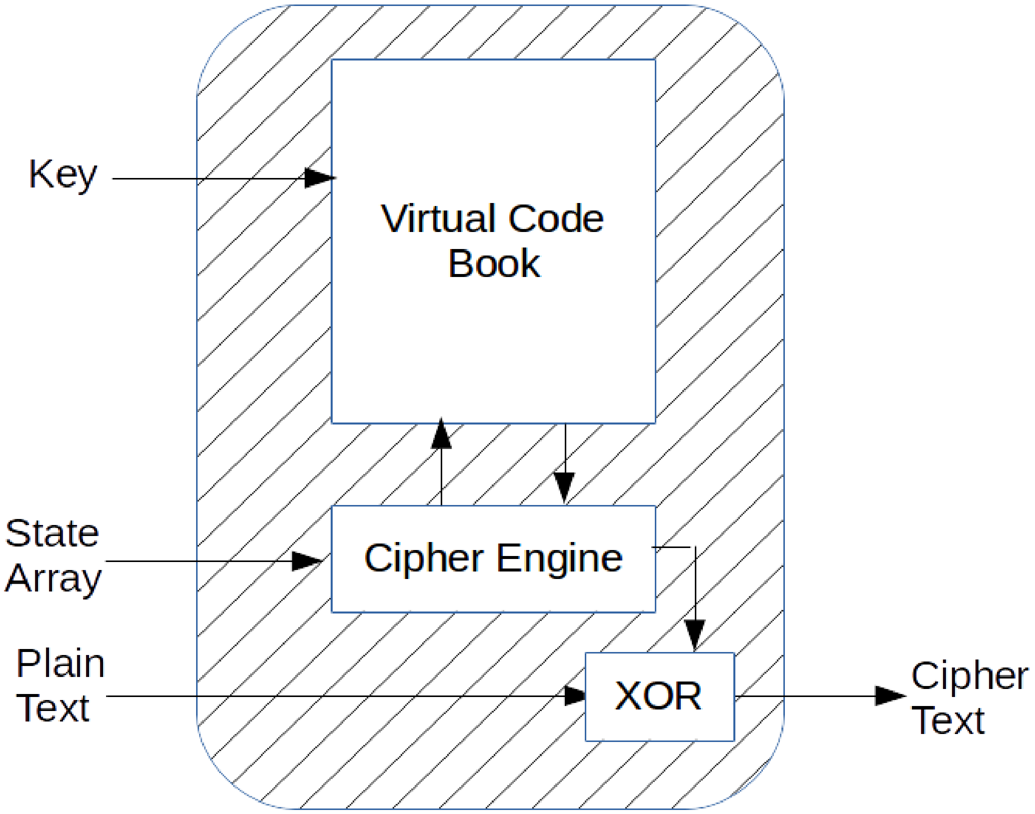
```java
/**
 * A simple example of AES using CBC mode with Cipher Text Stealing used.
 */
public class CTSExample
{
    public static void main(String[] args)
        throws Exception
    {
        byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f");

        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

        Cipher cipher = Cipher.getInstance("AES/CBC/CTSPadding", "BC");

        byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
                                + "a0a1a2a3a4a5a6a7a0");

        System.out.println("input    : " + Hex.toHexString(input));

        byte[] iv = Hex.decode("9f741fdb5d8845bdb48a94394e84f8a3");

        cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(iv));

        byte[] output = cipher.doFinal(input);

        System.out.println("encrypted: " + Hex.toHexString(output));

        cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));

        System.out.println("decrypted: "
                            + Hex.toHexString(cipher.doFinal(output)));
    }
}
```

Running the example will produce the following output:

```
input    : a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
encrypted: caa168f23f888234125f2e6bd9c1934eefd8d07f7b4d2cd46f
decrypted: a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
```

As you can see the encrypted output is identical in length to the original plain text.

Streaming Block Modes



Use of a Block Cipher in a Streaming Encryption Mode

The streaming modes are so called as they make it possible to encrypt data of an arbitrary size without the use of padding. In this case the block cipher is used to generate a key-stream which is then XORed with the data being encrypted. These modes have another useful feature. While at the heart of it the block cipher is still using its virtual code-book, the modes only require an implementation of the cipher's encryption algorithm as the code-book is now being used only to generate an encryption stream to XOR against either the data for encryption or the cipher text for decryption.

CTR Mode

Counter (CTR) mode, also known as Segmented Integer Counter (SIC) mode was standardised by NIST in SP 800-38a [4] and by the IETF in RFC 3686 [56]. Of the block streaming modes this is by far the most useful.

The mode is straight forward, the key stream is generated by encrypting a block which is a mixture of a random value (the nonce) and a counter. The amount of space available to the counter determines how many blocks the cipher can process safely - so an 8 bit counter only allows for 256 blocks, for example. A different nonce combined with the same initial counter should result in a unique key stream, so the size of the nonce, combined with its randomness, gives a level of certainty to the number of messages that can be processed by the key. Usually the nonce takes up the bits in the block generating the key stream that the counter is not using. Once the nonce space is exhausted the key to the cipher should be changed before any further messages are encrypted.

The following example makes use of CTR mode.

```java
 1 /**
 2  * A simple example of AES in CTR mode.
 3  */
 4 public class CTRExample
 5 {
 6     public static void main(String[] args)
 7         throws Exception
 8     {
 9         byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f");
10 
11         SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
12 
13         Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding", "BC");
14 
15         byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
16                                 + "a0a1a2a3a4a5a6a7a0");
17 
18         System.out.println("input    : " + Hex.toHexString(input));
19 
20         byte[] iv = Hex.decode("010203040506070809101112");
21 
22         cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(iv));
23 
24         byte[] output = cipher.doFinal(input);
25 
26         System.out.println("encrypted: " + Hex.toHexString(output));
27 
28         cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));
29 
30         System.out.println("decrypted: "
31                             + Hex.toHexString(cipher.doFinal(output)));
32     }
33 }
```

Looking at the code sample, the thing which hopefully stands out is that the IV specified on line 20 is 12 bytes long. Not every provider will support this and in some cases you may find it necessary to provide enough zeroes to make up a full block at the end of the IV. The reason the IV is truncated in this case is that the Bouncy Castle providers will interpret the truncated IV to mean that the missing 4 bytes make up the counter used for the CTR mode and will throw an exception if an attempt is made to exceed it.

Running the example will produce the following output:

```
input    : a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
encrypted: 80a819788cc4bb6d21da14c70dff3f63ace0346c48137cc6b2
decrypted: a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
```

As with CTS, we can see that despite the use of the block cipher, the encryption process has produced output which is the same length as the input plain text.

CFB Mode

CFB, or Cipher FeedBack mode, uses the plain text as well as an initial IV, to provide the source for the key stream. A CFB mode cipher starts with the IV and encrypts it using the underlying cipher forming an initial state array. As you encrypt each block of plain text, the plain text is XORed with the state array to get the next block of cipher text and the block of cipher text is then copied into the state array, which is then encrypted to provide the key stream for the next block. The mode is defined in SP 800-38a [4] and CFB mode is commonly used on 8 bit blocks as well as on whole blocks.

For a cipher processing a whole block of the underlying cipher each round, we just use the string "CFB" in the mode section of the cipher definition.

```
Cipher cipher = Cipher.getInstance("AES/CFB/NoPadding", "BC");
```

To create a cipher using CFB mode with 8 bit blocks we include the number of bits in the mode field, as in:

```
Cipher cipher = Cipher.getInstance("AES/CFB8/NoPadding", "BC");
```

Note this changes the way the state array is used, as only 8 bits is being processed on each round. Assuming an underlying cipher with a block size of more than 8 bits, the 8 bits will be shifted into the state array replacing the end of it, with the start of the state array being discarded. This also means that the encryption method on the cipher will be invoked for every 8 bits, instead of once for a full block. For a cipher like AES, with a block size of 16 bytes, this will make the cipher operation 16 times slower.

Like CBC mode, if there is an error in the data being decrypted, the error will propagate into a full word size of bytes before the stream resynchronizes.

OFB Mode

OFB (Output FeedBack) mode, is different from CFB mode in that the initial IV is treated as the cipher's original state and is simply encrypted as each new block of the key stream is required for XORing with the key stream. The plain text being encrypted is never processed directly by the underlying cipher. The mode is defined in SP 800-38a [4].

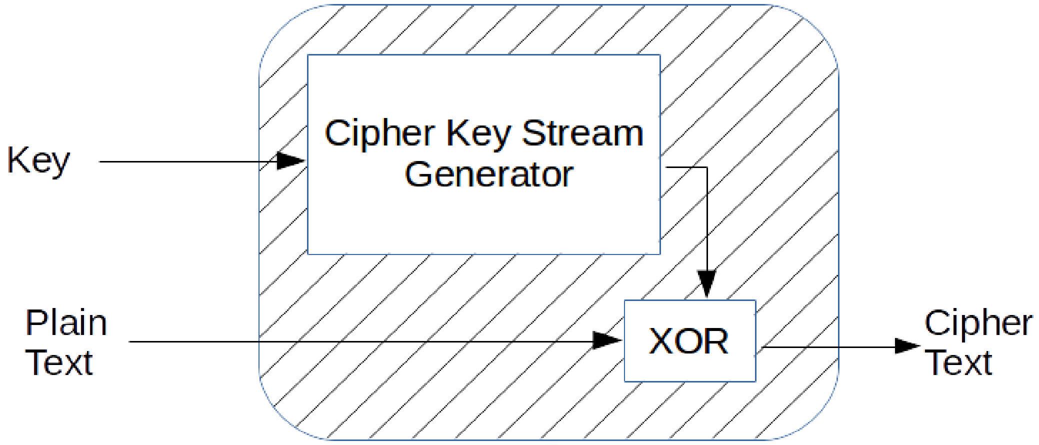
To use OFB mode from the JCE you simply include the OFB string in the mode, as in:

```
Cipher cipher = Cipher.getInstance("AES/OFB/NoPadding", "BC");
```

OFB mode does have the advantage that bit errors do not propagate, which in some situations, such as when dealing with video or audio can be useful. The disadvantage of OFB mode is that if a cycle exists in the key-stream (as in for a given key and a given input value it turns out the code-book returns the original value for the output when indexed) the stream can start to repeat unexpectedly. As you can imagine the odds against this happening are fairly large, however the more you work in this area, the more you will start to appreciate how easy it is to get “lucky”. For this reason while we have listed OFB mode, it is really for compatibility reasons and we would not use it for anything new. CTR mode also has all the advantages of OFB mode, without the disadvantages, so, compatibility needs aside, you are better off using CTR mode.

Stream Ciphers

Stream ciphers are ciphers designed to generate only key-streams.



A Basic Stream Cipher in Encryption Mode

Like a streaming mode of a block cipher the key stream is then XORed with the input plain text in order to produce cipher text. Also, like block ciphers, stream ciphers are initialised with a symmetric secret key and sometimes other parameters such as an IV. Unlike block ciphers, stream ciphers do not have modes, so you will normally just invoke them by name.

There is a list of the stream ciphers supported by Bouncy Castle in Appendix C.

The following example shows the use of the version of ChaCha20 defined in RFC 7539 [95].

```java
/**
 * Example of the use the RFC 7539 ChaCha stream cipher.
 */
public class StreamExample
{
    public static void main(String[] args)
        throws Exception
    {
        byte[] keyBytes = Hex.decode("000102030405060708090a0b0c0d0e0f"
                                   + "000102030405060708090a0b0c0d0e0f");

        SecretKeySpec key = new SecretKeySpec(keyBytes, "ChaCha7539");

        Cipher cipher = Cipher.getInstance("ChaCha7539", "BC");

        byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
                                + "a0a1a2a3a4a5a6a7a0");

        System.out.println("input    : " + Hex.toHexString(input));

        byte[] iv = Hex.decode("010203040506070809101112");

        cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(iv));

        byte[] output = cipher.doFinal(input);

        System.out.println("encrypted: " + Hex.toHexString(output));

        cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));

        System.out.println("decrypted: "
                            + Hex.toHexString(cipher.doFinal(output)));
    }
}
```

As you can see in the example code the mode and padding designators are missing from the Cipher definition as they are not required. The output from this example is:

```
input    : a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
encrypted: 3a77a592ab7b7c8fb1dca9e467e56dca5d3c26e623e7a59c73
decrypted: a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7a0
```

and we can see that the cipher is producing the same length of cipher text as the original plain text.

Cipher Based Input/Output

The JCE provides the classes `javax.crypto.CipherInputStream` and `javax.crypto.CipherOutputStream` which provide a way of introducing ciphers as filters on streams. Unfortunately these classes silently fail if there are issues with the call to the `doFinal()` method on the cipher they wrap. Whether this is an issue for you is mostly up to you to decide, although be aware that a side effect of this is that the silent failure means that tag exceptions thrown by AEAD ciphers are also ignored (we will look at what this means in Chapter 3). Not really a good thing.

Keeping the above in mind, the following example is using the Bouncy Castle versions of `CipherInputStream` and `CipherOutputStream` which you can find in the `org.bouncycastle.jcajce.io` package. You should find the example works just as well with the I/O classes from `javax.crypto` as well.

```
/**
 * An example of use of the BC library CipherInputStream/CipherOutputStream
 * classes.
 */
public class CipherIOExample
{
    public static void main(String[] args)
        throws Exception
    {
        KeyGenerator kGen = KeyGenerator.getInstance("AES", "BC");

        SecretKey key = kGen.generateKey();

        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding", "BC");

        byte[] input = Hex.decode("a0a1a2a3a4a5a6a7a0a1a2a3a4a5a6a7"
                                + "a0a1a2a3a4a5a6a7a0");

        System.out.println("input    : " + Hex.toHexString(input));

        cipher.init(Cipher.ENCRYPT_MODE, key);

        AlgorithmParameters ivParams = cipher.getParameters();
        // encrypt the plain text
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        CipherOutputStream cOut = new CipherOutputStream(bOut, cipher);

        cOut.write(input);
        cOut.close();

        byte[] output = bOut.toByteArray();

        System.out.println("encrypted: " + Hex.toHexString(output));

        cipher.init(Cipher.DECRYPT_MODE, key, ivParams);

        // decrypt the cipher text
        ByteArrayInputStream bIn = new ByteArrayInputStream(output);

        CipherInputStream cIn = new CipherInputStream(bIn, cipher);

        byte[] decrypted = Streams.readAll(cIn);

        System.out.println("decrypted: "
                            + Hex.toHexString(decrypted));
    }
}
```

You will note that the cipher used here is an AES cipher in CBC mode with PKCS #7 padding. If you run the example you should see that the encrypted output is, as expected, longer than the input, and that the padding has been cleanly removed by the `CipherInputStream` on the decryption step.

Make sure in the case of a `CipherOutputStream` you always call `close()` and in the case of a `CipherInputStream` you always read to end of file. It is the only way to be certain `doFinal()` is called on the underlying cipher.

Summary

In this chapter we have looked at Block and Stream ciphers. We covered the generation of keys used by the ciphers, the different sorts of modes that block ciphers can operate under with the approaches used for padding the encrypted data. We also covered how to use block ciphers for streaming using modes like CTR and CFB.

Finally, we also looked at the way that the JCE provides input and output using ciphers via the CipherInputStream and CipherOutputStream classes. This is one of the most common approaches for reading and writing encrypted data in Java. We also noted that if you are using AEAD ciphers you should use the the `CipherInputStream` and `CipherOutputStream` from the `org.bouncycastle.jcajce.io` as the JCE provided classes will fail silently if an AEAD tag fails to validate.

Chapter 3: Message Digests, MACs, HMACs, KDFs, and XOFs

This chapter looks at methods for verifying that data has not been tampered with - whether encrypted and decrypted or simply sent in the clear. This involves the use of functions called message digests to calculate a cryptographic hash for a set of input. In addition to tamper identification some surety can be provided as to the origins of the input by computing a message digest and including a shared secret. The cryptographic checksum, in this second case, means it is possible to assert that the message had a specific originator and can be used to avoid the risk of replays. The functions that provide this additional level of assurance are called MACs. The main reason these functions work is that they are not reversible. This lack of reversibility also makes both digests and MACs useful for random number generation and key derivation, both areas where it is desirable to hide the original seed material fed into the generator.

Message Digests

Message digests are used to calculate a cryptographic checksum, or hash, for a particular message. The key difference between a message digest and something like a CRC is that a small change to the input of a message digest will have an unpredictable and possibly huge effect on the value of the bits making up the result of the message digest function. While the output of the message digest for a particular message cannot be guessed without first doing the calculation, the size of the message digest and the function itself also provides an indication of how likely two different messages are to produce the same digest result, also known as a *collision*.

The MessageDigest Class

The JCA provides access to objects that support message digest calculation through the `MessageDigest` class which can be found in the `java.security` package. As it comes under the JCA, objects of this type are not created directly, but are created via `getInstance()` methods.

There is no setup required with a `MessageDigest`. After creation, data is added to the calculation using different `update()` methods and the result is returned via one of the `digest()` methods, as in the example that follows:

```
  /**
   * Return a digest computed over data using the passed in algorithm
   * digestName.
   *
   * @param digestName the name of the digest algorithm.
   * @param data the input for the digest function.
   * @return the computed message digest.
   */
  public static byte[] computeDigest(String digestName, byte[] data)
      throws NoSuchProviderException, NoSuchAlgorithmException
  {
      MessageDigest digest = MessageDigest.getInstance(digestName, "BC");

      digest.update(data);

      return digest.digest();
  }
```

A simple use of the `computeDigest()` function follows:

```
/**
 * A simple example of using a MessageDigest.
 */
public class DigestExample
{
    public static void main(String[] args)
        throws Exception
    {
        System.out.println(
            Hex.toHexString(
                computeDigest("SHA-256", Strings.toByteArray("Hello World!"))));
    }
}
```

If you run the example and all goes well you should be presented with the following output.

```
7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069
```

If you try tweaking the input string in `DigestExample` you will see that the resulting digest changes in an apparently random fashion, even if the string is kept at is current length of twelve characters, or 96 bits, well below the 256 bits the message digest is producing.

`MessageDigest` also supports a “one-shot” calculation method also called `digest()`, like the one in the example, which takes a single byte array as input. There is also a static `MessageDigest.isEqual()`

method on the class as well which takes 2 byte arrays. The `isEqual()` method offers constant time equality checking, although if you are really looking for a more paranoid constant-time, there is also `org.bouncycastle.util.Arrays.constantTimeAreEqual()` in the Bouncy Castle APIs.

MACs

MACs are used in situations where an assurance is required about how the cryptographic checksum is calculated. They involve the use of a shared secret between the party creating the MAC and the party using it. They go one step further than message digests in that they can be used to link the message the MAC is calculated on to a specific originator and also as a means to eliminate replay messages MACs can be generated using either using ciphers or message digests. Unlike message digests, MAC functions assume a key as well as the input data. With a constant key, MACs can be used to perform a similar function to message digests, but they are still better than a simple CRC as changes to the input to the MAC will result in unpredictable bit changes to the result of the MAC. When the key that is used to initialise the MAC is kept secret the MAC can also provide some assurances about the originator of the data that was MACed as well.

It is also quite common to truncate MACs, mostly this is quite safe with the following caveat: the smaller the MAC is the more likely you are to have a collision so truncating a MAC affects both the longevity of the key used to initialise it, as well as the reliability of the MAC itself. Before you start dropping bits, it is a good idea to make sure any affects like this are taken into account.

The Mac Class

The JCE provides access to MAC functions through the `javax.crypto.Mac` class. Like the `Cipher` class objects of this type are not created directly and use the `getInstance()` factory pattern.

Here is a basic method for computing a MAC.

```
/**
 * Return a MAC computed over data using the passed in MAC algorithm
 * type algorithm.
 *
 * @param algorithm the name of the MAC algorithm.
 * @param key an appropriate secret key for the MAC algorithm.
 * @param data the input for the MAC function.
 * @return the computed MAC.
 */
public static byte[] computeMac(String algorithm, SecretKey key, byte[] data)
    throws NoSuchProviderException, NoSuchAlgorithmException,
           InvalidKeyException
{
    Mac mac = Mac.getInstance(algorithm, "BC");

    mac.init(key);

    mac.update(data);

    return mac.doFinal();
}
```

As you can see the principle difference between `Mac` and `MessageDigest` is the presence of the `init()` method and the use of the `SecretKey`. As mentioned earlier, it is due to the use of the key MACs are often also used to authenticate the origins of, as well as to verify the integrity of, the contents of a message.

The other difference with the `Mac` class is we are now using `Mac.doFinal()` as the method for retrieving the final checksum, giving the class a similar feel to the `Cipher` class. The `init()` methods on `Mac` class largely follow the `Cipher` class as well and some MAC algorithms can take extra parameters, such as IVs, which serve similar purposes to what the parameters are used for in an equivalent cipher type. There is also a version of `doFinal()` which takes a byte array and can be used as a one-stop method for calculating a `Mac` just as the one-stop `digest()` method works on `MessageDigest`.

The following example provides a use of the `computeMac()` method to calculate a MAC using AES with the CMAC algorithm:

```
/**
 * A simple example of using AES CMAC.
 */
public class MacExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey macKey = new SecretKeySpec(
            Hex.decode("dfa66747de9ae63030ca32611497c827"), "AES");

        System.out.println(
            Hex.toHexString(
                computeMac("AESCMAC", macKey,
                                 Strings.toByteArray("Hello World!"))));
    }
}
```

If you run this and all goes well you should get:

```
24331a9b3a21f382e41c8a485fababf0
```

which, like the block size of the underlying cipher, is a 128 bit value. Note there are a number of different ways of calculating a MAC with a given cipher, such as CBC-MAC and GMAC to name two others. A full list can be found in Appendix C.

HMACs

As mentioned, it is also possible to use a message digest to compute a MAC. MACs calculated in this fashion are known as Hash MACs, or HMACs for short.

The following example shows use of a HMAC via the JCE `Mac` class. As you can see, other than the `macKey` setup and the algorithm name, the same `computeMac()` method we saw earlier can be used for a HMAC as well.

```
/**
 * A simple example of using a HMAC SHA-256.
 */
public class HMacExample
{
    public static void main(String[] args)
        throws Exception
    {     Security.addProvider(new BouncyCastleProvider());
        SecretKey macKey = new SecretKeySpec(
                Hex.decode(
                    "2ccd85dfc8d18cb5d84fef4b19855469" +
                    "9fece6e8692c9147b0da983f5b7bd413"), "HmacSHA256");

        System.out.println(
            Hex.toHexString(
                computeMac("HmacSHA256", macKey,
                                 Strings.toByteArray("Hello World!"))));
    }
}
```

Running this example will produce a 256 bit string:

```
8e079dd2d9b271a17a4d3dfa3a808b41d2105fee7aabe7ceccfdf66e8339e5d7
```

In this case, reflecting the size of the message digest used to support the HMAC function.

Key Derivation Functions

Key derivation functions (KDFs), make use of message digests and/or MACs to take a set of inputs, in some cases which need to be kept secret, and then produce a stream of data suitable for using for a key of varying size. Some KDFs do restrict the size of the key to the maximum size of the output digest. There is an example of this in the German standard BSI TR-03111 [108] in section 4.3.3 where the output of a digest, either SHA-1 or SHA-256, is used to produce a symmetric cipher session key. More commonly KDFs are flexible about the size of the output they produce.

The most common of the flexible KDFs is defined in X9.63 [36] and widely referred to elsewhere including in BSI TR-03111 and the NIST standards. The next most common is what was originally called the NIST Concatenation KDF, and is now described in SP 800-56C [11], Section 4. We will look at both of those functions here.

The X9.63 KDF

For inputs the KDF requires:

- Z which is normally a generated secret, usually from a key agreement scheme.
- $SharedInfo$ an optional byte string that is combined with Z.
- $keyLen$ the length of the key to be generated.
- a function $H()$ which is a message digest we are building the KDF on.
- $hLen$ which is the output size of $H()$.

$SharedInfo$ is an optional string of bytes used in the KDF calculation. It takes its name from the fact that where more than one entity is trying to use the same Z value to generate the same key material the string of bytes must be shared in some fashion before the KDF is used. The aim of it is to provide some additional data to help make sure the output of the KDF is unique compared to other KDFs regardless of whether the value of Z might clash.

After validation of the inputs the following algorithm is performed to calculate the key material:

```
1. Set $R = \lceil keyLen/hLen \rceil$
2. Set $C = 1$
3. For $I = 1$ to $R$
    a. Set $K = H(Z \parallel ItoB(C) \parallel SharedInfo)$
    b. Set $C = C + 1$
    c. Set $Result = Result \parallel K$
4. Return leftmost $keyLen$ bytes of $Result$
```

Where $ItoB()$ is a function that converts C into a 4 byte big endian integer.

The Concatenation KDF

Originally described in an earlier version of SP 800-56A, the algorithm representing the concatenation KDF is now described in SP 800-56C [11], Section 4, which is titled “One-Step Key Derivation”. Tables are provided in SP 800-56C which can be used for validation of the different inputs to the KDF.

For inputs the KDF requires:

- Z which is normally a generated secret, usually from a key agreement scheme.
- $FixedInfo$ a byte string that is combined with Z.
- $keyLen$ the length of the key to be generated.
- a function $H()$ which is a message digest or HMAC we are building the KDF on.
- $hLen$ which is the output size of $H()$.

As its name implies, $FixedInfo$ is a constant string of bytes used in the KDF calculation. As with the $SharedInfo$ parameter in the X9.63 KDF, the aim of the $FixedInfo$ is to provide some additional data to help make sure the output of the KDF is unique compared to other KDFs in use regardless of whether the value of Z might clash. Both SP 800-56A [9] and SP 800-56B [10] provide detailed recommendations on what can go into $FixedInfo$ to help with this goal. One of the recommendations includes an agreed secret value and we will see later how this can be used to introduce post-quantum key exchange algorithms without compromising the integrity of the key exchange algorithm that is providing the Z value.

After validation of the inputs the following algorithm is performed to calculate the key material:

```
1. Set $R = \lceil keyLen/hLen \rceil$
2. Set $C = 0$
3. For $I = 1$ to $R$
    a. Set $C = C + 1$
    b. Set $K = H(ItoB(C) \parallel Z \parallel FixedInfo)$
    c. Set $Result = Result \parallel K$
4. Return leftmost $keyLen$ bytes of $Result$
```

Where $ItoB()$ is also a function that converts C into a 4 byte big endian integer. As you can see the principal difference between this KDF and the one from X9.63 is that the ordering of the counter, C, and the generated value, Z, are reversed.

As you would expect, you cannot just generate an endless string of key material from either of the functions we have looked at. Limits and security strengths for the NIST function are documented for different hash/MAC functions in SP 800-56C. For our purposes, you will never come close to hitting the limit on the amount of key material that can be produced, but you should make sure the hash/MAC function used in the KDF is of the appropriate security strength for the key size you are generating.

Expandable Output Functions

With the arrival of SHA-3, two additional functions arrived, SHAKE128 and SHAKE256. While both sponge-based they are different from the SHA-3 digests per se as they are able to accommodate different output sizes by design. This new class of hashing algorithm are called Expandable Output Functions (XOFs) and SHAKE128 and SHAKE256 have been used as the basis of a range of hashing and MAC algorithms which can produce variable size output.

In Bouncy Castle's low-level API, XOF functions implement `org.bouncycastle.crypto.Xof` which is defined as:

```java
public interface Xof
    extends ExtendedDigest
{
    /**
     * Output the results of the final calculation for this digest
     * to outLen number of bytes.
     *
     * @param out output array to write the output bytes to.
     * @param outOff offset to start writing the bytes at.
     * @param outLen the number of output bytes requested.
     * @return the number of bytes written
     */
    int doFinal(byte[] out, int outOff, int outLen);

    /**
     * Start outputting the results of the final calculation for
     * his digest. Unlike doFinal, this method will continue producing
     * output until the Xof is explicitly reset, or signals otherwise.
     *
     * @param out output array to write the output bytes to.
     * @param outOff offset to start writing the bytes at.
     * @param outLen the number of output bytes requested.
     * @return the number of bytes written
     */
    int doOutput(byte[] out, int outOff, int outLen);
}
```

As you can see the additional methods allow for specifying a variable size of output on `doFinal()` and also the potential for continuous output - the `doOutput()` method - until a `doFinal()` is called. It is this ability to squeeze out extra data that the expandable functions get their names from.

The SHAKE Functions

The primary implementors of `Xof` are SHAKE128 and SHAKE256. The 128 in SHAKE128 and the 256 in SHAKE256 derives from the maximum bits of security associated with the algorithms and the default output sizes for the two of them are 256 bits and 512 bits respectively. If you create either of these using the `MessageDigest.getInstance()` method the minimum size is what you will be returned. Below these minimum sizes the actual strength of the output is $d/2$ where d is the length in bits of the output.

cSHAKE, KMAC, TupleHash, and ParallelHash

In 2016, just over a year after the publication of the SHA-3 standard, NIST released SP 800-185 [16], which defined a function built on SHAKE called cSHAKE, as well as an additional three functions, KMAC, TupleHash, and ParallelHash which were all built on cSHAKE.

cSHAKE

cSHAKE introduces two configuration parameters, N and S. N is referred to as the function name string and S is the customization string. N should only ever be set as part of a NIST standard - we will see some examples of this later. S on the other hand is for local use - we, and you, may set it to what we wish. If both N and S are empty strings then the output of cSHAKE is the same as the output of the corresponding SHAKE function.

At the time of writing cSHAKE can only be used properly from the low-level APIs in Bouncy Castle. For the general provider, N and S are set via the engine's constructor. For example:

```
CSHAKEDigest cshake = new CSHAKEDigest(
                  128, new byte[0], Strings.toByteArray("Email Signature"));
```

creates a `CSHAKEDigest` based on SHAKE128 with N empty and S represented by an octet string created from the ASCII encoding of the string “Email Signature”. On the other hand:

```
CSHAKEDigest cshake = new CSHAKEDigest(256, new byte[0], new byte[0]);
```

is the direct equivalent of SHAKE256.

KMAC, TupleHash, and ParallelHash

KMAC, TupleHash, and ParallelHash are all examples of cSHAKE where NIST has standardised the N value.

KMAC, represented by the class `KMAC` in the low-level APIs uses an octet string based on the ASCII bytes for the string “KMAC” as its N value. As its name implies, it is a MAC function so the low-level version in the package `org.bouncycastle.crypto.macs` and it can be initialised with a key. In the JCA it is available as “KMAC128”, and “KMAC256”.

TupleHash, represented by the class `TupleHash` in the low-level APIs uses an octet string based on the ASCII bytes for the string “TupleHash” as its N value. As its name implies, it is a digest function so the low-level version in the package `org.bouncycastle.crypto.digests`. TupleHash is an interesting function where you want the output of the hash to reflect not only the data that was entered, but how it was entered. For example, feeding [“abc”, “def”, “ghi”] into TupleHash will produce a different result to feeding in [“abcdef”, “ghi”]. The extra update makes the difference. In the JCA TupleHash is available as “TupleHash128”, and “TupleHash256”.

ParallelHash, represented by the class `ParallelHash` in the low-level APIs uses an octet string based on the ASCII bytes for the string “ParallelHash” as its N value. It is also a digest function so the low-level version in the package `org.bouncycastle.crypto.digests`. ParallelHash is designed to be executed in a multi-threaded fashion and also takes a block size B as a parameter. At the moment we just simulate this in Bouncy Castle so the thread count is actually one. In the JCA it is available as “ParallelHash128”, and “ParallelHash256” with a blocking size of 1024 bits.

⚠ Note: in all cases the JCA implementations of SP 800-185 XOFs assume an empty customization string.

Bouncy Castle Calculator Interfaces for Digests and MACs

Bouncy Castle’s PKIX package provides general interfaces for implementing a calculators for message digests and MACs. The interfaces are stream based and provide methods for retrieving the ASN.1 AlgorithmParameters associated with the algorithm concerned, an output stream to write the data to be processed to, as well as a method for retrieving the calculated digest/MAC when the stream has been closed.

These interfaces are used throughout the PKIX APIs for introducing message digest and MAC operations implementations for this interface exist both for the JCA and the BC low-level APIs. You can also write your own backed by other cryptography APIs if you need to.

The DigestCalculator Interface

The `DigestCalculator` interface is given below:

```
/**
 * General interface for an operator that is able to calculate a digest from
 * a stream of output.
 */
public interface DigestCalculator
{
    /**
     * Return the algorithm identifier representing the digest implemented by
     * this calculator.
     *
     * @return algorithm id and parameters.
     */
    AlgorithmIdentifier getAlgorithmIdentifier();

    /**
     * Returns a stream that will accept data for the purpose of calculating
     * a digest. Use org.bouncycastle.util.io.TeeOutputStream if you want to
     * accumulate the data on the fly as well.
     *
     * @return an OutputStream
     */
    OutputStream getOutputStream();

    /**
     * Return the digest calculated on what has been written to the calculator's
     * output stream.
     *
     * @return a digest.
     */
    byte[] getDigest();
}
```

Normally you would create one of these from a `DigestCalculatorProvider` and in a situation where a high-level PKIX class needs to create a `DigestCalculator` you will normally find a method taking a `DigestCalculatorProvider` as an argument. In cases where you are using a cryptography API not directly support by Bouncy Castle already, such as an API specific to a HSM, and you need to be able to make use of it to calculate message digests as part of using the Bouncy Castle PKIX APIs you would normally implement the `DigestCalculatorProvider` interface to create a provider for your implementation of the digest calculator.

The `DigestCalculatorProvider` interface is also based on the AlgorithmIdentifier class and has one method.

```
/**
 * The base interface for a provider of DigestCalculator implementations.
 */
public interface DigestCalculatorProvider
{
    DigestCalculator get(AlgorithmIdentifier digestAlgorithmIdentifier)
        throws OperatorCreationException;
}
```

Assuming you do not need to provide your own implementation for one of these, if you are working with a JCA provider you would normally use the JcaDigestCalculatorProviderBuilder to create a DigestCalculatorProvider. You can use the DefaultDigestAlgorithmIdentifierFinder to resolve most common message digest names with their associated AlgorithmIdentifier structures as well. For example, when creating a SHA-256 message digest calculator using the BC provider, it would look something like this:

```
  /**
   * Return a DigestCalculator for the passed in algorithm digestName.
   *
   * @param digestName the name of the digest algorithm.
   * @return a DigestCalculator for the digestName.
   */
  public static DigestCalculator createDigestCalculator(String digestName)
      throws OperatorCreationException
  {
      DigestAlgorithmIdentifierFinder algFinder =
                                new DefaultDigestAlgorithmIdentifierFinder();
      DigestCalculatorProvider        digCalcProv =
          new JcaDigestCalculatorProviderBuilder().setProvider("BC").build();

      return digCalcProv.get(algFinder.find(digestName));
  }
```

The argument to createDigestCalculator - digestName is passed in as “SHA-256”. For a contrived example like this, it does look more complicated than MessageDigest.getInstance("SHA-256"), however in the context of the requirements of the BC PKIX APIs this approach actually makes life a lot easier.

Using a DigestCalculator is a matter of getting the output stream and writing the information to be digested to the output stream. For example, a reworking of our original digest example for SHA-256 would look as follows:

```
/**
 * Creation and use of a SHA-256 DigestCalculator.
 */
public class DigestCalculatorExample
{
    public static void main(String[] args)
        throws Exception
    {
        DigestCalculator digCalc = createDigestCalculator("SHA-256");

        OutputStream dOut = digCalc.getOutputStream();

        dOut.write(Strings.toByteArray("Hello World!"));

        dOut.close();

        System.out.println(Hex.toHexString(digCalc.getDigest()));
    }
}
```

If you run the above example you will find it produces the same result as the original SHA-256 example using `MessageDigest` that we looked at earlier.

You can find further examples of implementations of the `DigestCalculatorProvider` interface and `DigestCalculator` interface in the `org.bouncycastle.operator.bc` package. These implementation do not make any use of the JCA and can be useful as a guide for implementing your own versions of the interfaces.

The MacCalculator Interface

The `MacCalculator` interface is given below:

```
/**
 * General interface for a key initialized operator that is able to calculate a
 * MAC from a stream of output.
 */
public interface MacCalculator
{
    AlgorithmIdentifier getAlgorithmIdentifier();

    /**
     * Returns a stream that will accept data for the purpose of calculating
     * the MAC for later verification. If you want to accumulate the data on
     * the fly as well use org.bouncycastle.util.io.TeeOutputStream.
     *
     * @return an OutputStream
     */
    OutputStream getOutputStream();

    /**
     * Return the calculated MAC based on what has been written to the stream.
     *
     * @return calculated MAC.
     */
    byte[] getMac();


    /**
     * Return the key used for calculating the MAC.
     *
     * @return the MAC key.
     */
    GenericKey getKey();
}
```

The use of `MacCalculator` tends to be more specialised in the PKIX APIs so rather than having one provider class for them, there are specialised builders and providers for each area such as for CMS, represented by `JceCMSMacCalculatorBuilder` and PKCS #12, represented by `JcePKCS12MacCalculatorBuilderProvider`.

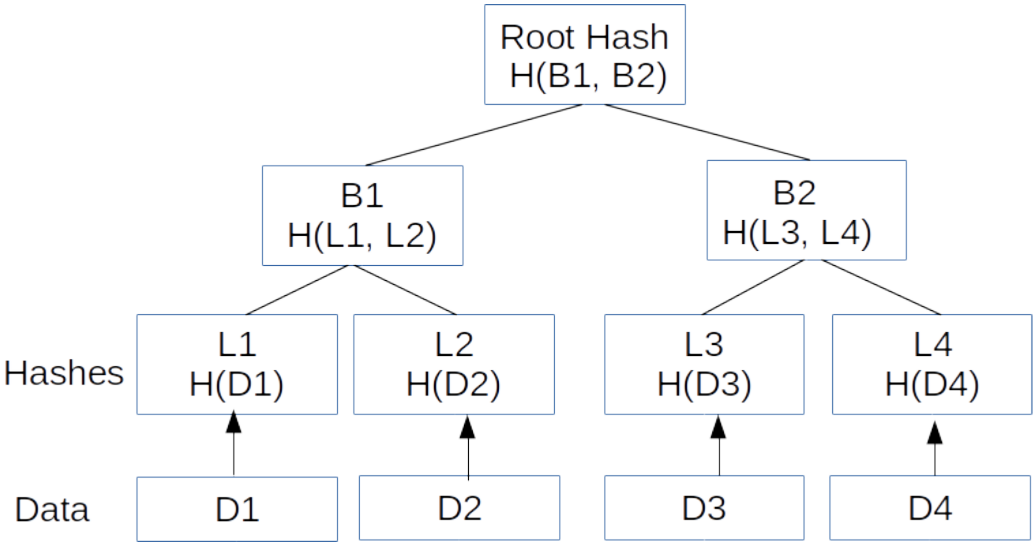
Once created, a `MacCalculator` is used in the same manner as a `DigestCalculator`. You can also find further examples of implementations of `MacCalculator` in some of the the `.bc.` packages associated with different PKIX protocols. These implementation do not make any use of the JCA and can be useful as a guide for implementing your own versions of the interfaces.

An Application of Digests: Merkle Trees

A good example of just how much of a work horse message digests are is the Merkle Tree [124]. Merkle trees are an idea originally conceived by Ralph Merkle around 1979. They have become a lot better known recently largely due to BitCoin, but have also seen regular use in other applications and, for example, appear in RFC 6962 [88] and also, more recently, RFC 7574 [96].

What Are Merkle Trees?

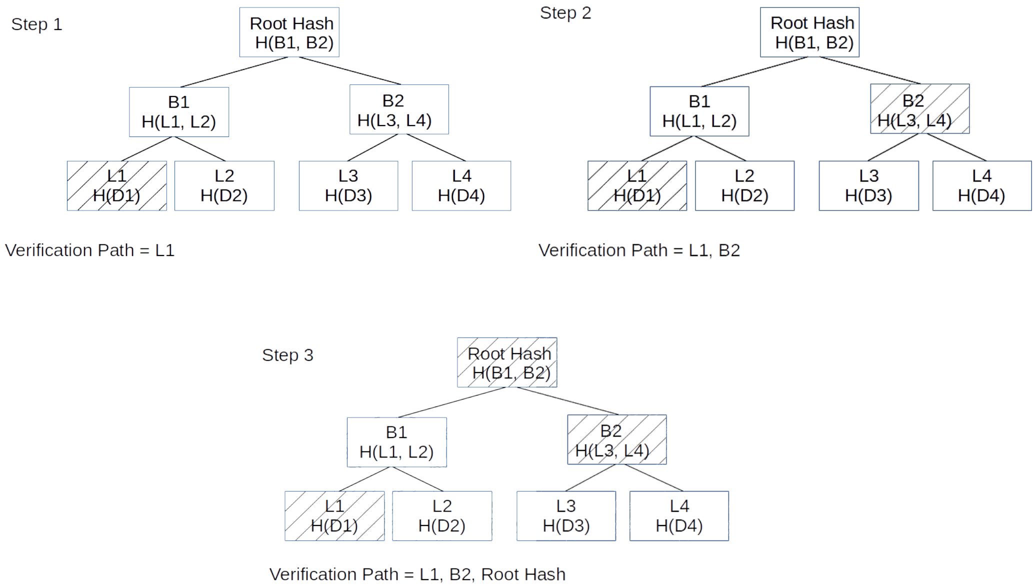
Merkle trees are trees of message digests, with leaf node digests representing some data and the branch node digests representing a digest constructed from the message digests of the nodes below it.



A Basic Merkle Tree - Data is used to generate the leaf node hashes.

As you can see from the diagram the root hash, in a sense represents a hash of the tree, and as such a hash of the entirety of the data currently stored in the tree. This is one of the things that make Merkle trees useful, regardless of how many data values there are you can use a Merkle tree to reduce them to a single hash and if you need to add another data value later, there is no need to re-compute all the original hashes. You just need to calculate the hash for the new data value and then recalculate the branch nodes affected by the insertion of the data value back up to the root hash.

This leads us to another useful feature of the Merkle tree, you can generate what is called an *audit path* as a proof that a data value has been added to the tree. We can see how an audit path is generated in the diagram below.



The steps involved in generating an audit path as proof of the presence of the value D2.

As you can see it is a matter of collecting the hashes for the branches related to the path to the data. The reason for doing this is once you calculate those and a hash of the data you are interested in proving as been in the tree you can recalculate the root hash value. If the root hash value calculation gives you the same hash as the actual root hash for the tree, you have confirmed that the audit path shows that the value you are interested in is actually in the the tree.

The best approach to implementing a Merkle tree is to use a balanced tree algorithm - the main reason being that it will keep the audit paths shortest and, consequently, provide the cheapest proofs to evaluate. The other thing about a Merkle tree is it also provides a cautionary tale of what can happen if you simply concatenate hashes to feed into a hash function as you can find yourself vulnerable to a second preimage attack. We won't go into the details here, however from the point of view of a Merkle tree this pans out as if you simply hash the data using a regular hash function, and someone can fool you into accepting an actual hash value into the tree the tree will then have a root that might be able to be used to prove a document was inserted into it when it was not.

Fortunately this situation is quite easy to avoid by adding a prefix to the data before it is hashed, and using a different prefix for the branch node hashes - this is the approach used in RFC 6962.

The MerkleTree class

The following code implements a MerkleTree using an AVL+ tree, which is a variation of the balanced AVL tree that stores the inserted data in the leaf nodes. You will note that the tree implementation provides second preimage resistance by making use of prefixes on the hash calculations, in the same fashion as RFC 6962. To be able to make use of some higher proofs of security around Merkle trees you would also need to modify this implementation to limit the tree size.

Here is the code:

```java
/**
 * Implementation of a Merkle Tree based on an AVL+ tree (where data is always
 * stored in the leaf node).
 */
public class MerkleTree
{
    // Base class for a tree node.
    private class Node
    {
        byte[] keyValue;
        byte[] hash;

        // Return true if the keyValue in this node is
        // greater than the keyValue in other.
        boolean isGreaterThan(Node other)
        {
            for (int i = 0; i != keyValue.length; i++)
            {
                int thisB = this.keyValue[i] & 0xff;
                int otherB = other.keyValue[i] & 0xff;
                if (thisB > otherB)
                {
                    return true;
                }
                else if (thisB < otherB)
                {
                    return false;
                }
            }

            return false;
        }
    }

    // the leaf node.
    private class LeafNode
        extends Node
    {
        final byte[] nodeValue;

        LeafNode(byte[] nodeHash, byte[] nodeValue)
        {
            this.keyValue = nodeHash;
            this.nodeValue = nodeValue;
            this.hash = nodeHash;
        }
    }

    // the branch node.
    private class BranchNode
        extends Node
    {
        Node left, right;

        BranchNode(Node left, Node right)
        {
            this.left = left;
            this.right = right;
            this.recalculateHash();
            this.recalculateKeyValue();
        }

        void recalculateKeyValue()
        {
            this.keyValue = findMaxValue(left);
        }

        void recalculateHash()
        {
            this.hash = calculateBranchHash(left, right);
        }

        int getBalance()
        {
            return getHeight(this.left) - getHeight(this.right);
        }

        private int getHeight(Node root)
        {
            if (root instanceof BranchNode)
            {
                return 1 + Math.max(
                    getHeight(((BranchNode)root).left),
                    getHeight(((BranchNode)root).right));
            }

            return 0;
        }

        private byte[] findMaxValue(Node root)
        {
            if (root instanceof BranchNode)
            {
                return findMaxValue(((BranchNode)root).right);
            }
            else
            {
                return ((LeafNode)root).keyValue;
            }
        }
    }

    private final MessageDigest digest;
    private final byte[] emptyHash;

    Node root;

    /**
     * Base Constructor
     *
     * @param digest the digest to generate the tree's hashes with.
     */
    public MerkleTree(MessageDigest digest)
    {
        this.digest = digest;
        this.emptyHash = digest.digest();
    }

    /**
     * Insert value into the Merkle tree.
     *
     * @param value the value to be inserted.
     */
    public void insert(byte[] value)
    {
        LeafNode leafNode = createLeafNode(Arrays.clone(value));

        if (root == null)
        {
            root = leafNode;
        }
        else
        {
            root = insertLeaf(root, leafNode);
        }
    }

    /**
     * Return an AuditPath leading to value if it is in the tree. Null otherwise.
     *
     * @param value the value we want the audit path for.
     * @return an AuditPath, or null if value is not present.
     */
    public AuditPath generateAuditPath(byte[] value)
    {
```

```java
        LeafNode leafNode = createLeafNode(Arrays.clone(value));

        if (root == null)
        {
            return null;
        }
        else
        {
            List pathElements = new ArrayList();

            boolean found = buildAuditPath(pathElements, root, leafNode);

            if (found)
            {
                return new AuditPath(root.hash, pathElements);
            }

            return null;
        }
    }

    private LeafNode createLeafNode(byte[] nodeValue)
    {
        // The zero prefix is to make sure we use a different calculation
        // from a branch node. See section 2.1 of RFC 6962 for further details.
        return new LeafNode(hash((byte)0x00, nodeValue), nodeValue);
    }

    private boolean buildAuditPath(
        List<AuditPath.Element> proof, Node treeNode, LeafNode newNode)
    {
        if (newNode.isGreaterThan(treeNode))
        {
            if (treeNode instanceof BranchNode)
            {
                BranchNode branch = (BranchNode)treeNode;

                if (buildAuditPath(proof, branch.right, newNode)) // found
                {
                    proof.add(new AuditPath.Element(true, getHash(branch.left)));

                    return true;
                }
            }

            return false;
        }
        else
        {
            if (treeNode instanceof BranchNode)
            {
                BranchNode branch = (BranchNode)treeNode;

                if (buildAuditPath(proof, branch.left, newNode)) // found
                {
                    proof.add(new AuditPath.Element(false, getHash(branch.right)));

                    return true;
                }

                return false;
            }
            else
            {
                return Arrays.constantTimeAreEqual(treeNode.hash, newNode.hash);
            }
        }
    }

    private byte[] getHash(Node node)
    {
        if (node == null)
        {
            return emptyHash;
        }
        else
        {
            return node.hash;
        }
    }

    // the 01 prefix is to make sure we use a different calculation
    // from a leaf node. See section 2.1 of RFC 6962 for further details.
    private byte[] calculateBranchHash(Node left, Node right)
    {
        byte[] leftHash = (left != null) ? left.hash : emptyHash;
        byte[] rightHash = (right != null) ? right.hash : emptyHash;

        return hash((byte)0x01, Arrays.concatenate(leftHash, rightHash));
    }

    private byte[] hash(byte preFix, byte[] value)
    {
        digest.update(preFix);
        digest.update(value);

        return digest.digest();
    }

    // balancing action - left rotate
    private BranchNode rotateLeft(BranchNode n)
    {
        n.left = new BranchNode(n.left, ((BranchNode)n.right).left);
        n.right = ((BranchNode)n.right).right;

        balance((BranchNode)n.left);

        n.recalculateKeyValue();
        n.recalculateHash();

        return n;
    }

    // balancing action - right rotate
    private BranchNode rotateRight(BranchNode n)
    {
        n.right = new BranchNode(((BranchNode)n.left).right, n.right);
        n.left = ((BranchNode)n.left).left;

        balance((BranchNode)n.right);

        n.recalculateKeyValue();
        n.recalculateHash();

        return n;
    }

    private Node insertLeaf(Node treeNode, LeafNode newNode)
    {
        if (treeNode == null)
        {
            return newNode;
        }
        else if (treeNode.isGreaterThan(newNode))
        {
            if (treeNode instanceof BranchNode)
            {
                BranchNode branch = (BranchNode)treeNode;

                branch.left = insertLeaf(branch.left, newNode);

                branch.recalculateKeyValue();
                branch.recalculateHash();

                return balance(branch);
            }
            else
            {
                return new BranchNode(newNode, treeNode);
            }
        }
        else
        {
            if (treeNode instanceof BranchNode)
            {
                BranchNode branch = (BranchNode)treeNode;

                branch.right = insertLeaf(branch.right, newNode);

                branch.recalculateKeyValue();
                branch.recalculateHash();

                return balance(branch);
            }
            else
            {
                return new BranchNode(treeNode, newNode);
            }
        }
    }

    // Check BranchNode to see if it's as balanced as we can make it.
    private BranchNode balance(BranchNode node)
    {
        int balance = node.getBalance();

        // left side is higher
        if (balance == 1)
        {
            return rotateRight(node);
        }

        // right side is higher
        if (balance == -1)
        {
            return rotateLeft(node);
        }

        // left right probably required
        if (balance >= 2)
        {
            if (((BranchNode)node.left).right instanceof LeafNode)
            {
                node.left = rotateRight((BranchNode)node.left);

                return node;
            }
            else
            {
                node.left = rotateLeft((BranchNode)node.left);

                return rotateRight(node);
            }
        }

        // right left probably required
        if (balance <= -2)
        {
            if (((BranchNode)node.right).left instanceof LeafNode)
            {
                node.right = rotateLeft((BranchNode)node.right);

                return node;
            }
            else
            {
                node.right = rotateRight((BranchNode)node.right);

                return rotateLeft(node);
            }
        }

        return node;
    }

    /**
     * Return a dump of the tree - inorder traversal.
     *
     * @return a String dump of the tree.
     */
    public String toString()
    {
        StringBuilder sBld = new StringBuilder();

        toString(sBld, root);

        return sBld.toString();
    }

    private void toString(StringBuilder sBld, Node root)
    {
        if (root instanceof BranchNode)
        {
            sBld.append("(N[");
            sBld.append(new BigInteger(1, root.keyValue).toString(16));
            sBld.append("]: ");
            toString(sBld, ((BranchNode)root).left);
            sBld.append(", ");
            toString(sBld, ((BranchNode)root).right);
            sBld.append(")");
        }
        else
        {
            sBld.append("(L[");
            sBld.append(new BigInteger(1, root.keyValue).toString(16));
            sBld.append("]: ");
            sBld.append(new BigInteger(1, ((LeafNode)root).nodeValue).toString());
            sBld.append(")");
        }
    }
}
```

The AuditPath class

Hopefully you will have noticed the `generateAuditPath()` method in `MerkleTree` which returns an `AuditPath`. The `AuditPath` class provides a carrier for the hashes that constitute a proof that a particular value was stored in a tree. The full evaluation of the path to see if it matches with some possible data is done in the `isMatched()` method.

```
/**
 * AuditPath carrier created by the MerkleTree class to provide proof of presence for
 * a value in the tree.
 */
public class AuditPath
{
    /**
     * A carrier for a node hash, the combine() method ensures the hash
     * is added correctly to the hash it is being combined with on evaluation.
     */
    public static class Element
    {
        byte[] value;
        boolean isLeft;

        Element(boolean isLeft, byte[] value)
        {
            this.isLeft = isLeft;
            this.value = value;
        }

        /**
         * Return this element's hash combined with the passed in hash
         * in a format suitable for passing into a digest function.
         *
         * @param hash the hash to be combined with this element.
         * @return a concatenation of the element hash and the passed in hash.
         */
        public byte[] combine(byte[] hash)
        {
            if (isLeft)
            {
                return Arrays.concatenate(new byte[]{1}, value, hash);
            }
            else
            {
                return Arrays.concatenate(new byte[]{1}, hash, value);
            }
        }
    }

    private final byte[] rootHash;
    private final List<Element> elements;

    AuditPath(byte[] rootHash, List<Element> elements)
    {
        this.rootHash = rootHash;
        this.elements = Collections.unmodifiableList(elements);
    }

    /**
     * Return the root hash for the tree this audit path is from.
     *
     * @return the root hash for the tree.
     */
    public byte[] getRootHash()
    {
        return Arrays.clone(rootHash);
    }

    /**
     * Return the list representing the elements of the
     * audit path leading to the root of the tree.
     *
     * @return a list of hash elements.
     */
    public List<Element> getElements()
    {
        return elements;
    }

    /**
     * Return true if the passed in data value is the one matched
     * by the audit path.
     *
     * @param digest the digest to use for hash calculations
     * @param data the value that is being checked for.
     * @return true if the audit path is for the data, false otherwise.
     */
    public boolean isMatched(MessageDigest digest, byte[] data)
    {
        digest.update((byte)0x00);
        digest.update(data, 0, data.length);

        byte[] dig = digest.digest();

        for (AuditPath.Element element : elements)
        {
            dig = digest.digest(element.combine(dig));
        }

        return Arrays.constantTimeAreEqual(getRootHash(), dig);
    }
}
```

You can see from the `isMatched()` method, that despite the size of the originating Merkle tree, an audit path only contains a few hashes. Those few hashes are enough to allow you to confirm a value was added to the tree as they provide enough data to recalculate the tree's root hash when combined with the hash of the value of interest.

Trying the Merkle Tree out

Having come this far, it would be nice to actually try using the class. The following code fragment will populate the tree with just over 2000 numbers and then see if it can find a proof for that a particular value was added to the tree by generating an audit path for it.

```
        MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

        MerkleTree tree = new MerkleTree(sha256);

        // populate the tree with data.
        for (int i = 0; i != 1000; i += 2)
        {
            tree.insert(BigIntegers.asUnsignedByteArray(BigInteger.valueOf(i)));
        }

        for (int i = 1001; i > 0; i -= 2)
        {
            tree.insert(BigIntegers.asUnsignedByteArray(BigInteger.valueOf(i)));
        }

        // generate an audit path for a value of interest.
        byte[] value = BigIntegers.asUnsignedByteArray(BigInteger.valueOf(239));

        AuditPath path = tree.generateAuditPath(value);

        System.out.println("Value on path: " + path.isMatched(sha256, value));

        // try using the path to match a different value.
        value = BigIntegers.asUnsignedByteArray(BigInteger.valueOf(100));

        System.out.println("Value on path: " + path.isMatched(sha256, value));
```

If everything has worked correctly, the sample code will produce the output:

```
Value on path: true
Value on path: false
```

Showing that the first number, 239, is both in the tree and was the one the audit path was generated for. On the other hand, 100, the second number checked with the audit path, while having been stored in the tree, is not the number the audit path is for so fails to match.

Summary

In this chapter we have examined the variety of approaches for making our messages tamper evident using Message Digests, MACs, and HMACs. We also looked at how the same algorithms could be re-purposed for use in key derivation functions.

We also looked at the interfaces provided in the Bouncy Castle PKIX package for dealing with Digests and MACs as well as naming the classes that produce default implementations of them. These interfaces were flagged as important because, as well as providing the way of providing implementations of Digest and MAC operators to the different PKIX classes, they also provide a way of introducing alternate implementations using other providers and cryptography APIs such as those based on HSMs.

Finally, we had a look at a general example of the application of message digest functions, the Merkle Tree.

Chapter 4: Authenticated Modes, Key Wrapping, and the SealedObject

This chapter introduces authenticated modes, a variant of symmetric key encryption which includes a cryptographic checksum in the data stream so the correctness of decrypted data can be determined by the party decrypting it. We will also look at modes for doing key wrapping, that is encrypting symmetric and private keys. Finally we will take a look at a simple way of encrypting a serializable object using the Java provided `SealedObject` class.

Setup for the Examples

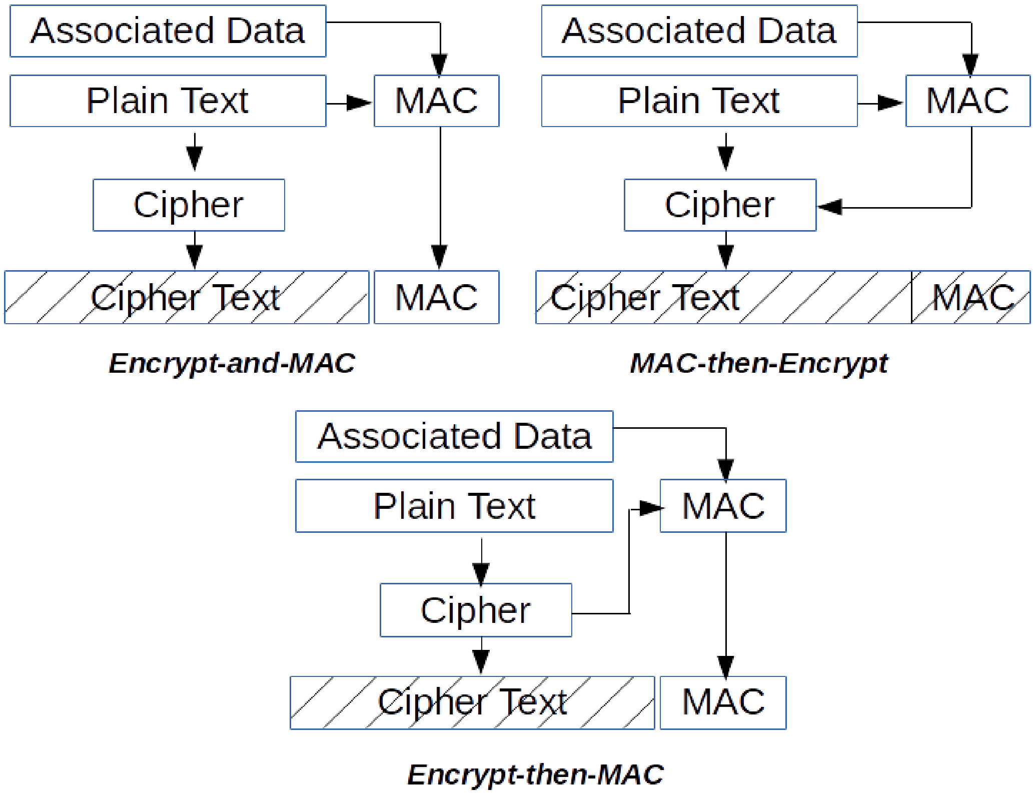
The example make use of the following method for generating a constant value 128 bit AES key:

```
/**
 * Create a constant value AES key.
 *
 * @return a constant AES key.
 */
static SecretKey createConstantKey()
{
    return new SecretKeySpec(
                   Hex.decode("000102030405060708090a0b0c0d0e0f"), "AES");
}
```

We are avoiding the `KeyGenerator` class as we want the output of and behaviour of the examples we are looking at below to be easy to predict.

Authenticated Encryption Modes

Authenticated encryption modes provide modes which allow a stream of encrypted data to also be verified, either prior to decryption, or after decryption as being likely to be an accurate representation of the data that was originally encrypted. The check on the data is done using a MAC which is verified by the party decrypting the cipher text. The MAC is not restricted to being just related to the encrypted plain text either, any of the authenticated data modes can be also be extended to take "associated data" into the calculation, with the MAC including the "associated data" as well. It is because of this extendability that you will see these modes referred to both as AE (Authenticated Encryption) modes, and also AEAD (Authenticated Encryption with Associated Data) modes.



Encrypt-and-MAC

MAC-then-Encrypt

Encrypt-then-MAC

The Basic Types of Authenticated Encryption Modes

Bellare and Namprempre [111] defined three basic types of authenticated encryption modes. Briefly the basic types are as follows.

Encrypt-and-MAC
: In this case, the MAC is calculated on the plain text and just the plain text is encrypted. The resulting cipher text and the MAC are then sent to the recipient. As described previously, associated data might also be involved. The difference with this approach from *Encrypt-then-MAC* is the recipient has to decrypt the cipher text first before being able to check the MAC.

MAC-then-Encrypt
: Here a MAC is calculated on the plain text, both the plain text and the MAC are encrypted and the resulting cipher text is sent to the recipient (with any associated data). This approach also requires decrypting the cipher text to verify the MAC.

Encrypt-then-MAC
: In this last case, the data is encrypted first and then the MAC is calculated on the associated cipher text and, if present, any associated data. The cipher text, associated data, and MAC are then sent to the recipient, who can often verify the first two without having to recover any of the original plain text.

Keeping the definitions in mind we will now look at some actual implementations.

GCM

Galois/Counter Mode is defined in NIST SP 800-38D [7] and uses the *Encrypt-then-MAC* approach. It can be used with associated data and GMAC is actually an associated data only version of GCM.

The following example shows how to use GCM with encryption.

```
/**
 * Encrypt the passed in data pText using GCM with the passed in parameters.
 *
 * @param key secret key to use.
 * @param iv the IV to use with GCM.
 * @param tagLen the length of the MAC to be generated by GCM.
 * @param pText the plain text input to the cipher.
 * @return the cipher text.
 */
static byte[] gcmEncrypt(SecretKey key,
                         byte[] iv,
                         int    tagLen,
                         byte[] pText)
    throws Exception
{
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");

    GCMParameterSpec spec = new GCMParameterSpec(tagLen, iv);

    cipher.init(Cipher.ENCRYPT_MODE, key, spec);

    return cipher.doFinal(pText);
}
```

As you can see in the code sample GCM also has its own parameter spec - `GCMParameterSpec` - which includes an IV and a tag length. With GCM an IV can be of arbitrary length, but it is very important that it is unique for a given key as the underlying cipher mode in GCM is CTR, and so reuse of the IV will result in the same key stream being generated for an encryption. The tag length is used to specify the size of the MAC to use - there is guidance on MAC sizes included in Appendix C of NIST SP 800-38D, but the general rule of thumb, as expressed by Niels Ferguson in “Authentication weaknesses in GCM” [114] is to use the maximum tag size for the underlying GHASH algorithm, which is used to calculate the MAC.

In the example we are not using associated data either. So other than the presence of the `GCMParameterSpec` both the encryption step and the decryption step below look very similar to what we saw with symmetric ciphers in Chapter 2.

```
/**
 * Decrypt the cipher text cText using the passed in key and other
 * parameters.
 *
 * @param key secret key to use.
 * @param iv the IV to use with GCM.
 * @param tagLen the length of the MAC previously generated by GCM.
 * @param cText the encrypted cipher text.
 * @return the original plain text.
 */
static byte[] gcmDecrypt(SecretKey key,
                          byte[] iv,
                          int    tagLen,
                          byte[] cText)
    throws Exception
{
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");

    GCMParameterSpec spec = new GCMParameterSpec(tagLen, iv);

    cipher.init(Cipher.DECRYPT_MODE, key, spec);

    return cipher.doFinal(cText);
}
```

The following code fragment puts the encryption and decryption steps together. Note that the tag length has been set the maximum available value of 128.

```
/**
 * A simple GCM example without Additional Associated Data (AAD)
 */
public class GCMExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        byte[] iv = Hex.decode("bbaa99887766554433221100");
        byte[] msg = Strings.toByteArray("hello, world!");

        System.out.println("msg  : " + Hex.toHexString(msg));

        byte[] cText = gcmEncrypt(aesKey, iv, 128, msg);

        System.out.println("cText: " + Hex.toHexString(cText));

        byte[] pText = gcmDecrypt(aesKey, iv, 128, cText);

        System.out.println("pText: " + Hex.toHexString(pText));
    }
}
```

If you run the example and all goes well, you should see the following output:

```
msg  : 68656c6c6f2c20776f726c6421
cText: 4024f07c59c758d2d0c7d09934e2fdfe12c72351c5ff0e4c9e47785299
pText: 68656c6c6f2c20776f726c6421
```

As we would expect, the cipher text is longer than the plain text, this time due to the addition of the MAC, not some underlying padding mechanism as we saw in Chapter 2.

A Look at Error Detection

A good question to ask at this point is what happens if an implementation of a cipher encounters a tag failure as the MAC is no longer valid. The following code gives a simple example which corrupts the cipher text before attempting to decrypt it.

```
/**
 * A simple GCM example that shows data corruption.
 */
public class GCMFailExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        byte[] iv = Hex.decode("bbaa99887766554433221100");
        byte[] msg = Strings.toByteArray("hello, world!");

        byte[] cText = gcmEncrypt(aesKey, iv, 128, msg);

        // tamper with the cipher text
        cText[0] = (byte)~cText[0];

        byte[] pText = gcmDecrypt(aesKey, iv, 128, cText);
    }
}
```

Running the example will throw an exception and, for Java 7 or later, you will see a stack trace starting with the following line:

```
Exception in thread "main" javax.crypto.AEADBadTagException: mac check in GCM failed
```

On earlier JVMs, if you are using a provider such as Bouncy Castle that supports GCM, you will probably see something like:

```
Exception in thread "main" javax.crypto.BadPaddingException: mac check in GCM failed
```

The `AEADBadTagException` exception is an extension of `BadPaddingException` which `Cipher.doFinal()` lists as throwing. You only need to catch `AEADBadTagException` specifically if you have a particular requirement to.

Keep in mind too that if the tag calculation does fail to verify the message, baring errors in setup, there is most likely something wrong with the message. It is important to make sure anyone consuming the data you are verifying is aware of this.

> ⚠ The tag on an AEAD encrypted message is there for a reason. If a tag fails to validate an AEAD encrypted message make sure any data involved in the tag calculation or decrypted from the failing message is not presented to a user in a way that may have them conclude it is valid data.

Parameter Generation

As with modes such as CBC which require IVs, there are also implementations of `AlgorithmParameterGenerator` and `AlgorithmParameters` for GCM available in Bouncy Castle which can be created using the `getInstance()` methods available on both classes.

The following example shows how to create an `AlgorithmParameters` object for a GCM cipher using an `AlgorithmParameterGenerator`.

```java
/**
 * A GCM example showing the use of AlgorithmParameterGenerator
 */
public class GCMWithParamGenExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        AlgorithmParameterGenerator pGen =
            AlgorithmParameterGenerator.getInstance("GCM","BC");

        byte[] msg = Strings.toByteArray("hello, world!");

        System.out.println("msg  : " + Hex.toHexString(msg));

        AlgorithmParameters pGCM = pGen.generateParameters();

        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");

        cipher.init(Cipher.ENCRYPT_MODE, aesKey, pGCM);

        byte[] cText = cipher.doFinal(msg);

        System.out.println("cText: " + Hex.toHexString(cText));

        GCMParameterSpec gcmSpec = pGCM.getParameterSpec(GCMParameterSpec.class);

        byte[] pText = gcmDecrypt(
                                aesKey, gcmSpec.getIV(), gcmSpec.getTLen(), cText);

        System.out.println("pText: " + Hex.toHexString(pText));
    }
}
```

Unlike the previous example the IV used with GCM is now random, so running the sample will result in different output being produced.

The GCM parameters also have an ASN.1 definition, defined in RFC 5084 [66] as:

```
 GCMParameters ::= SEQUENCE {
   aes-nonce        OCTET STRING, -- recommended size is 12 octets
   aes-ICVlen       AES-GCM-ICVlen DEFAULT 12 }

AES-GCM-ICVlen ::= INTEGER (12 | 13 | 14 | 15 | 16)
```

Note that the ICVlen is in octets, not bits as with the `GCMParameterSpec`, and also remember that in this context DEFAULT refers to when a value should not be encoded in DER, not what the actual default value would be. As we have already mentioned, its best to use a GCM tag length of 128 bits (or 16 octets from an ASN.1 perspective). While the structure is specifically defined for use with AES, JCE providers generally allow the same format to be used with other ciphers with a 128 bit block-size.

You can create a copy of the ASN.1 structure, by using `pGCM.getEncoded()`, which by default will return an ASN.1 encoding, or you can specify it fully, and pass the resulting byte array to the `GCMParameters.getInstance()` as follows:

```
GCMParameters asn1Params = GCMParameters.getInstance(pGCM.getEncoded("ASN.1"));
```

Likewise an `AlgorithmParameters` implementation can be initialised with the encoding of the ASN.1 structure `asn1Params` by calling the `AlgorithmParameters.init()` method as in:

```
      AlgorithmParameters pGCM = AlgorithmParameters.getInstance("GCM", "BC");

      pGCM.init(asn1Params.getEncoded());
```

Using Associated Data

So far our examples have not included any associated data. Associated data is data which is not encrypted but is still used in the calculation of the MAC that provides the tag that appears at the end of the encrypted data. From Java 7 the `Cipher` class also includes methods for adding associated data with AEAD ciphers.

The encrypt example following shows how to include associated data with the `Cipher.updateAAD()` method made available in Java 7.

```java
/**
 * Encrypt the passed in data pText using GCM with the passed in parameters
 * and incorporating aData into the GCM MAC calculation.
 *
 * @param key secret key to use.
 * @param iv the IV to use with GCM.
 * @param pText the plain text input to the cipher.
 * @param aData the associated data to be included in the GCM MAC.
 * @return the cipher text.
 */
static byte[] gcmEncryptWithAAD(SecretKey key,
                                byte[] iv,
                                byte[] pText,
                                byte[] aData)
    throws Exception
{
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");

    GCMParameterSpec spec = new GCMParameterSpec(128, iv);

    cipher.init(Cipher.ENCRYPT_MODE, key, spec);

    cipher.updateAAD(aData);

    return cipher.doFinal(pText);
}
```

Bouncy Castle also provides an algorithm parameter specification class that allows passing of associated data: the `AEADParameterSpec`. While it is not as flexible as the approach taken in Java 7, it is the only JCE based mechanism available in Java 6 and earlier. The GCM decrypt code below shows the equivalent for adding the associated data using the `AEADParameterSpec` class instead.

```java
/**
 * Decrypt the passed in cipher text cText using GCM with the passed in
 * parameters and incorporating aData into the GCM MAC calculation.
 *
 * @param key secret key to use.
 * @param iv the IV originally used with GCM.
 * @param cText the encrypted cipher text.
 * @param aData the associated data to be included in the GCM MAC.
 * @return the plain text.
 */
static byte[] gcmDecryptWithAAD(SecretKey key,
                          byte[] iv,
                          byte[] cText,
                          byte[] aData)
    throws Exception
{
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");

    AEADParameterSpec spec = new AEADParameterSpec(iv, 128, aData);

    cipher.init(Cipher.DECRYPT_MODE, key, spec);

    return cipher.doFinal(cText);
}
```

Porting requirements aside, where the `updateAAD()` methods are available it is better to use those. You can use the following example to see the two approaches working together.

```
/**
 * A simple GCM example with Additional Associated Data (AAD)
 */
public class GCMWithAADExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        byte[] iv = Hex.decode("bbaa99887766554433221100");
        byte[] msg = Strings.toByteArray("hello, world!");
        byte[] aad = Strings.toByteArray("now is the time!");

        System.out.println("msg  : " + Hex.toHexString(msg));

        byte[] cText = gcmEncryptWithAAD(aesKey, iv, msg, aad);

        System.out.println("cText: " + Hex.toHexString(cText));

        byte[] pText = gcmDecryptWithAAD(aesKey, iv, cText, aad);

        System.out.println("pText: " + Hex.toHexString(pText));
    }
}
```

If you run the example you will see the following output:

```
msg  : 68656c6c6f2c20776f726c6421
cText: 4024f07c59c758d2d0c7d09934dfe5a6760d5c1f6d778efc3e48e9d073
pText: 68656c6c6f2c20776f726c6421
```

If you compare this output to the simple GCM example we looked at earlier, you will see the effect of the associated data in the MAC. The initial cipher text, as we would expect has stayed the same as neither the key nor the data to be encrypted have changed. The MAC, on the other hand, shows the effect of adding the associated data. As this is the case if you were to modify the example to corrupt the associated data, you will also find that the `doFinal()` on decrypt will throw an exception as it did when the cipher text was corrupted.

CCM

The standard definition of Counter with CBC/MAC (CCM) is defined in NIST SP 800-38C [6]. It uses the *MAC-then-Encrypt* approach, using the plain text to calculate a CBC-MAC which is then encrypted along with the plain text using a CTR mode cipher. It can also be used with associated data, although its MAC "equivalent" CMAC is not the same as CCM with no plain text.

Like GCM, CCM requires an IV (in this case really a nonce) and a tag length to specify the size of the MAC to include. The following code shows the use of CCM for encryption, other than the cipher specification, it looks identical to the GCM one as the JCE API does not actually define a `CCMParameterSpec` class, so many vendors, including Bouncy Castle simply accept the `GCMParameterSpec` for the same use as you will see below:

```
/**
 * Encrypt the passed in data pText using CCM with the passed in parameters
 * and incorporating aData into the CCM MAC calculation.
 *
 * @param key secret key to use.
 * @param nonce the nonce to use with CCM.
 * @param pText the plain text input to the cipher.
 * @param aData the associated data to process with the plain text.
 * @return the cipher text.
 */
static byte[] ccmEncryptWithAAD(SecretKey key,
                          byte[] nonce,
                          byte[] pText,
                          byte[] aData)
    throws Exception
{
    Cipher cipher = Cipher.getInstance("AES/CCM/NoPadding", "BC");
```

```

        GCMParameterSpec spec = new GCMParameterSpec(128, nonce);

        cipher.init(Cipher.ENCRYPT_MODE, key, spec);

        cipher.updateAAD(aData);

        return cipher.doFinal(pText);
    }
```

In this case we are using a tag length of 128 bits as well. While this is the same before with the GCM example, unlike with GCM where the tag is based on the GHASH function, CCM uses a cipher for calculating the MAC, so the tag length is only limited by the block size of the underlying cipher used.

CCM, in Bouncy Castle, also supports the AEADParameterSpec class, as we can see in the sample decrypt method below:

```
    /**
      * Decrypt the passed in cipher text cText using CCM with the passed in
      * parameters and incorporating aData into the CCM MAC calculation.
      *
      * @param key secret key to use.
      * @param nonce the nonce originally used with CCM.
      * @param cText the encrypted cipher text.
      * @param aData the associated data to be included in the CCM MAC.
      * @return the plain text.
      */
    static byte[] ccmDecryptWithAAD(SecretKey key,
                                    byte[] nonce,
                                    byte[] cText,
                                    byte[] aData)
        throws Exception
    {
        Cipher cipher = Cipher.getInstance("AES/CCM/NoPadding", "BC");

        AEADParameterSpec spec = new AEADParameterSpec(nonce, 128, aData);

        cipher.init(Cipher.DECRYPT_MODE, key, spec);

        return cipher.doFinal(cText);
    }
```

We can put the two methods together using the following sample program.

```java
/**
 * A simple CCM Example with Additional Associated Data (AAD)
 */
public class CCMWithAADExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        byte[] iv = Hex.decode("bbaa99887766554433221100");
        byte[] msg = Strings.toByteArray("hello, world!");
        byte[] aad = Strings.toByteArray("now is the time!");

        System.out.println("msg  : " + Hex.toHexString(msg));

        byte[] cText = ccmEncryptWithAAD(aesKey, iv, msg, aad);

        System.out.println("cText: " + Hex.toHexString(cText));

        byte[] pText = ccmDecryptWithAAD(aesKey, iv, cText, aad);

        System.out.println("pText: " + Hex.toHexString(pText));
    }
}
```

Which should produce the following output:

```
msg  : 68656c6c6f2c20776f726c6421
cText: f659bc8216422ed3a6ae7cf3401b5b4ea1e93ae0a7c01d401b3cb25315
pText: 68656c6c6f2c20776f726c6421
```

also showing our cipher text has increased in length from the plain text by the length of the tag. As with GCM, an error in evaluating the tag will result in an AEADBadTagException being thrown in a JVM that is compliant with Java 7 or later, or a BadPaddingException exception on an earlier JVM.

Algorithm parameters for CCM can be generated in the same fashion as we saw with GCM. RFC 5084 [66] also defines an ASN.1 definition for CCM parameters as:

```
CCMParameters ::= SEQUENCE {
    aes-nonce         OCTET STRING (SIZE(7..13)),
    aes-ICVlen        AES-CCM-ICVlen DEFAULT 12 }

AES-CCM-ICVlen ::= INTEGER (4 | 6 | 8 | 10 | 12 | 14 | 16)
```

Bouncy Castle provides a `CCMParameters` class in its ASN.1 library. Looking at the definition you can see that it is structurally the same as the ASN.1 definition of GCMParameters, although the restrictions placed on the field values are not quite the same. Again, while the definition is clearly aimed at AES, Bouncy Castle allows for use of the structure with CCM implementations based on other block ciphers as well.

EAX

Another AEAD mode that often comes up is EAX, originally proposed in "The EAX Mode of Operation" [109] by Bellare, Rogaway, and Wagner as a simpler more efficient way of achieving the aims of CCM. Unlike CCM it uses the *Encrypt-then-MAC* approach but it is still based on CTR and uses CMAC as its underlying MAC algorithm - the MAC is actually a composite of the CMAC of the nonce, CMAC of the cipher text, and the CMAC of the associated data.

In this case Bouncy Castle only supports the use of the `AEADParameterSpec` as we can see in the example below:

```java
/**
 * Encrypt the passed in data pText using EAX mode with the passed in
 * parameters.
 *
 * @param key secret key to use.
 * @param nonce the nonce to use with the EAX mode.
 * @param pText the plain text input to the cipher.
 * @return the cipher text.
 */
static byte[] eaxEncrypt(SecretKey key,
                          byte[] nonce,
                          byte[] pText)
    throws Exception
{
    Cipher cipher = Cipher.getInstance("AES/EAX/NoPadding", "BC");

    AEADParameterSpec spec = new AEADParameterSpec(nonce, 128);

    cipher.init(Cipher.ENCRYPT_MODE, key, spec);

```

```
      return cipher.doFinal(pText);
  }
```

In this case, as the tag length refers to the output size of CMAC, the 128 bits value is related to the largest tag size possible for the underlying cipher.

Decrypt follows the same pattern as before:

```
  /**
   * Decrypt the cipher text cText using the passed in key and other
   * parameters using EAX mode.
   *
   * @param key secret key to use.
   * @param nonce the nonce to use with EAX.
   * @param cText the encrypted cipher text.
   * @return the original plain text.
   */
  static byte[] eaxDecrypt(SecretKey key,
                            byte[] nonce,
                            byte[] cText)
      throws Exception
  {
      Cipher cipher = Cipher.getInstance("AES/EAX/NoPadding", "BC");

      AEADParameterSpec spec = new AEADParameterSpec(nonce, 128);

      cipher.init(Cipher.DECRYPT_MODE, key, spec);

      return cipher.doFinal(cText);
  }
```

Putting the two methods together in a simple example produces the following example:

```
/**
 * A simple main for using the EAX methods.
 */
public class EAXExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        byte[] iv = Hex.decode("bbaa99887766554433221100");
        byte[] msg = Strings.toByteArray("hello, world!");

        System.out.println("msg  : " + Hex.toHexString(msg));

        byte[] cText = eaxEncrypt(aesKey, iv, msg);

        System.out.println("cText: " + Hex.toHexString(cText));

        byte[] pText = eaxDecrypt(aesKey, iv, cText);

        System.out.println("pText: " + Hex.toHexString(pText));
    }
}
```

If all has worked well, the sample should produce the following output.

```
msg  : 68656c6c6f2c20776f726c6421
cText: 9153fdf091f0d023ab18645fd2eaa916f0b27f3b1eddf6afac6d950bc4
pText: 68656c6c6f2c20776f726c6421
```

While associated data is not shown in either of these examples, associated data can be added either through the `AEADParameterSpec` or the `Cipher.updateAAD()` methods.

As with GCM and CCM, an error in evaluating the tag will result in an `AEADBadTagException` being thrown in a JVM that is compliant with Java 7 or later, or a `BadPaddingException` exception on an earlier JVM.

Other AEAD Modes

Bouncy Castle also offers OCB mode as defined in [93]. OCB mode was put forward originally as a more efficient alternative to CCM as an OCB mode cipher requires half as many block cipher operations as a CCM mode cipher does for roughly the same effect. From a programming point of view you can treat this one as you would EAX, although the Bouncy Castle implementation is fixed only to use 128 bit block ciphers.

Key Wrapping Algorithms

Key wrapping is primarily about the use of symmetric keys to wrap other keys. We will also see examples later where asymmetric keys are used for encrypting symmetric keys, but operations like that are really done for the purposes of key transport, not for any long term key storage[2].

[2] There are a couple of reasons for this, but the primary one is security, there is little point storing a 256 bit AES key (256 bits of security) by encrypting it with a 2048 bit RSA key (112 bits of security), unless you really do want to downgrade the security of your application to 112 bits.

In this section we will mainly look at the NIST standard key wrapping algorithms defined in NIST SP 800-38F [8] and are defined around the use of symmetric ciphers - in the case of NIST SP 800-38F specifically AES and Triple-DES. The key wrapping algorithms defined in NIST SP 800-38F include some checksum information to help assure that an unwrapped key is the actual key that was wrapped and they are robust enough that they could be used for preserving general data as well, which makes them useful for wrapping asymmetric keys as well - these are generally presented as ASN.1 DER encoded blobs. The robustness property comes from a feature of the algorithms which means that every bit in the finally wrapped encoding is dependent on every bit entered into the encoding.

KW Mode

KW mode is the basic wrap mode without padding. Note that in this case the data must be block aligned on half the block size of the cipher being used. So in the below case, which uses AES, we need to ensure the input key has an encoding which is a multiple of 8 bytes as AES has a 16 byte block size.

The following example wraps a 16 byte Blowfish key using the JCE API.

```java
/**
 * An example of KW style key wrapping - note in this case the input must be
 * aligned on an 8 byte boundary (for AES).
 */
public class KWExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        SecretKeySpec keyToWrap = new SecretKeySpec(
            Hex.decode("00010203040506070706050403020100"), "Blowfish");

        // wrap the key
        Cipher wrapCipher = Cipher.getInstance("AESKW", "BC");

        wrapCipher.init(Cipher.WRAP_MODE, aesKey);

        byte[] cText = wrapCipher.wrap(keyToWrap);

        // unwrap the key
        Cipher unwrapCipher = Cipher.getInstance("AESKW", "BC");

        unwrapCipher.init(Cipher.UNWRAP_MODE, aesKey);

        SecretKey unwrappedKey =
            (SecretKey)unwrapCipher.unwrap(cText, "Blowfish", Cipher.SECRET_KEY);

        System.out.println("key: " + unwrappedKey.getAlgorithm());
        System.out.println("   : " + Arrays.areEqual(
                                         keyToWrap.getEncoded(),
                                         unwrappedKey.getEncoded()));
    }
}
```

Assuming all goes well, you should see the following output:

```
key: Blowfish
   : true
```

Note that in the code we have used `Cipher.WRAP_MODE` and `Cipher.UNWRAP_MODE` to encrypt and decrypt the key. This means that we do not have to call the `getEncoded()` method on the key that we wish to wrap, or go through the steps required to recover the key properly by the provider on doing the unwrap. This can become very important when you are using a Hardware Security Module (HSM), as calling `getEncoded()` may actually return null if the key you wish to wrap is not allowed to leave the HSM unless encrypted (so the key object is really just a token). In such a situation the JCE's wrapping methods will still work, assuming the cipher used for wrapping also exists on the HSM.

It is also worth mentioning here that if you are using a HSM, in some situations, rather than using `Cipher.UNWRAP_MODE`, you may actually want to use `Cipher.DECRYPT_MODE` - occasionally you do want the data that was wrapped back as an encoding you can use in your application. It is no fun discovering that your HSM has automatically locked the unwrapped secret away and returned a token object because you used `Cipher.UNWRAP_MODE`.

KWP Mode

KWP mode defines a padding mechanism which allows us to escape the half-block alignment restriction while at the same time preserving the robustness of the wrapping algorithm.

This example is a little out of sequence as we will not properly look at the `KeyPairGenerator`, `KeyPair`, and `PrivateKey` classes till Chapter 6, but using a private key does give the example something meaningful to use[3]. If you look at the following code you will see that usage is basically the same as for the KW example above; the main difference being the key type we are working with.

[3] So yes, this is one way of generating a 2048 bit RSA key pair.

```
/**
 * An example of KWP style key wrapping with padding.
 */
public class KWPExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        // generate an RSA key pair so we have something
        // interesting to work with!
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("RSA", "BC");

        kpGen.initialize(2048);

        KeyPair kp = kpGen.generateKeyPair();

        // wrap the key
        Cipher wrapCipher = Cipher.getInstance("AESKWP", "BC");

        wrapCipher.init(Cipher.WRAP_MODE, aesKey);

        byte[] cText = wrapCipher.wrap(kp.getPrivate());

        // unwrap the key
        Cipher unwrapCipher = Cipher.getInstance("AESKWP", "BC");

        unwrapCipher.init(Cipher.UNWRAP_MODE, aesKey);

        PrivateKey unwrappedKey =
            (PrivateKey)unwrapCipher.unwrap(cText, "RSA", Cipher.PRIVATE_KEY);

        System.out.println("key: " + unwrappedKey.getAlgorithm());
        System.out.println("   : " + Arrays.areEqual(
            kp.getPrivate().getEncoded(), unwrappedKey.getEncoded()));
    }
}
```

Running the example you should be able to confirm you get back the key you started with by seeing the following output:

```
key: RSA
   : true
```

Other Key Wrapping Algorithms

NIST SP 800-38F also defines TKW, which is built around Triple-DES. TKW is also supported by Bouncy Castle but is rapidly falling out of use. There is no padded version of TKW.

Bouncy Castle also supports the additional IETF key wrapping algorithms as defined in RFC 3370 [54] which were originally from RFC 3217 [53] and RFC 3211 [52]. The first two just apply to RC2 and Triple-DES. The latter one is a more general algorithm based on CBC for use with any block cipher.

The SealedObject Class

In addition to providing an API for invoking key wrapping, the JCE also provides a simple class for allowing the carriage and recovery of encrypted objects. The class that does this is `javax.crypto.SealedObject`.

In creating a `SealedObject` we need to pass the serializable we want sealed and also a `Cipher` instance that we wish to use to do the sealing. The `Cipher` instance needs to be initialised for encryption. Have a look at the following example which is configured to use AES in CCM mode:

```java
/**
 * An example of use of a SealedObject to protect a serializable object. In this
 * case we use a private key, but any serializable will do.
 */
public class SealedObjectExample
{
    public static void main(String[] args)
        throws Exception
    {
        SecretKey aesKey = createConstantKey();

        // create our interesting serializable
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("RSA", "BC");

        kpGen.initialize(2048);

        KeyPair kp = kpGen.generateKeyPair();

        // initialize the "sealing cipher"
        Cipher wrapCipher = Cipher.getInstance("AES/CCM/NoPadding", "BC");

        wrapCipher.init(Cipher.ENCRYPT_MODE, aesKey);

        // create the sealed object from the serializable
        SealedObject sealed = new SealedObject(kp.getPrivate(), wrapCipher);

        // simulate a "wire transfer" of the sealed object.
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        ObjectOutputStream    oOut = new ObjectOutputStream(bOut);

        oOut.writeObject(sealed);

        oOut.close();

        SealedObject transmitted = (SealedObject)new ObjectInputStream(
            new ByteArrayInputStream(bOut.toByteArray())).readObject();

        // unseal transmitted, extracting the private key
        PrivateKey unwrappedKey =
                       (PrivateKey)transmitted.getObject(aesKey, "BC");

        System.out.println("key: " + unwrappedKey.getAlgorithm());
        System.out.println("   : " + Arrays.areEqual(
                kp.getPrivate().getEncoded(), unwrappedKey.getEncoded()));
    }
}
```

Note that to invoke `SealedObject.getObject()` we only need to pass the key to use and the provider name for the services we want. Internally the `SealedObject` also stores the algorithm specification and an encoded version of the `AlgorithmParameters` object returned by the `Cipher` by using `Cipher.getParameters()`. It can then use the provider given to recreate the algorithm parameters and create and initialize a `Cipher` instance appropriate for recovering the object originally stored.

Running this example should produce the same output as the KWP wrap example did before:

```
key: RSA
   : true
```

Confirming that the RSA private key has been extracted successfully from the `SealedObject` instance.

Summary

In this chapter we have looked at a new concept in symmetric key encryption called "authenticated modes". This allows for validating the data stream for correctness when decrypting and also allows for non-encrypted text, such as headers, to be validated if required.

We also discovered how to transport keys securely using key wrapping and the SealedObject class.

Chapter 5: Password Based Key Generation and Key Splitting

Password based key generation and key splitting represent two ways of taking a collection of inputs and ending up with a single secret. They serve different purposes. In the first case a we take something a user knows and use it to derive the secret we want to use. In the second case we start with the secret we want to use, but then conceal it by giving parts of it to different users - the idea being that some number of them have to work together at the same time in order to reveal the original secret. How this is done requires a trip back in time, using a result first proposed by Lagrange in 1771, finally proved for the general case in 1861, and brought to attention for cryptography in 1979.

We will start by looking at the first case.

Password Based Key Generation

In the case of "taking something easy to remember and producing an effective symmetric key" there are a range of algorithms which make use of message digests and random salts in order to produce keys. There is a variety of techniques for doing this, but mostly they are built around message digests. Collectively, these functions are referred to as Password Based Key Derivation Functions, or PBKDFs for short. Encryption done using keys generated with PBKDFs is simply known as Password Based Encryption, or PBE.

Inputs to a PBKDF

In addition to a password, PBKDFs also take a salt and an iteration count, and require a pseudo-random function (PRF) to be defined. The PRF acts as the mixer combining the password, salt, and iteration count together and is used to produce the stream of bytes from the PBKDF which are then generally used to create a secret key for a symmetric cipher. Often the PRF is either a message digest, or a HMAC.

The Password

Of the inputs to a PBKDF only the password is secret. It is important to realise that the password is where the entropy for the final key comes from. As always, the entropy of the key is one of the principal factors in determining the security of any encryption.

Put another way, you can easily generate a 256 bit stream using a PBKDF from a single character password. However, if the password is going to be in the range of 'a' to 'z' inclusive, it will not take a lot of work to realise that only 26 256 bit streams are possible - in reality a security strength of less than 5 bits.

So a really short password is not a great idea.

The other consideration is that the password is always converted into a byte array for feeding into the PRF and mixing with the other components. It is worth knowing how this conversion takes place. While some algorithms will treat characters as 16 bits and others will convert characters to UTF-8 before processing, there are some algorithms that will convert any input they are given into bytes as though the input is 7 bit ASCII. This is important as, if you are trying to take advantage of a non-English character set, or even an extended one, you want to be sure that all your hard work is not then thrown away by the converter the PBKDF uses.

The Salt

The salt is a public value. The salt is always a random string of bytes and allows us to hide the accidental, or intentional, reuse of the same password as using a salt will change the key stream generated from the PBKDF.

The salt can be of any length, but it is best to make it at least as large as the size of the output the PRF produces when it outputs a single block. Normally this is the same size as the length of the output from the associated HMAC, or message digest, being used to support the PRF itself. This means that any “rainbow table” of pre-computed hashes that might be used to attack the PBKDF needs to be of the same order of size as the bit length of the PRF’s output. This will make it extremely difficult, if not impossible, to create enough tables to carry out a meaningful dictionary attack.

The Iteration Count

The iteration count is also a public value. The sole purpose of the iteration count is to increase the computation time required to convert a password to a key, the idea being that if an iteration count of 100 is used, rather than 1, then the calculation will be 100 times more expensive to perform.

In the year 2000, an iteration count of 1024 seemed big enough. These days you want to find a number that is big enough that you are still comfortable with the delay as it increases the computational costs any attacker will face if they are trying to guess passwords.

Other Input Considerations

Originally just the CPU burn that the iteration count determined was enough to provide some level of certainty about the level of protection that your password hashing algorithm had, the assumption being that your server was the most powerful computer on the block. As far as general purpose computers go, this is probably still true. However, with the arrival of programmable GPUs and the ready access to custom VLSI, it is no longer safe to say that as far as a special purpose hashing computer goes your server is the most powerful computer on the block. In fact with traditional PBKDFs custom approaches based on VLSI and GPUs are capable of operating at iteration counts far higher than anything a regular server would cope with.

These days memory usage and other, difficult to scale, factors are now being added into the mix in order to protect the results of password hashing algorithms. From 2013 to 2015 there was a password hashing competition [106] run on the Internet. The winner was an algorithm called Argon2, with special mention for four of the finalists Catena, Lyra2, Makwa, and yescrypt. Argon2 has not yet been added to the Bouncy Castle APIs, but one of the reasons for yescrypt's commendation was the relationship to an algorithm called scrypt, one of the original "extra cost" PBKDFs, which is already in the BC APIs. We will have a look at scrypt a bit further on.

PKCS5 Scheme 2

At the time of writing, the most common technique for creating a PBKDF is the second scheme described in PKCS #5, known as PBKDF2. Part of its popularity is due to the scheme also being recommended by NIST for use in SP 800-132 [15]. PBKDF2 uses a HMAC as a PRF. By default the message digest providing the underlying digest in the HMAC is SHA-1, which gives some indication of the algorithm's age. These days you would use at least SHA-256 or SHA-512.

The algorithm follows the classic pattern for a derivation algorithm, taking a password, salt, and iteration count as input and then repeatedly applying a specific HMAC as determined by the iteration count.

PBKDF2 Using the JCE

The following example shows how to create a JCE `SecretKeyFactory` which is supported by PBKDF2 and uses SHA-256 as the PRF.

```java
/**
 * Calculate a derived key using PBKDF2 based on SHA-256 using
 * the BC JCE provider.
 *
 * @param password the password input.
 * @param salt the salt parameter.
 * @param iterationCount the iteration count parameter.
 * @return the derived key.
 */
public static byte[] jcePKCS5Scheme2(char[] password, byte[] salt,
                                     int iterationCount)
        throws GeneralSecurityException
{
    SecretKeyFactory fact = SecretKeyFactory.getInstance(
                            "PBKDF2WITHHMACSHA256","BC");

    return fact.generateSecret(
            new PBEKeySpec(password, salt, iterationCount, 256))
                                                                .getEncoded();
}
```

PBKDF2 Using the BC Low-level API

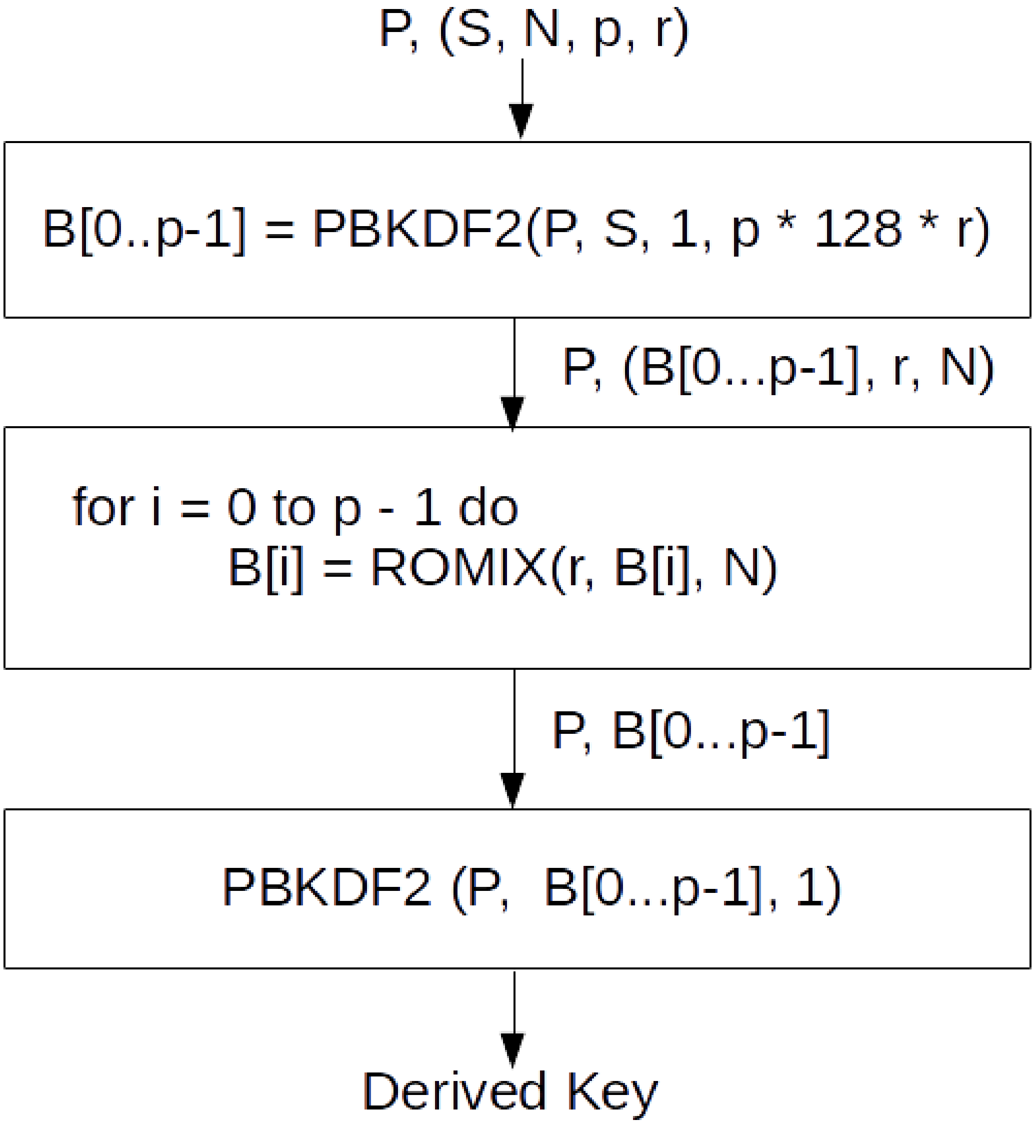
The scrypt algorithm can also be invoked directly in the BC low-level API, using the `PKCS5S2ParametersGenerator` class.

```
/**
 * Calculate a derived key using PBKDF2 based on SHA-256 using
 * the BC low-level API.
 *
 * @param password the password input.
 * @param salt the salt parameter.
 * @param iterationCount the iteration count parameter.
 * @return the derived key.
 */
public static byte[] bcPKCS5Scheme2(char[] password, byte[] salt,
                                    int iterationCount)
{
    PBEParametersGenerator generator = new PKCS5S2ParametersGenerator(
                                                new SHA256Digest());

    generator.init(
            PBEParametersGenerator.PKCS5PasswordToUTF8Bytes(password),
            salt,
            iterationCount);

    return ((KeyParameter)generator
                            .generateDerivedParameters(256)).getKey();
}
```

Note the class takes a message digest as a constructor argument. The message digest is used to determine which HMAC should be used with the PBKDF2 function.

SCRYPT

The scrypt algorithm was originally presented by Colin Percival in 2009 [107] and was followed up by RFC 7914 [97] in 2016. The scrypt algorithm uses PBKDF2 internally, but differs from PBKDF2 in that the algorithm also adds memory hard functions. The additional use of memory resources helps protect the algorithm from solutions based around custom circuitry or the "artful" use of GPUs. You can see how scrypt differs from a regular hash-based PBKDF in the following diagram, not only does it take a salt (S) but it also takes some additional cost parameters N, p, and r, which represent a cpu/memory cost, a parallelization parameter and a blocksize parameter.



A schematic of scrypt - derived key length is not shown but assumed

As you can see rather than just a single round of hashing and salting, the scrypt algorithm is broken up into 3 steps, the first of which is the production of an extended array made up of hashes of the password using PBKDF2. The version of PBKDF2 specified in RFC 7914 for use with scrypt is HMAC SHA-256. Each block in the B array is $(128 * r)$ octets long.

The middle step then applies the ROMIX function to each block in the B array. The ROMIX function is defined in RFC 7914 as `scryptROMix` and is based on the Salsa20 core function.

Fnally, PBKDF2 is applied again to the password using the using the B array as the salt.

You can see it is the use of the B array which makes scrypt “memory hard”. Some care is needed in choosing the parameters as the algorithm will chew through memory, but it makes it a lot harder for an attacker to cheat when trying to guess passwords from the resulting derived keys.

SCRYPT Using the JCE

The scrypt algorithm is available in the JCE via the `SecretKeyFactory` class.

The following example shows how to invoke scrypt in the JCE.

```
/**
 * Calculate a derived key using SCRYPT using the BC JCE provider.
 *
 * @param password the password input.
 * @param salt the salt parameter.
 * @param costParameter the cost parameter.
 * @param blocksize the blocksize parameter.
 * @param parallelizationParam the parallelization parameter.
 * @return the derived key.
 */
public static byte[] jceSCRYPT(char[] password, byte[] salt,
                               int costParameter, int blocksize,
                               int parallelizationParam)
        throws GeneralSecurityException
{
    SecretKeyFactory fact = SecretKeyFactory.getInstance(
                            "SCRYPT","BC");

    return fact.generateSecret(
            new ScryptKeySpec(password, salt,
                    costParameter, blocksize, parallelizationParam,
                    256)).getEncoded();
}
```

Note the `ScryptKeySpec` is a proprietary BC class. There is no equivalent for the `ScryptKeySpec` class in the JCE at the moment.

SCRYPT Using the BC Low-level API

The scrypt algorithm can also be invoked directly in the BC low-level API, using the `SCrypt` class.

```
/**
 * Calculate a derived key using SCRYPT using the BC low-level API.
 *
 * @param password the password input.
 * @param salt the salt parameter.
 * @param costParameter the cost parameter.
 * @param blocksize the blocksize parameter.
 * @param parallelizationParam the parallelization parameter.
 * @return the derived key.
 */
public static byte[] bcSCRYPT(char[] password, byte[] salt,
                              int costParameter, int blocksize,
                              int parallelizationParam)
{
    return SCrypt.generate(
            PBEParametersGenerator.PKCS5PasswordToUTF8Bytes(password),
            salt, costParameter, blocksize, parallelizationParam,
            256 / 8);
}
```

Other PBKDFs

The Bouncy Castle providers support a number of other PBKDFs, including PKCS #12, the original schemes specified in PKCS #5, the original OpenSSL PBKDF, and also those in OpenPGP.

Key Splitting

There are a few algorithms for doing secret sharing, but here we are going to look at Adi Shamir's secret sharing algorithm which was introduced in the 1979 ACM paper "How to Share a Secret" [40].

The algorithm involves splitting a secret into shares and is interesting for two reasons. First, when a secret is split up using the algorithm none of the shares by themselves will allow recovery of the secret. Second, the shares can be distributed on a threshold basis, meaning that it is possible to create shares so that only a percentage of the owners of the shares need to come together to recover the secret. As this is the case, you will often see this algorithm referred to as Shamir's Threshold Scheme.

The Basic Algorithm

The initial set up of Shamir's requires a trusted party to begin with some secret integer S which is to be shared among n users. We also need to have a threshold value, t, which means that any group of t users from our group of n, should be able to recover the secret integer S.

Next our trusted party chooses a prime p which is greater than both S and n and defines a polynomial {$$}f(x){$$} with the coefficients a_0 to a_{t-1} where $0 < a_i <= p - 1$, for $i = 1$ to $t - 1$, and $a_0 = S$.

Finally, using the resulting polynomial the trusted party computes n shares, $S_i = f(i) \mod p$, where i is the index of the share. S_i is then given to user i.

Recovering of the shares then requires t users to come together and use Lagrange Interpolation to compute $a0$ of the mystery polynomial. a0 is the recovered secret S.

Other than expressing the Lagrange Interpolation as an operation over a prime field, the algorithm is fairly straight forward to implement. If you are interested in following up the mathematics, there is a fuller explanation of what is happening in [122] and also on [123] which includes a worked example.

An Implementation of Key Splitting

The following code provides an implementation of key splitting. It's presented here in full, partly to prove it is not outrageously complicated, but also as, despite its usefulness, we have not quite worked out where in Bouncy Castle we would actually put this.

The implementation is divided into 3 parts, a carrier class for the split secret, the actual splitter, and then a Lagrange Weight calculator to help with the interpolation calculation. After the implementation code there is a short example showing how to put the 3 routines together to split and recover a secret. Is this the only way of putting an implementation together? No, but we hope it will explain enough to allow you to adapt it if you wish.

The Split Secret Carrier

The `SplitSecret` class is used to carry the share values needed for distribution to the parties to be entrusted with the ability to recover the secret. As you would expect, these are just big integers over the prime field that has been chosen for polynomial construction.

```java
/**
 * A holder for shares from a split secret for a BigInteger value.
 */
public class SplitSecret
{
    private final BigInteger[] shares;

    /**
     * Base constructor.
     *
     * @param shares the shares the initial secret has been split into.
     */
    public SplitSecret(BigInteger[] shares)
    {
        this.shares = shares.clone();
    }

    /**
     * Return a copy of the shares associated with the split secret.
     *
     * @return an array of the secret's shares.
     */
    public BigInteger[] getShares()
    {
        return shares.clone();
    }
}
```

The Secret Splitter

The 'ShamirSecretSplitter' class provides the business end of the construction of the shares representing the split secret. After configuring it with the number of peers, a threshold, and a prime order representing the field we are working with, we can use the `split()` method to divide up a passed in secret to create the appropriate number of shares.

```
/**
 * A secret splitter based on Shamir's method.
 * <p>
 * Reference: Shamir, Adi (1979), "How to share a secret",
 * Communications of the ACM, 22 (11): 612-613.
 * </p>
 */
public class ShamirSecretSplitter
{
    private final int numberOfPeers;
    private final int k;
    private final BigInteger order;
    private final SecureRandom random;
    private final BigInteger[] alphas;
    private final BigInteger[][] alphasPow;

    /**
     * Creates a ShamirSecretSplitter instance over the specified field
     * to share secrets among the specified number of peers
     *
     * @param numberOfPeers the number of peers among which the secret is
     *                      shared.
     * @param threshold number of peers that must be available for secret
     *                  reconstruction.
     * @param field the prime field representing the group we are operating on.
     * @param random a source of randomness.
     */
    public ShamirSecretSplitter(
            int numberOfPeers, int threshold, BigInteger field,
            SecureRandom random)
    {
        this.numberOfPeers = numberOfPeers;
        this.k = threshold;
        this.order = field;
        this.random = random;

         // Pre-calculate powers for each peer.
        alphas = new BigInteger[numberOfPeers];
        alphasPow = new BigInteger[numberOfPeers][k];

        if (k > 1)
        {
            for (int i = 0; i < numberOfPeers; i++)
            {
                alphas[i] = alphasPow[i][1] = BigInteger.valueOf(i + 1);
                for (int degree = 2; degree < k; degree++)
                {
                    alphasPow[i][degree] = alphasPow[i][degree - 1]
                                                    .multiply(alphas[i]);
                }
            }
        }
        else
        {
            for (int i = 0; i < numberOfPeers; i++)
            {
                alphas[i] = BigInteger.valueOf(i + 1);
            }
        }
    }

    /**
     * Given the secret, generate random coefficients (except for a<sub>0</sub>
     * which is the secret) and compute the function for each privacy peer
     * (who is assigned a dedicated alpha). Coefficients are picked from (0,
     * fieldSize).
     *
     * @param secret the secret to be shared
     * @return the shares of the secret for each privacy peer
     */
    public SplitSecret split(BigInteger secret)
    {
        BigInteger[] shares = new BigInteger[numberOfPeers];
        BigInteger[] coefficients = new BigInteger[k];

        // D0: for each share
        for (int peerIndex = 0; peerIndex < numberOfPeers; peerIndex++)
        {
            shares[peerIndex] = secret;
        }

        coefficients[0] = secret;

        // D1 to DT: for each share
        for (int degree = 1; degree < k; degree++)
        {
            BigInteger coefficient = generateCoeff(order, random);

            coefficients[degree] = coefficient;

            for (int peerIndex = 0; peerIndex < numberOfPeers; peerIndex++)
            {
                shares[peerIndex] = shares[peerIndex].add(
                                      coefficient
                                         .multiply(alphasPow[peerIndex][degree])
                                         .mod(order)
                                    ).mod(order);
            }
        }

        return new SplitSecret(shares);
    }

    // Shamir's original paper actually gives the set [0, fieldSize) as the range
    // in which coefficients can be chosen, this isn't true for the highest
    // order term as it would have the effect of reducing the order of the
    // polynomial. We guard against this by using the set (0, fieldSize) and
    // so removing the chance of 0.
    private static BigInteger generateCoeff(BigInteger n, SecureRandom random)
    {
        int nBitLength = n.bitLength();
        BigInteger k = new BigInteger(nBitLength, random);

        while (k.equals(BigInteger.ZERO) || k.compareTo(n) >= 0)
        {
            k = new BigInteger(nBitLength, random);
        }

        return k;
    }
}
```

A Lagrange Weight Calculator

The `LagrangeWeightCalculator` is a utility class used to help reassemble the available shares to recover the split secret. Note in the `computeWeights()` method, elements of the activePeers array are checked for null as some peers are likely to be missing, but for the correct weight to be calculated the index of the peer in the original array of shares must be known.

```
/**
 * A basic calculator for Lagrangian weights for a given number of peers in a
 * given field.
 */
public class LagrangeWeightCalculator
{
    private final int numberOfPeers;
    private final BigInteger field;
    private final BigInteger[] alphas;

    /**
     * Construct a calculator over the specified field to calculate weights for
     * use to process secrets shared among the specified number of peers
     *
     * @param numberOfPeers the number of peers among which the secret is shared
     * @param field the group's field.
     */
    public LagrangeWeightCalculator(int numberOfPeers, BigInteger field)
    {
        this.numberOfPeers = numberOfPeers;
        this.field = field;

        this.alphas = new BigInteger[numberOfPeers];

        for (int i = 0; i < numberOfPeers; i++)
        {
            alphas[i] = BigInteger.valueOf(i + 1);
        }
    }

    /**
     * Computes the Lagrange weights used for interpolation to reconstruct the
     * shared secret.
     *
     * @param activePeers an ordered array of peers available, entries are null
     *                    if no peer present.
     * @return the Lagrange weights
     */
    public BigInteger[] computeWeights(Object[] activePeers)
    {
        BigInteger[] weights = new BigInteger[numberOfPeers];

        for (int i = 0; i < numberOfPeers; i++)
        {
            if (activePeers[i] != null)
            {
                BigInteger numerator = BigInteger.ONE;
                BigInteger denominator = BigInteger.ONE;

                for (int peerIndex = 0; peerIndex < numberOfPeers; peerIndex++)
                {
                    if (peerIndex != i && activePeers[peerIndex] != null)
                    {
                        numerator = numerator.multiply(alphas[peerIndex])
                                             .mod(field);
                        denominator = denominator.multiply(alphas[peerIndex]
                                                .subtract(alphas[i]).mod(field))
                                                .mod(field);
                    }
                }

                weights[i] = numerator.multiply(denominator.modInverse(field))
```

```
                                          .mod(field);
            }
        }

        return weights;
    }
}
```

Putting it Together

Finally we put it together.

The following code shows how to recombine the secret using the approach expected by the Lagrange weight calculator and gives a simple example of usage for a generated secret. Running this should produce the words `matched: true` in the output.

```
/**
 * Simple example of use of the ShamirSecretSplitter and
 * LagrangeWeightCalculator for the splitting and recovery of a BigInteger
 * secret.
 */
public class ShamirExample
{
    // recover the shared secret.
    // Note: we use the position of the share in the activeShares array to
    // determine its index value for the f(i) function.
    static BigInteger recoverSecret(BigInteger field, BigInteger[] activeShares)
    {
        LagrangeWeightCalculator lagrangeWeightCalculator =
                       new LagrangeWeightCalculator(activeShares.length, field);

        BigInteger[] weights =
                       lagrangeWeightCalculator.computeWeights(activeShares);

        // weighting
        int index = 0;
        while (weights[index] == null)
        {
            index++;
        }
        BigInteger weightedValue = activeShares[index]
                                          .multiply(weights[index]).mod(field);
        for (int i = index + 1; i < weights.length; i++)
        {
            if (weights[i] != null)
            {
                weightedValue = weightedValue.add(
                    activeShares[i].multiply(weights[i]).mod(field)).mod(field);
            }
        }

        return weightedValue;
    }

    // Create a new shares array with just the ones listed in index present and
    // the rest of the entries null.
    private static BigInteger[] copyShares(int[] indexes, BigInteger[] shares)
    {
        // note: the activeShares array  needs to be the same size as the shares
        // array. The order of the shares is important.
        BigInteger[] activeShares = new BigInteger[shares.length];

        for (int i = 0; i != indexes.length; i++)
        {
            activeShares[indexes[i]] = shares[indexes[i]];
        }

        return activeShares;
    }

    /**
     * Create a shared secret and generate a polynomial for the prime field p
     * to split it over, do the split and then show a recovery of the secret.
     */
    public static void main(String[] args)
    {
        SecureRandom random = new SecureRandom();
        BigInteger p = new BigInteger(384, 256, random);
        int numberOfPeers = 10;
        int threshold = 4;

        byte[] secretKey = new byte[32];

        random.nextBytes(secretKey);

        BigInteger secretValue = new BigInteger(1, secretKey);

        ShamirSecretSplitter splitter =
                    new ShamirSecretSplitter(
                            numberOfPeers, threshold, p, random);

        SplitSecret secret = splitter.split(secretValue);

        BigInteger[] s = secret.getShares();

        BigInteger recoveredSecret = recoverSecret(p,
                                       copyShares(new int[] { 1, 2, 3, 7 }, s));

        System.err.println(Hex.toHexString(
            BigIntegers.asUnsignedByteArray(secretValue)));
        System.err.println(Hex.toHexString(
            BigIntegers.asUnsignedByteArray(recoveredSecret))
            + ", matched: " + secretValue.equals(recoveredSecret));
    }
}
```

It is worth noting that the weight calculator is configured using the length of the active shares array, but that the value is only used to determine the calculations across the share array. If the active shares array was shorter than the number of actual shares originally distributed you would still get the same weights calculated, so the weight calculator really only needs to know the maximum index from the set of active peers being processed, rather than the original maximum index.

Summary

In this chapter we have looked at two of the fundamental approaches for taking multiple sources of input, and creating a secret from the input. Password based key generation allows for user created data to derive the secret. Key Splitting starts with the secret that is then distributed in pieces to multiple parties.

We examine in detail how to use both PKCS5 snd SCRYPT for creating secrets from user data entry using both the JCE, and the BouncyCastle low level API.

Finally, we provide a detailed look at Key Splitting algorithms using Shamir’s Threshold Scheme.

Chapter 6: Signatures

This chapter introduces the use of public key cryptography and how it is applied to the creation of and verification of digital signatures. We will also look at key pair generation and key importing.

Key Pair Generation and Import

Any discussion on public or asymmetric key cryptography in Java needs to start with a discussion of the basic classes for generating and building public keys.

The JCA provides two base interfaces for defining asymmetric keys:

- `java.security.PublicKey`
- `java.security.PrivateKey`

Asymmetric key pairs are generated using the `java.security.KeyPairGenerator` class. We will see examples of key pair generation as we look at each algorithm. Another class which is useful to know about is `java.security.KeyFactory`. This class is used for importing keys from an external source and converting them so they can be used with a specific provider.

The following short method shows how to generate a public key from a key specification for a given algorithm name and key specification:

```java
/**
 * Return a public key for algorithm built from the details in keySpec.
 *
 * @param algorithm the algorithm the key specification is for.
 * @param keySpec a key specification holding details of the public key.
 * @return a PublicKey for algorithm
 */
public static PublicKey createPublicKey(String algorithm, KeySpec keySpec)
    throws GeneralSecurityException
{
    KeyFactory keyFact = KeyFactory.getInstance(algorithm, "BC");

    return keyFact.generatePublic(keySpec);
}
```

You can also perform the equivalent task to create a private key like this:

```java
/**
 * Return a private key for algorithm built from the details in keySpec.
 *
 * @param algorithm the algorithm the key specification is for.
 * @param keySpec a key specification holding details of the private key.
 * @return a PrivateKey for algorithm
 */
public static PrivateKey createPrivateKey(String algorithm, KeySpec keySpec)
    throws GeneralSecurityException
{
    KeyFactory keyFact = KeyFactory.getInstance(algorithm, "BC");

    return keyFact.generatePrivate(keySpec);
}
```

The `KeyFactory` methods are able to process ASN.1 encodings, such as those you might find in a certificate encoding or a PKCS #12 file. You will see the specs that carry these encodings used further on. At the moment just be aware that public key encodings can be produced by `PublicKey.getEncoded()` and are passed into the `KeyFactory.generatePublic()` method using an `X509EncodedKeySpec` object and the private key encodings can be produced by the `PrivateKey.getEncoded()` and are passed into `KeyFactory.generatePrivate()` by using a `PKCS8EncodedKeySpec` object.

In addition to the specs carrying ASN.1 encodings we will also see that some algorithms have specific spec types for carrying the values needed to construct their public and private keys.

Digital Signatures

Digital signature schemes are generally hybrid schemes which use a private key for signature creation where the signature contains as part of its calculation a hash of the data the signature applies to, or holds some encoded representation of the hash of the data. The idea behind a digital signature is to bind a piece of public information (in this case the document/message being signed) to a piece of private information (the private key of the signer) so that others can confirm that the holder of the private information was responsible for the binding (by using the public key of the signer).

Digital signatures come in two flavors: deterministic and non-deterministic. Several algorithms including DSA, rely on the use of random values to protect the private key of the signer with the result that no two signatures of the same document by the same party are the same. Others that include those based on RSA produce deterministic signatures, which always look the same where the signing key and the document being signed are also the same.

Signature Security Strengths

One of the considerations with a choice of key size and hash algorithm for a digital signature is the security strength of the component parts. Before we look at the algorithms supported in Bouncy Castle and the JCA, it is worth pausing briefly to look at the relative security strengths of the algorithms involved. In our case we will just look at the published NIST algorithms as they appear in SP 800-57 [12] as the data on those is readily available. The security of a signature depends on the public key algorithm used, the key size of the public/private key pair, and the message digest used.

Table 3 in SP 800-57 gives the security strengths for different digest algorithms as:

Digest Algorithm Security Strengths

Digest Algorithm	Security Strength (bits)
SHA-1	<= 80
SHA-224, SHA-512/224, SHA3-224	112
SHA-256, SHA-512/256, SHA3-256	128
SHA-384, SHA3-384	192
SHA-512, SHA3-512	256

For digest algorithms not described in the table, you could consider this as a place to start, but caution should be used trying to extrapolate from this table to other digest algorithms. The first thing to be aware of is that the strength of a digest is related both to its output size and the internal buffer used for its compression function. The second thing to be aware of is that research and analysis of the algorithm might also show other factors apply and need to be taken into account in the usage of the algorithm. As an example, in the case of its use in a signature, while SHA-1 is clearly below the 112 bit security level currently required in FIPS, the security of SHA-1 when combined with a public key algorithm is still subject to debate, so verifying a SHA-1 based signature is potentially risky, but probably okay. Producing a new SHA-1 signature, on the other hand, is not allowed under FIPS as there are now better algorithms which are readily available.

The public key algorithms applicable to NIST standards are described in Table 2 of SP 800-57 as:

Public Key Algorithm Security Strengths

Public Key Algorithm	Key Size	Security Strength (bits)
DSA	1024	<= 80
DSA	2048	112
DSA	3072	128
DSA	7680[4]	192
DSA	15360[4]	256
RSA	1024	<= 80
RSA	2048	112
RSA	3072	128

[4] these key sizes are not currently used in the NIST standards.

Public Key Algorithm Security Strengths

Public Key Algorithm	Key Size	Security Strength (bits)
RSA	7680[5]	192
RSA	15360[5]	256
ECDSA	160-223	<= 80
ECDSA	224-255	112
ECDSA	256-383	128
ECDSA	384-511	192
ECDSA	512+	256

With public key cryptography, the general rule is to try and make the public key algorithm the weakest link, so try to always use a digest algorithm which is at least as strong as the public key used. Likewise if there are functions, such as mask generation functions, involved in signature generation you also want the digest used in the mask generation function to be at least the same security as the digest in the signature. The security of something being "subject to debate" does introduce the possibility of having to apologize and possibly re-engineer in a hurry at a later date - this is really better avoided.

The Signature Class

Services related to signature creation and verification in the JCA are made available through the `Signature` class which can be found in the `java.security` package. Like other JCA classes, objects of the `Signature` type are not created directly, but are created via `getInstance()` methods. As signature algorithms usually involve a message digest, a standard form for signature algorithm names has evolved. The standard format has the form of:

<message digest>with<public key algorithm>

For example, SHA256withDSA is the algorithm name for DSA using the SHA-256 message digest for processing the message, and SHA512withRSA is the algorithm name for the original RSA signature algorithm using SHA-512 as the hash function for the message.

How a `Signature` object is initialized depends partly on whether the object will be used for creating signatures or for the verification of signatures. In the case of signature creation, which is a private key operation, the `initSign()` method is used. In the case of signature verification, which is a public key operation, the `initVerify()` method is called with either a public key or a `Certificate` object that contains the public key. Occasionally the signature algorithm will also require parameters. If that applies, then the `setParameter()` method taking an `AlgorithmParameterSpec` needs to be called as well.

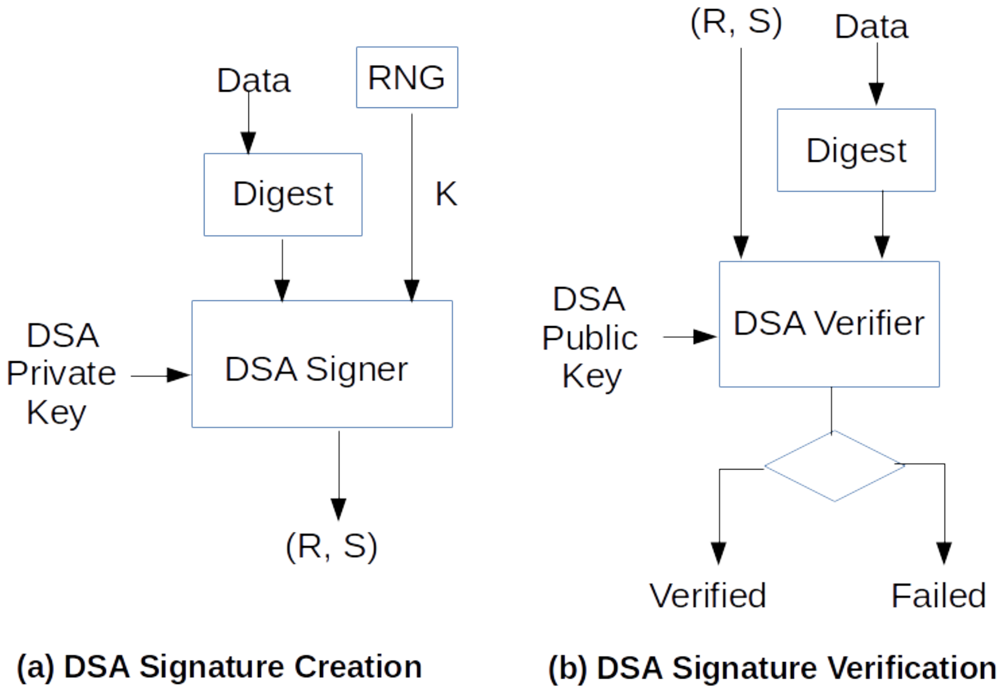
Once the `Signature` object is initialized, the data that is related to the signature being generated or verified is then passed to the appropriate `update()` methods. Once all the data has been passed to the `Signature` object, if the object was initialized for signature generation, the `sign()` method is used to generate a signature. If the object was initialized for signature verification instead, the `verify()` method is called passing in the byte array representing the signature that was originally generated for the data with the originator's private key.

[5] these key sizes are not currently used in the NIST standards.

In essence, three steps, initialize, update, and then sign or verify depending on which `init()` method was called. Keeping this description in mind, we will now take a look at some real signature schemes and see how the `Signature` class is used in practice.

The Digital Signature Algorithm

The Digital Signature Algorithm (DSA) dates back to August 1991 and was the first signature algorithm recognized by NIST with the publication of FIPS PUB 186 [1], which described a standard built around the DSA algorithm called the Digital Signature Standard, or DSS. The DSS is now described in FIPS PUB 186-4 [2] and covers a broader range of signature algorithms but the original DSA is still with us, albeit with much larger keys than originally specified.



(a) DSA Signature Creation

(b) DSA Signature Verification

A schematic of DSA signature verification and generation

DSA style signatures are always made up of two integers called R and S, which are usually generated from a hash of the data to be signed, the private key, and a secret value, usually random. The use of the secret random value in the generation of DSA signatures means the DSA algorithms are usually non-deterministic in their behaviour.

DSA Based on Big Integers

A DSA key pair is based around a set of parameters (P, Q, G) where the parameters have the following characteristics.

- P: a prime with a bit length from the set (1024, 2048, 3072).
- Q: a prime divisor of $(P - 1)$ with a bit length from the set (160, 224, 256). Which bit length is chosen is also dependent on the bit length of P. 160 is only used where P has a bit length of 1024, where P has length of 2048 either 224 or 256 can be used, and for P of length 3072 only 256 can be used.
- G: a generator of a subgroup of order Q in the multiplicative group of Galois Field $GF(P)$, such that $1 < G < P$.

In Java these parameters are carried in a `java.security.spec.DSAParameterSpec` object and are required to generate keys. The parameters can be generated internally in the key pair generator, or you can pass in pre-defined ones, or ones you have generated using a `java.security.AlgorithmParameterGenerator`. A simple example of generating a set of parameters suitable for use with 2048 bit keys follows.

```java
/**
 * Return a generated set of DSA parameters suitable for creating 2048
 * bit keys.
 *
 * @return a DSAParameterSpec holding the generated parameters.
 */
public static DSAParameterSpec generateDSAParams()
    throws GeneralSecurityException
{
    AlgorithmParameterGenerator paramGen =
        AlgorithmParameterGenerator.getInstance("DSA", "BC");

    paramGen.init(2048);

    AlgorithmParameters params = paramGen.generateParameters();

    return params.getParameterSpec(DSAParameterSpec.class);
}
```

The algorithm used to generate DSA parameters is described in FIPS PUB 186-4. It is effective, but if you try using the method above you will realise it is not fast.

Key Pair Generation

Key pair generation involves the creation of a public value (called Y) and a private value (called X). The private value X is randomly generated where $0 < X < Q$ and Y is calculated as $Y = G^X \mod P$. The private X value is then used with the key parameters for creating signatures and the Y value is used with the parameters to verify them.

In Java, the most basic way of generating a key pair is to simply specify the size.

```
/**
 * Generate a 2048 bit DSA key pair using provider based parameters.
 *
 * @return a DSA KeyPair
 */
public static KeyPair generateDSAKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("DSA", "BC");

    keyPair.initialize(2048);

    return keyPair.generateKeyPair();
}
```

In this case the service generating the key pair will either choose or generate a parameter set which will support the generation of a 2048 bit key. If it is generating a parameter set there will probably be a noticeable time delay. Creating DSA parameters is quite expensive. You can also specify your own parameters for generating a key if you wish using the `DSAParameterSpec` by making use of the `KeyPairGenerator.initialize()` method that takes an `AlgorithmParameterSpec`. We can adapt our `generateDSAKeyPair()` method from the previous example to do this by modifying the code to the following:

```
/**
 * Generate a DSA key pair using our own specified parameters.
 *
 * @param dsaSpec the DSA parameters to use for key generation.
 * @return a DSA KeyPair
 */
public static KeyPair generateDSAKeyPair(DSAParameterSpec dsaSpec)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("DSA", "BC");

```

```
        keyPair.initialize(dsaSpec);

        return keyPair.generateKeyPair();
    }
```

and then pass the `DSAParameterSpec` returned from the `generateDSAParams()` method above as the parameters for generating keys.

In both cases, the `KeyPair` returned by the example method will consist of implementations of `DSAPublicKey` and `DSAPrivateKey` from the `java.security.interfaces` package.

Signature Generation

DSA derives its security from the discrete logarithm problem, with one slight twist. A DSA signature is made up of an R and S value, values which are calculated using the following steps:

1. Generate a random K such that $0 < K < Q$
2. Calculate $R = (G^K \mod P) \mod Q$
3. Calculate $S = (K^{-1}(H(m) + XR) \mod Q$

Where $H(m)$ is a hash of the message m that we are generating the signature for.

Note again the K value is meant to be random. This means that two signatures of the same data can be expected to be different. Furthermore if K is not random, or is somehow known, it becomes a lot easier to recover the value X - the private value that forms the secret part of the private key. If someone can do that, they can start generating forged signatures which cannot be detected. When you are generating DSA signatures using a random K, you want to make sure you are using a good random number generator.

DSA's security relies on the randomness of K. If you cannot find a good RNG to generate K with you should be using a different algorithm, or using the deterministic variant of DSA.

Keeping all this in mind, the Java code for creating a DSA signature is quite simple. The following example creates a DSA signature using the SHA-256 message digest as the message hash algorithm.

```
/**
 * Generate an encoded DSA signature using the passed in private key and
 * input data.
 *
 * @param dsaPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateDSASignature(PrivateKey dsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withDSA", "BC");

    signature.initSign(dsaPrivate);

    signature.update(input);

    return signature.sign();
}
```

You will note that `signature.sign()` returns a `byte[]` and you are probably wondering what happened to R and S. In this case R and S are returned as two ASN.1 INTEGER objects in an ASN.1 SEQUENCE. Keep in mind that ASN.1 INTEGERs are signed, so DSA signatures can vary slightly in length, due the presence, or lack, of a sign byte on one or both of the component integers.

Signature Verification

Verification with DSA is slightly more complex and, assuming validation of the inputs, requires the following steps:

1. Calculate $W = s^{-1} \mod Q$
2. Calculate $U_1 = (W * H(m)) \mod Q$
3. Calculate $U_2 = (RW) \mod Q$
4. Calculate $V = ((G^{U_1} Y^{U_2}) \mod P) \mod Q$
5. verified = $(V == R)$

Once again, in Java, this is all kept “under the hood”, and the process for verifying a DSA signature that was generated by our signature generation example looks as follows:

```java
/**
 * Return true if the passed in signature verifies against
 * the passed in DSA public key and input.
 *
 * @param dsaPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyDSASignature(
    PublicKey dsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withDSA", "BC");

    signature.initVerify(dsaPublic);

    signature.update(input);

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

While this should always be straightforward, one common mistake in third party libraries for generating DSA signatures is to forget that the component integers in the signature are signed. If you have a signature which really should verify and it does not, the most likely cause of the error is that the DSA signature was originally created as though the integers were unsigned and one of them has the top bit set, making it a negative number. A review of the numerical steps above for verification indicates that if either R or S are negative things are not going to turn out well. If you are faced with a situation like this and unable to get the third party library fixed, you can use the Bouncy Castle ASN.1 library to pre-process the signature, re-encoding it in an ASN.1 SEQUENCE using the correct ASN.1 INTEGER values and then pass the resulting byte array to the `Signature` class for verification.

The following utility method shows one way of doing this re-encoding using the Bouncy Castle ASN.1 library.

```
/**
 * Fix a faulty DSA signature that has been encoded using
 * unsigned integers.
 *
 * @param encSignature the encoded signature.
 * @return the corrected signature with signed integer components.
 */
public static byte[] patchDSASignature(byte[] encSignature)
    throws IOException
{
    ASN1Sequence sigSeq = ASN1Sequence.getInstance(encSignature);

    ASN1EncodableVector sigV = new ASN1EncodableVector();

    // fix R
    sigV.add(new ASN1Integer(
        ASN1Integer.getInstance(sigSeq.getObjectAt(0)).getPositiveValue()));

    // fix S
    sigV.add(new ASN1Integer(
        ASN1Integer.getInstance(sigSeq.getObjectAt(1)).getPositiveValue()));

    return new DERSequence(sigV).getEncoded();
}
```

Of course it is always better if you can get the organisation producing the faulty encoding to mend their ways. There are situations where that is not always possible, in which case recoding will be necessary.

DSA With Elliptic Curve

DSA over Elliptic Curve, or ECDSA as it known for short, was originally described in X.962 [35] and is now also covered in FIPS PUB 186-4 [2].

ECDSA is defined over two finite fields: the field GF_p and the field GF_{2^m}. The difference between the two fields is the manner in whch operations are done. In the case of GF_p operations are done modulo an odd prime and in the case of GF_{2^m} operations are done modulo an irreducible polynomial. For our purposes here we decide which formulation we use by selecting a specific curve to generate our keys with. In the case of FIPS PUB 186-4 all the GF_p curves start with the characters “P-“ and the field GF_{2^m} start with either “B-“ or “K-“. The “K-“ in this case indicates a Koblitz curve, which is a special case of GF_{2^m}.

In Java the domain parameters for different curves can be defined either by name, using the `java.security.spec.ECGenParameterSpec` class, or in terms of their component values using the `java.security.spec.ECParameterSpec` parameter spec class.

Key Pair Generation

Having selected the domain parameters for a specific curve, generating a key pair involves generating a secret value d and then calculating a point Q which serves as the public key. Q is calculated on the curve off the base point G by calculating $Q = dG$. In addition to the base point G the other value we will make use of in the curve is its order N.

Note that in Java, the naming convention used is different to the one used in X9.62 and FIPS PUB 186-4 - s is used to represent d and W is used to represent Q. The values still represent the same things, but it is important to be aware of this as the standards use d and Q.

Generating a key pair is done via the `KeyPairGenerator` class. For named curves an `ECGenParameterSpec` is passed to the `KeyPairGenerator.initialize()` method. The next example shows a simple method to generate an elliptic curve key pair from a curve with the passed in name curveName.

```java
/**
 * Generate a EC key pair on the passed in named curve.
 *
 * @param curveName the name of the curve to generate the key pair on.
 * @return a EC KeyPair
 */
public static KeyPair generateECKeyPair(String curveName)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("EC", "BC");

    keyPair.initialize(new ECGenParameterSpec(curveName));

    return keyPair.generateKeyPair();
}
```

Using the above method to create an EC key pair on the curve P-256, a 256 bit curve over GF_p would then require the following code:

```
/**
 * Generate a EC key pair on the P-256 curve.
 *
 * @return a EC KeyPair
 */
public static KeyPair generateECKeyPair()
    throws GeneralSecurityException
{
    return generateECKeyPair("P-256");
}
```

An alternative to using an `ECGenParameterSpec` is to pass an `ECParameterSpec` to `KeyPairGenerator.initialize()` instead. To replicate the initialization for P-256 you would create an `ECParameterSpec` which carried all the `P-256` domain parameters in it instead. It is generally easier, and clearer, to do things by name. Creating the key by name also means that if the keys are ever encoded via the `getEncoded()` method, the encoder will be able to use the specific OID representing P-256 in the key parameters, rather than encoding the EC domain parameters in full as components.

The `KeyPair` carrying the EC keys will be made up of implementations of `ECPublicKey` and `ECPrivateKey` from the `java.security.intefaces` package.

Signature Generation

Signature generation in ECDSA follows a very similar pattern to regular DSA, it is just in this case we are dealing with points on an Elliptic Curve rather than simple primes. The basic steps for calculating a signature are:

1. Generate a random K such that $0 < K < N$
2. Calculate $(x, y) = KG$
3. Calculate $R = x \mod N$ if $R == 0$ go back to step 1.
4. Calculate $S = (K^{-1}(H(m) + dR) \mod N$

Where $H(m)$ is again the hash of the message we are signing and N is the order of the EC curve. The following example shows the use of SHA-256, as we saw before, and you can see the JCA steps look the same as those for regular DSA.

```
/**
 * Generate an encoded ECDSA signature using the passed in EC private key
 * and input data.
 *
 * @param ecPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateECDSASignature(
    PrivateKey ecPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withECDSA", "BC");

    signature.initSign(ecPrivate);

    signature.update(input);

    return signature.sign();
}
```

Likewise the same approach is also used for the encoding of the signature. As the steps return integers R and S the `signature.sign()` call is also returning an encoding of an ASN.1 SEQUENCE containing two ASN.1 INTEGER objects.

Signature Verification

Verification with ECDSA is slightly more complex than with DSA as there is a chance an intermediate step in the process may produce an invalid EC point. For ECDSA, assuming validation of the inputs, the following steps are required to verify a signature:

1. Calculate $W = s^{-1} \mod N$
2. Calculate $U_1 = (W * H(m)) \mod N$
3. Calculate $U_2 = (RW) \mod N$
4. Calculate $(x, y) = U_1G + U_2Q$. If $((x, y)$ is at $\infty)$ return invalid.
5. verified = $(x == R)$

Here is an example of signature verification for the generation method given above.

```java
/**
 * Return true if the passed in ECDSA signature verifies against
 * the passed in EC public key and input.
 *
 * @param ecPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyECDSASignature(
    PublicKey ecPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withECDSA", "BC");

    signature.initVerify(ecPublic);

    signature.update(input);

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

Note that the warning for DSA about incorrect encoding of signatures by third party libraries applies just as much for ECDSA as well.

Deterministic DSA (DDSA/ECDDSA)

As we saw earlier, DSA relies on the use of a random number K which must also be kept secret. One of the downsides of this is that re-signing the same document with the same key will produce a different signature as the signatures is non-deterministic. This non-determinism is due to the use of the random K value. A method for creating deterministic DSA signatures is described in RFC 6979 [89]. The “trick” used is to calculate K rather than use a random number, but to calculate K in a manner which keeps it as secret as the private key. As K is only required to generate a DSA signature, this also means that signatures generated using the deterministic formulation of DSA can be verified by any other implementation of DSA. In Bouncy Castle the two formulations of deterministic DSA have the names DDSA, and ECDDSA.

The following method generates an ECDDSA signature, rather than an ECDSA one.

```
    /**
     * Generate an encoded Deterministic ECDSA (ECDDSA) signature using the
     * passed in EC private key and input data.
     *
     * @param ecPrivate the private key for generating the signature with.
     * @param input the input to be signed.
     * @return the encoded signature.
     */
    public static byte[] generateECDDSASignature(
        PrivateKey ecPrivate, byte[] input)
        throws GeneralSecurityException
    {
        Signature signature = Signature.getInstance("SHA256withECDDSA", "BC");

        signature.initSign(ecPrivate);

        signature.update(input);

        return signature.sign();
    }
```

If you drop the "EC" from the name, you will get the equivalent functionality in regular DSA as well.

The following example makes use of the deterministic generate method and then uses the ECDSA verify method we defined earlier `verifyECDSASignature()` to verify the signature.

```
/**
 * An example of using Deterministic ECDSA (ECDDSA) to sign data and then
 * verifying the resulting signature.
 */
public class EcDDsaExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        byte[] msg = Strings.toByteArray("hello, world!");

        PKCS8EncodedKeySpec ecPrivSpec = new PKCS8EncodedKeySpec(
          Base64.decode(
            "MIGTAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBHkwdwIBAQQgguOIC1cI1lPdLPHglG"
          + "qRYLLYbQJ03/bSyHdTGTqGcwegCgYIKoZIzj0DAQehRANCAATQN4K61MQt/SrSqkJ+"
          + "SAMm6g7BjATXKG1f4QqXf8V4syevh6kck426Jb7A5apWZjktuEKfzFvzMj0IaDa1zM"
          + "18"));

        X509EncodedKeySpec ecPubSpec = new X509EncodedKeySpec(
            Base64.decode(
             "MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE0DeCutTELf0q0qpCfkgDJuoOwYwE1"
          + "yhtX+EKl3/FeLMnr4epHJONuiW+wOWqVmY5LbhCn8xb8zI9CGg2tczNfA=="));

        ECPrivateKey ecPriv = (ECPrivateKey)createPrivateKey("EC", ecPrivSpec);
        byte[] ecdsaSignature = generateECDDSASignature(ecPriv, msg);

        // Note that the verification step is the same as for regular ECDSA.
        ECPublicKey ecPub = (ECPublicKey)createPublicKey("EC", ecPubSpec);
        System.out.println("ECDSA verified: " + verifyECDSASignature(
            ecPub, msg, ecdsaSignature));
        System.out.println("sig: " + Hex.toHexString(ecdsaSignature));
    }
}
```

As you can see in the code the example makes use of constant keys so it can be seen that the signature really does come out constant as well. The keys were generated using a regular EC key pair generator and then the Base64 data making up the key specs was generated using the output of the `getEncoded()` methods on the resulting `PublicKey` and `PrivateKey` objects.

When you run the example you should get (less the line wrap) the following output:

```
ECDSA verified: true
sig: 304502202e5fe9a1992cf5bf4f575ab4490dd9f968957e8b3bd47386cacce208a9fe26e10221009\
697fc5b9c6592db755e36d84b8b549d7397b177e3af5e1f2af870a11904b370
```

Try changing the different inputs to the example - you will see that the signature value only depends on the keys used and the input to be signed. Unlike regular ECDSA there is no randomness in the signature whatsoever.

DSA with Edwards Curves (EdDSA)

Edwards Curves have been around for a while now and a formal standard was issued by the IETF with the release of RFC 8032 [101]. The use of Edwards Curves in DSA represents a departure from regular ECDSA at two levels. The first is that only one digest algorithm can be used with each curve and there are also some additional inputs, in addition to the message to be signed, that are required to be hashed in the digest. The second is that the security of the signature does not rely on the use of a unique random number for each signature generated - the signatures are deterministic. The formulation of the curves should also make calculations involving them more resilient to side channel attacks as well.

There are two curves, of differing signature strength, specified for EdDSA. The first curve, Ed25519, is intended to provide 128 bits of security, and the second curve Ed448 offers around 224 bits of security. Ed448 is provided for applications that are willing to trade off some performance in order to better guard against analytical attacks on the curve structures. SHA-512 is used as the digest algorithm with Ed25519 and the SHA-3 algorithm, SHAKE-256, is used as the digest algorithm for Ed448.

Key Pair Generation

Key pairs for EdDSA can be generated by creating an instance of `KeyPairGenerator` for the specific curve of interest, using either the string "Ed25519" or the string "Ed448".

The following example shows key pair generation for Ed448:

```java
/**
 * Generate a EdDSA key pair for Ed448.
 *
 * @return a EdDSA KeyPair
 */
public static KeyPair generateEd448KeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("Ed448", "BC");

    return keyPair.generateKeyPair();
}
```

The `KeyPair` carrying the EdDSA keys will be made up of implementations of `EdDSAKey` interface from the `org.bouncycastle.jcajce.interfaces` package.

Signature Generation

In addition to the message to be signed, the signature generation process also takes a context C. The context C is meant to serve the purpose of locking the signatures being generated down to a particular use and it is needed to be known by both the signer and the verifier. It can be empty, but where it is not it must be no greater than 255 bytes. As there is a previous definition of Ed25519, RFC 8032 specifies 3 versions of Ed25519 as well as 2 versions of Ed448. The 2 versions of both signature schemes most relevant to RFC 8032 provide for the introduction of context and specifically deal with pre-hashing the message to be signed to allow the schemes to be used with very large messages. For reasons we will look at soon, RFC 8032 recommends only using pre-hashing where strictly necessary.

We won't go into the exact specifics of the signature generation algorithm from RFC 8032 as they vary for each curve, however the following outline gives an indication of how signature generation actually works. Given a hash scheme, $Hash()$, specific to the curve, a context, C, of 0 to 255 octets, and the context length in octets $CLen$, the process is roughly as follows:

1. Set F to 0 for a pre-hash version of the algorithm, set F to 1 if the message is directly used.
2. Set DOM to be $classifier \parallel OCTET(F) \parallel OCTET(CLen) \parallel C$, where classifier is "SigEd448" for Ed448, and "SigEd25519 no Ed25519 collisions" for Ed25519, with both text strings been encoded as ASCII. The $OCTET()$ function simply converts its argument to a single octet.
3. Hash the private key, splitting the result into two halves, $H1$ and $H2$.
4. Set $S = (H1 \parallel A)$, where A is the public key as a byte array.
5. Set the prefix value P to be $H2$.
6. Set r to be the little endian integer resulting from $Hash(DOM(F, C) \parallel P \parallel M)$ where F is 0, or $Hash(DOM(F, C) \parallel P \parallel PreHash(M))$ in the pre-hash case where F is 1.
7. Calculate the point $r * B$, where B is the base point of the curve. Let R be the byte encoding of this point.
8. Set k to be the little endian integer resulting from $Hash(DOM(F, C) \parallel R \parallel A \parallel M)$ where F is 0, or $Hash(DOM(F, C) \parallel R \parallel A \parallel PreHash(M))$ in the pre-hash case where F is 1.
9. Calculate $S = (r + k * s) \mod L$.
10. Form the signature of the concatenation of R and the little-endian encoding of S.

The signatures are constant length: for Ed25519 both the R and S octet-strings should be 32 octets long and with Ed448 both strings are 57 octets long.

The idea behind the hash construction employed in the signature calculation is to try and avoid any vulnerabilities in the hash function used to create the signatures. As pre-hashing creates a single point vulnerability around any weaknesses in the hash function used for the pre-hash, while we can all hope for the best, it is safer to avoid pre-hashing if possible.

The following example code shows how to generate a signature using EdDSA:

```
/**
 * Generate an encoded EdDSA signature using the passed in EdDSA private key
 * and input data.
 *
 * @param edPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateEdDSASignature(
    PrivateKey edPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("EdDSA", "BC");

    signature.initSign(edPrivate);

    signature.update(input);

    return signature.sign();
}
```

The type of the signature is determined by the key, so in this case you can use the generic string "EdDSA" to specify which `Signature` implementation you want to use. The key used to initialize the `Signature` object will then be used to determine which signature type to create.

Signature Verification

Signature verification follows the same basic pattern as signature generation for EdDSA, still varies according to the curve used. The pattern is much simpler than signing and consists of 3 steps using the equivalent definition of DOM that was used for signing.

The basic steps are:

1. Split the signature into 2 halves, decoding the first half as a point R and the second half as an integer S.
2. Set $k = Hash(dom(F, C) \parallel R \parallel A \parallel PH(M))$ where A is the public key as byte array.
3. Check $S * B = R + k * A'$ where B is the base point of the curve, and A' is the public key as a point.

In the case of the JCA the key type is again used to determine which type of EdDSA signature is being processed, and you can see the following example code follows the usual pattern:

```
/**
 * Return true if the passed in EdDSA signature verifies against
 * the passed in EdDSA public key and input.
 *
 * @param edPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyEdDSASignature(
    PublicKey edPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("EdDSA", "BC");

    signature.initVerify(edPublic);

    signature.update(input);

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

Where you want to lock the signature being used to a specific curve, you can also use “Ed25519” and “Ed448” as the names of the signature algorithm passed to `Signature.getInstance()`.

DSTU 4145

Published in 2002, DSTU 4145 [115] is a Ukrainian standard built around elliptic curve. There is no RFC released for this standard yet, however its usage has been reported for the signing of PDFs and also CMS messages.

Key Pair Generation

DSTU 4145 provides 10 curve parameters sets, numbered 0 to 9, that are off the base branch “1.2.804.2.1.1.1.1.3.1.1.2”. The following code will set up and initialise a `KeyPairGenerator` for one of these parameter sets. Internally key pair generation involves the generation of a private value d and a public point Q which is the basis of the public key for the key pair.

```
/**
 * Generate a DSTU 4145-2002 key pair for the passed in named parameter set.
 *
 * @param curveNo the curve number to use (range [0-9])
 * @return a EC KeyPair
 */
public static KeyPair generateDSTU4145KeyPair(int curveNo)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance(
                                "DSTU4145", "BC");

    keyPair.initialize(
        new ECGenParameterSpec("1.2.804.2.1.1.1.1.3.1.1.2." + curveNo));

    return keyPair.generateKeyPair();
}
```

We should point out with this example, that if there is a friendlier way of naming things, we would love to hear about it, but we are at least confident that the use of the OBJECT IDENTIFIER arc is at least correct.

In Bouncy Castle, DSTU 4145 public and private keys carry implementations of the `DSTU4145PublicKey` and `DSTU4145PrivateKey` interfaces. These interfaces are only defined in the Bouncy Castle APIs.

Signature Generation

The signature generation algorithm makes use of a random value like ECDSA does, however the algorithm is specified using the EC field elements for the curve in use and is also little-endian in the way it converts the message hash $H(m)$ to an integer. The functions $FE()$ and $INT()$ are used to convert integers into field elements and back. The private value is still represented by d and the base point by G.

1. Calculate $h = FE(H(m))$. If $h == 0$ then $h = 1$.
2. Generate a random K such that $0 < K < N$
3. Calculate $(x, y) = KG$. If $x == 0$ go back to step 2.
4. Calculate $R = INT(hx)$. If $R == 0$ go back to step 2.
5. Calculate $S = (K + dR) \mod N$. If $S == 0$ go back to step 2.

The digest used for $H()$ is GOST R 34.11-94.

The following code provides an example of signature generation.

```java
/**
 * Generate an encoded DSTU 4145 signature based on the SM3 digest using the
 * passed in EC private key and input data.
 *
 * @param ecPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateDSTU4145Signature(
    PrivateKey ecPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("DSTU4145", "BC");

    signature.initSign(ecPrivate);

    signature.update(input);

    return signature.sign();
}
```

Unlike ECDSA, the encoded signature is not returned as ASN.1 INTEGER objects in a SEQUENCE. In this case ASN.1 encoding is still used, but the signature is encoded in an OCTET STRING which contains the encoding of the two signed values making up the signature. The values are also scaled so that they occupy exactly half the octets making up the OCTET STRING, so leading zeroes may be added in order to allow splitting the string so that the bytes making up the R and S values are easily extracted.

Signature Verification

After input validation, verification of a DSTU 4145 signature requires a three step process.

1. Calculate $h = FE(H(m))$. If $h == 0$ then $h = 1$.
2. Calculate $(x, y) = GS + QR$. If $((x, y)$ is at $\infty)$ return invalid.
3. verified = $(INT(hx) == R)$

From the higher level of the JCA this implementation ends up looking like:

```java
/**
 * Return true if the passed in DSTU 4145 signature verifies against
 * the passed in EC public key and input.
 *
 * @param ecPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyDSTU4145Signature(
    PublicKey ecPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("DSTU4145", "BC");

    signature.initVerify(ecPublic);

    signature.update(input);

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

Putting Everything Together

The following example shows the use of the methods we have defined for key pair generation, signature generation, and signature verification.

```
/**
 * An example of using DSTU 4145-2002 to sign data and then
 * verifying the resulting signature.
 */
public class DSTU4145Example
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        KeyPair ecKp = generateDSTU4145KeyPair(0);

        byte[] dstuSig = generateDSTU4145Signature(
            ecKp.getPrivate(), Strings.toByteArray("hello, world!"));

        System.out.println("DSTU 4145-2002 verified: " +
                    verifyDSTU4145Signature(
                        ecKp.getPublic(), Strings.toByteArray("hello, world!"),
                         dstuSig));
    }
}
```

Assuming everything goes well, if you run the example you should see the following output:

```
DSTU 4145-2002 verified: true
```

GOST

The GOST algorithms have followed a similar evolution to DSA, originally starting over big integer fields and later evolving to elliptic curve. The initial standards, GOST R 34.10-94 and the associated message digest GOST R 34.11-94, were released in 1994 and are also covered in RFC 4491 [62]. In 2001 an EC algorithm was added in GOST R 34.10-2001 which is also covered in RFC 5832 [79]. An updated version of both the signature algorithm and the message digest to use was then released in 2012 with GOST R 34.10-2012 (also covered in RFC 7091 [92]) and the digest GOST R 34.11-2012 “Streebog” (also covered in RFC 6986 [90]). In addition to Bouncy Castle’s implementation there is also a JavaScript implementation published at the WebCrypto GOST Library website [112].

The basic details for the differing GOST signature algorithms can be seen in the following table.

Standard	Key Algorithm Name	Digest Used	Signature Algorithm Name
GOST R 34.10-94	"GOST3410"	GOST R 34.11-94	"GOST3410"
GOST R 34.10-2001	"ECGOST3410"	GOST R 34.11-94	"ECGOST3410"
GOST R 34.10-2012	"ECGOST3410-2012"	GOST R 34.11-2012 (256)	"ECGOST3410-2012-256"
GOST R 34.10-2012	"ECGOST3410-2012"	GOST R 34.11-2012 (512)	"ECGOST3410-2012-512"

As you might have guessed, in addition to adding some bigger curve parameters GOST 34.10-2012 is also designed to work with 256 bit and 512 bit message digests.

One further note: like DSA/ECDSA all the GOST signature algorithms require the hash of the message being signed to be converted to a `BigInteger`, however unlike DSA/ECDSA the conversion is a little-endian one. If you are ever in a situation where you have to debug against a home grown implementation of a GOST signature algorithm that is not quite working, this is a good one to check for. Keeping that in mind we will now look at the algorithms in more detail.

The Basic Signature Algorithms

We will now take a look at the basic operations involved in the different GOST signature algorithms.

GOST R 34.10-94

The original GOST signature algorithm used domain parameters consisting of a prime P, a value Q which was a 254-256 bit prime factor of $P - 1$, and another value A which is any number less than $P - 1$ such that $A^Q \mod P = 1$. Once a set of domain parameters have been established a private value X is generated and then the public value Y is calculated as $Y = A^X \mod P$.

Like a DSA signature, a signature produced by GOST R 34.10-94 is made up of an R and S value. These values are calculated using the following steps:

1. Generate a random K such that $0 < K < Q$
2. Calculate $R = (A^K \mod P) \mod Q$
3. Calculate $S = (XR + K * H(m)) \mod Q$

Where $H(m)$ is a hash of the message m that we are generating the signature for. In the case of GOST R 34.10-94 H is defined as the algorithm described in GOST R 34.11-94.

Reflecting the change in the calculation of S the verification step is quite different to what we saw with DSA. After input validation, the following steps are required to verify a signature:

1. Calculate $V = H(m)^{Q-2} \mod Q$
2. Calculate $Z_1 = (SV) \mod Q$
3. Calculate $Z_2 = ((Q - R) * V) \mod Q$
4. Calculate $U = ((A^{Z_1} * Y^{Z_2}) \mod P) \mod Q$
5. verified = $(U == R)$

Note that unlike DSA, GOST R 34.10-94 produces an encoding containing two 256 bit unsigned numbers, padded with leading zeroes if necessary. There is no ASN.1 encoding involved and the signatures are always 64 bytes long.

In Bouncy Castle, GOST R 34.10-94 public and private keys carry implementations of the `GOST3410PublicKey` and `GOST3410PrivateKey` interfaces. These interfaces are only defined in the Bouncy Castle APIs.

GOST R 34.10-2001

GOST R 34.10-2001 is elliptic curve based. Like ECDSA, the algorithm makes use of the base point G, the curve's order N, a private value d and the public point Q. Unlike ECDSA it is not quite the same calculation - you can see from the following that step 4 is different.

1. Generate a random K such that $0 < K < N$
2. Calculate $(x, y) = KG$
3. Calculate $R = x \mod N$. If $R == 0$ go back to step 1.
4. Calculate $S = (K * H(m) + dR) \mod N$. If $S == 0$ go back to step 1.

As with GOST R 34.10-94, GOST R 34.10-2001 produces an encoding containing two 256 bit unsigned numbers, padded with leading zeroes if necessary. There is no ASN.1 encoding involved and the signatures are always 64 bytes long.

After input validation, signature verification consists of the following steps:

1. Calculate $V = H(m)^{-1} \mod N$
2. Calculate $Z_1 = (SV) \mod N$
3. Calculate $Z_2 = ((N - R) * V) \mod N$
4. Calculate $(x, y) = GZ_1 + QZ_2$. If $((x, y)$ is at $\infty)$ return invalid.
5. verified = $(x == R)$

In Bouncy Castle, GOST R 34.10-2001 public and private keys carry implementations of the `ECPublicKey` and `ECPrivateKey` interfaces.

GOST R 34.10-2012

The algorithms used for signing and verification by GOST R 34.10-2012 are the same as those for GOST R 34.10-2001. The difference in the new standard is that new curves and new message digests were added. The two new message digests are based on the "Streebog" hash function defined in RFC 6986 [90], which can produce digests that are 256 bits and 512 bits in length. Some of the new curves, in a similar fashion, are also larger bringing them into line with the level of security offered by the 512 bit version of GOST R 34.11-2012.

GOST R 34.10-2012 signatures can be either 512 bits or 1024 bits long. As with the earlier versions of the GOST R 34.10 standard the signatures are composed of 2 unsigned values representing the R and S values.

In Bouncy Castle, GOST R 34.10-2012 public and private keys also carry implementations of the `ECPublicKey` and `ECPrivateKey` interfaces.

Example Code - GOST R 34.10-2012

We have chosen to only look at GOST R 34.10-2012 in this section for two reasons. The first is the fact that, other than some name changes, from the JCA perspective, the generation of keys and the use of the algorithms are all basically the same. The second reason is that GOST R 34.10-94, GOST R 34.11-94, and GOST R 34.10-2001 are officially getting phased out in 2018, so it is better to concentrate of GOST R 34.10-2012.

Key Pair Generation

There are 4 principal parameter sets that can be used for generation of GOST keys: - Tc26-Gost-3410-12-256-paramSetA - Tc26-Gost-3410-12-512-paramSetA - Tc26-Gost-3410-12-512-paramSetB - Tc26-Gost-3410-12-512-paramSetC

As you might guess, the 256 and 512 numbers refer to the version of GOST R 34.11-2012 that is used for the particular parameter set.

In the JCA we make use of the `ECGenParameterSpec` class to pass in a parameter set name. The following example code will generate a GOST R 34.11-2012 key pair using one of the listed parameter names.

```
/**
 * Generate a GOST 3410-2012 key pair for the passed in named parameter set.
 *
 * @param paramSetName the name of the parameter set to base the key pair on.
 * @return a EC KeyPair
 */
public static KeyPair generateGOST3410_2012KeyPair(String paramSetName)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance(
                                "ECGOST3410-2012", "BC");

    keyPair.initialize(new ECGenParameterSpec(paramSetName));

    return keyPair.generateKeyPair();
}
```

Signature Generation

Signature generation for GOST R 34.11-2012 orientates around either the 256 bit or 512 bit digests as well. The two signature algorithms available are: - ECGOST3410-2012-256 - ECGOST3410-2012-512

The following method is written to take one of the two algorithm names given above as a parameter and generate a signature.

```
/**
 * Generate an encoded GOST 3410-2012 signature using the passed in
 * GOST 3410-2012 private key and input data.
 *
 * @param ecPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @param sigName the name of the signature algorithm to use.
 * @return the encoded signature.
 */
public static byte[] generateGOST3410_2012Signature(
    PrivateKey ecPrivate, byte[] input, String sigName)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance(sigName, "BC");

    signature.initSign(ecPrivate);

    signature.update(input);

    return signature.sign();
}
```

The two signature types produce signatures of a different size and rely on an appropriate curve been used. For the 256 bit digest based signature a 512 bit signature is produced consisting of R and S values. In the case of the 512 bit digest based signature a 1024 bit signature is produced.

Signature Verification

Our sample method for verification is also written to take the name of the algorithm being used. The code for that is below.

```
  /**
   * Return true if the passed in GOST 3410-2012 signature verifies against
   * the passed in GOST 3410-2012 public key and input.
   *
   * @param ecPublic the public key of the signature creator.
   * @param input the input that was supposed to have been signed.
   * @param sigName the name of the signature algorithm to use.
   * @param encSignature the encoded signature.
   * @return true if the signature verifies, false otherwise.
   */
  public static boolean verifyGOST3410_2012Signature(
      PublicKey ecPublic, byte[] input,
      String sigName, byte[] encSignature)
      throws GeneralSecurityException
  {
      Signature signature = Signature.getInstance(sigName, "BC");

      signature.initVerify(ecPublic);

      signature.update(input);

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

Putting Everything Together

Finally it would be worth seeing both these methods put together with an actual parameter set and signature algorithm specified.

The following sample code uses our defined methods and creates and verifies a signature using one of the 512 bit parameter sets.

```
/**
 * An example of using GOST R 34.10-2012 to sign data and then
 * verifying the resulting signature.
 */
public class GostR3410_2012Example
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        KeyPair ecKp = generateGOST3410_2012KeyPair(
                                    "Tc26-Gost-3410-12-512-paramSetA");

        byte[] ecGostSig = generateGOST3410_2012Signature(
            ecKp.getPrivate(), Strings.toByteArray("hello, world!"),
            "ECGOST3410-2012-512");

        System.err.println("ECGOST3410-2012-512 verified: " +
                    verifyGOST3410_2012Signature(
                        ecKp.getPublic(), Strings.toByteArray("hello, world!"),
                        "ECGOST3410-2012-512", ecGostSig));
    }
}
```

If everything goes according to plan you should see the following output:

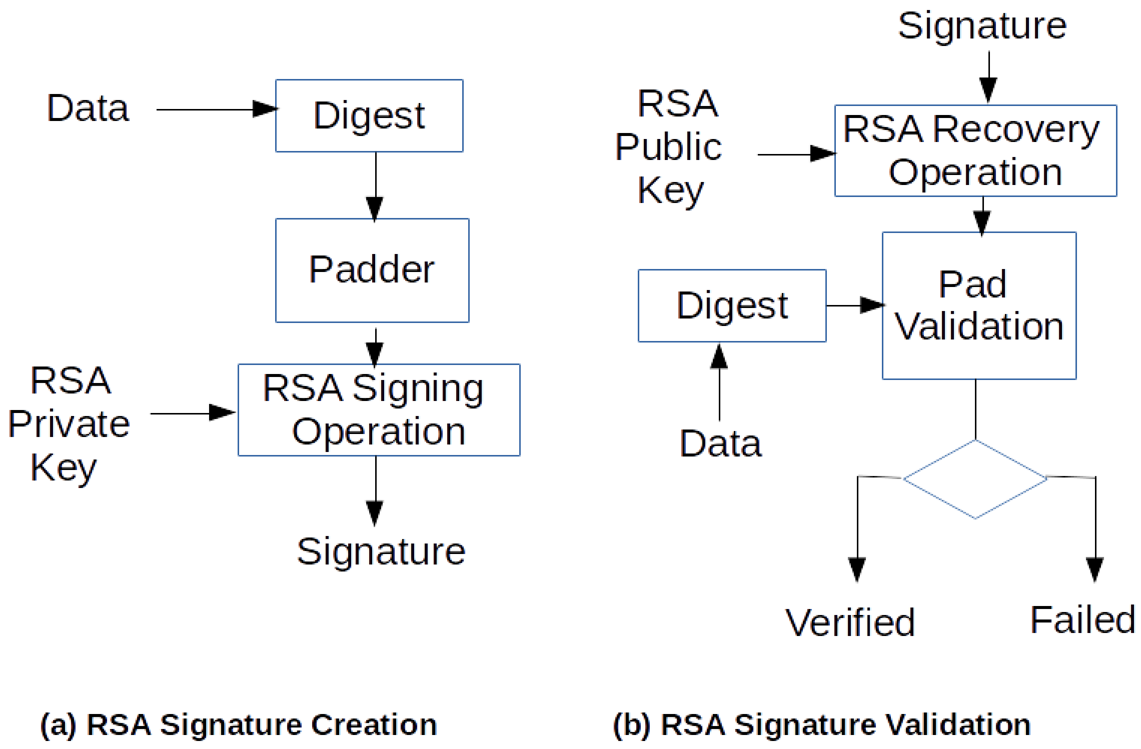
```
ECGOST3410-2012-512 verified: true
```

RSA Signature Algorithms

In many ways, to describe the RSA algorithm as a major workhorse for public key cryptography in the last 30 years would be to miss the point. The algorithm itself was initially discovered by Clifford Cocks in 1973 at GCHQ in Great Britain, but this work was not declassified until 1997. In the meanwhile the algorithm was also discovered independently and published in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman, which is where the RSA name comes from. RSA is used heavily, both for signature generation and also secret key transport - which we will look at later.

RSA key pair creation requires the calculation of a modulus value N which is the product of two appropriate primes P and Q. Having arrived at a modulus value we need to calculate the values E and D such that $ED \equiv 1 \mod ((P-1)(Q-1))$. The E value is referred to as the public exponent and the D value is referred to as the private exponent. For RSA to be used effectively the D value must be kept secret. The modulus N and the public exponent E, on the other hand, are quite safe to distribute. As this is the case in key generation the convention is to choose a public exponent value which will be most efficient for signature verification or encryption.

The reason making this choice is effective has to do with how basic signing and verification operations are carried out in using RSA. As with DSA/ECDSA we calculate a hash of the message being signed, and, in this case, as we are dealing with simple big integer exponentiation operations, some form of padding is introduced to ensure that the signature is appropriately secured by making it a number close to N, the modulus. We will refer to the padding function as $PAD()$ and the hash function as $H()$. We will see as we look at various methods of signing based on RSA that it is the padding function that changes; the basic arithmetic operations for creating a signature remain the same.



A schematic of RSA signature generation and verification

The basic signing operation for RSA can then be described as $S = PAD(H(M))^D \mod N$ and the verification operation is described as $PAD(H(M)) == S^E \mod N$[6]. You will probably realise when we get to Chapter 7 that this is the reverse of the encryption operation used with RSA. This can be useful if you are trying to understand what is in a signature you have been sent, possibly for debugging purposes. Decrypting an RSA signature with a public key will recover $PAD(H(M))$ and, in the event you have signatures mysteriously failing, it might help find what is going wrong.

Key Pair Generation

In the JCA it is possible to create RSA keys by either just giving the desired bit length for the modulus, or by providing the desired bit length and a public exponent to use. The algorithm parameter specification to use in the last case is the `RSAKeyGenParameterSpec`. In the example below the public exponent has been set to the Fermat number F4 (hex value 0x10001). The hex value helps explain why, from the point of a binary computer, the number is so friendly and efficient to use.

[6] In practice it is a little more complicated but the fundamentals are correct. We can use Chinese Remainder Theorem (CRT) to speed up the signing operation, but then we also need to introduce blinding to prevent timing attacks, and then there is Arjen Lenstra's CRT attack. It never ends.

```
/**
 * Generate a 2048 bit RSA key pair using user specified parameters.
 *
 * @return a RSA KeyPair
 */
public static KeyPair generateRSAKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA", "BC");

    keyPair.initialize(
        new RSAKeyGenParameterSpec(2048, RSAKeyGenParameterSpec.F4));

    return keyPair.generateKeyPair();
}
```

The `KeyPair` carrying the RSA keys will be made up of implementations of `RSAPublicKey` and `RSAPrivateKey` from the `java.security.intefaces` package. For most providers you will find that the private key also implements `RSAPrivateCrtKey` indicating it has the necessary components to make optimisation using Chinese Remainder Theorem possible.

If you examine the private key generated by running this method you will find that the number produced for the private exponent D is much larger than the public exponent we chose and, from a big integer math point of view, nowhere near as friendly.

PKCS #1 Version 1.5 Signatures

The original padding mechanism for RSA signatures was described in PKCS #1 and remained the only method described in the standard up until after PKCS #1 Version 1.5. The latest version of the standard can be found in RFC 8017 [99].

Signature Generation

Given the value h representing the digest of the message M to be signed. The padding function used by PKCS #1 Version 1.5 is defined as:

$$PAD(h) = 0x00 \parallel 0x01 \parallel F \parallel 0x00 \parallel DigInfo(h)$$

where F is a string of bytes which is at least 8 bytes long and where each byte has the value 0xFF. F has to be long enough to ensure that the length of the string produced by $PAD(h)$ is the same as the length of the modulus of the RSA key being used. The $DigInfo()$ function is defined as the byte string produced by DER encoding the following ASN.1 structure:

```
DigestInfo ::= SEQUENCE {
    digestAlgorithm DigestAlgorithm,
    digest OCTET STRING
}

DigestAlgorithm ::= AlgorithmIdentifier

AlgorithmIdentifier  ::=  SEQUENCE  {
    algorithm               OBJECT IDENTIFIER,
    parameters              ANY DEFINED BY algorithm OPTIONAL  }
```

Owing to a typo in an early version of an associated ASN.1 standard the optional parameters field is defined as being the ASN.1 NULL value[7]. So for example, if the signature was based on SHA-256 the DigestInfo block would be the DER encoding of the following structure:

```
SEQUENCE {
    SEQUENCE {
        OBJECT IDENTIFIER("2.16.840.1.101.3.4.2.1"),
        NULL }
    },
    OCTET STRING(SHA-256(M))
}
```

where SHA-256(M) represents the SHA-256 digest of the message M that we are signing.

This padding mechanism is known as type 1 padding and has the property of producing constant padding strings where it is the same document that is being signed. This can be quite useful as it makes it possible to see that two signatures are related to the same document if they were generated with the same key, unlike with regular DSA/ECDSA. It is this property which makes PKCS #1 Version 1.5 a deterministic signature scheme.

From a JCA perspective the use of the algorithm follows the pattern we have come to expect. The following code will generate a signature for the passed in input and private key using PKCS #1 Version 1.5:

[7] Note this is not the same as a Java `null`. ASN.1 NULL does have a specific encoding associated with it. For our purposes using a Java `null` in an ASN.1 field is equivalent to saying the field is absent.

```
/**
 * Generate an encoded RSA signature using the passed in private key and
 * input data.
 *
 * @param rsaPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generatePKCS1dot5Signature(
    PrivateKey rsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withRSA", "BC");

    signature.initSign(rsaPrivate);

    signature.update(input);

    return signature.sign();
}
```

The return value in this case is the byte encoding of the integer returned from the RSA operation, treated as an unsigned value and, if necessary, padded with leading zeroes to make it the same size as the modulus.

Signature Verification

Assuming the matching public key and the same input, the code to verify the signature generated in the previous example looks as follows:

```
/**
 * Return true if the passed in signature verifies against
 * the passed in RSA public key and input.
 *
 * @param rsaPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyPKCS1dot5Signature(
    PublicKey rsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withRSA", "BC");

    signature.initVerify(rsaPublic);

    signature.update(input);

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

Under the covers this process will apply the RSA verification operation after reconstructing $PAD(h)$ for the message we are verifying the signature against.

PSS Signatures

While there do not appear to have been any successful attacks against correctly formed and correctly implemented PKCS #1 Version 1.5 signatures, advances in cryptanalysis have suggested it might be better to implement the $PAD(h)$ function used by RSA to include a level of randomness in the signature. Enter the Probabilistic Signature Scheme (PSS) function. Originally proposed in “PSS: Provably Secure Encoding Method for Digital Signatures” by Bellare and Rogaway [117], the scheme is provably secure and was adapted for PKCS #1 Version 2 and appears in RFC 8017 [99] as RSA-PSS.

Signature Generation

Unlike the padding mechanism used in PKCS #1 Version 1.5, the RSA-PSS scheme also includes an optional $salt$, of length $sLen$, and $MGF()$ a mask generation function. Given our message digest value h produced using the digest function $Digest()$ with output length $hLen$ and an RSA modulus $modBits$ in length we can describe the steps of the new padding function $PAD()$ as follows:

1. Set $maxLen = \lceil (modBits - 1)/8 \rceil$
2. Generate a random octet string $salt$ of length $sLen$; if $sLen == 0$, then $salt$ is the empty string.
3. Set $M' = Z \parallel h \parallel salt$ where Z is eight zero octets.
4. Set $H = Digest(M')$.
5. Set $DB = PS \parallel 0x01 \parallel salt$ where PS is a string of zero octets to bring the length of DB to maxLen - hLen - 1.
6. Set $dbMask = MGF(H, maxLen - hLen - 1)$.
7. Set $maskedDB = DB \oplus dbMask$.
8. Set $EM = maskedDB \parallel H \parallel 0xBC$.
9. Set the leftmost $8 * maxLen - (modBits - 1)$ bits of the leftmost octet in EM to zero.
10. Return EM.

You can find the definition for the standard mask generation function (MGF1) in RFC 8017, Section B.2.1. The source for an implementation of the MGF1 is also available in Bouncy Castle.

As you might have guessed if the salt length is zero, then the signature produced is deterministic. Otherwise the signature will change even if the key and the input data are the same. Decreasing the salt length does lower the security of the algorithm as discussed by Coron in [118] although it is not currently believed it is much of a risk to do so. RFC 8017 gives the rough rule of thumb that salt length is optimal for a given hash algorithm if the bit length of the seed is the same length as the bit length of the output of the hash function.

The following example shows the basic use of RSA-PSS in the JCA:

```java
/**
 * Generate an encoded RSA signature using the passed in private key and
 * input data.
 *
 * @param rsaPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateRSAPSSSignature(
    PrivateKey rsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withRSAandMGF1", "BC");

    signature.initSign(rsaPrivate);

    signature.update(input);

    return signature.sign();
}
```

In the example above internal defaults will be used for selecting the seed size. We will see a bit later on that there is a way of configuring a `Signature` class using an `AlgorithmParameterSpec` as well.

Signature Verification

Signature verification involves recovering the padded string and then reversing the steps used in the padding generation after validating the recovered string by checking the lengths involved and confirming the expected zero bits at the start of the string and the presence of the 0xBC byte at the end.

In the JCA the verification step appears as follows:

```
/**
 * Return true if the passed in signature verifies against
 * the passed in RSA public key and input.
 *
 * @param rsaPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyRSAPSSSignature(
    PublicKey rsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA256withRSAandMGF1", "BC");

    signature.initVerify(rsaPublic);

    signature.update(input);

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

Under the covers things are more involved. Taking the recovered byte string as the string EM, the algorithm used looks like this:

1. Set maskedDB be the leftmost $maxLen - hLen - 1$ octets of EM, and let H be the next $hLen$ octets.
2. Set $dbMask = MGF(H, maxLen - hLen - 1)$.
3. Set $DB = maskedDB \oplus dbMask$.
4. Set the leftmost $8 * maxLen - (modBits - 1)$ bits of the leftmost octet in DB to zero.
5. If the $maxLen - hLen - sLen - 2$ leftmost octets of DB are not zero or if the octet at position $maxLen - hLen - sLen - 1 \neq 0x01$, return invalid.
6. Set $salt$ be the last $sLen$ octets of DB.
7. Set $M' = Z \parallel h \parallel salt$ where Z is eight zero octets.
8. Set $H = Digest(M')$.
9. verified = $H == H'$.

As you can see looking at the algorithm, the salt length is not stored in the signature, but is a configuration parameter. More importantly, if you have the wrong salt length, things are not going to end very well. The two most common issues with verifying RSA-PSS signatures which should otherwise be correct are mismatching the salt length and people using unexpected digests for the $MGF()$ function - ideally the digest used for the MGF function should be the same as the digest used for processing the message as they will have the same security strength.

Using PSSParameterSpec

The JCA provides the class java.security.spec.PSSParameterSpec as a mechanism for configuring a Signature object with the appropriate details in regards to message digest, salt, mask generation function and the trailer.

Generating a signature with the PSSParameterSpec is just a matter of calling Signature.setParameter() with a PSSParameterSpec object.

```java
/**
 * Generate an encoded RSA signature using the passed in private key and
 * input data.
 *
 * @param rsaPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateRSAPSSSignature(
    PrivateKey rsaPrivate, PSSParameterSpec pssSpec, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("RSAPSS", "BC");

    signature.setParameter(pssSpec);

    signature.initSign(rsaPrivate);

    signature.update(input);

    return signature.sign();
}
```

As we can see below, the same applies for verification.

```java
/**
 * Return true if the passed in signature verifies against
 * the passed in RSA public key and input.
 *
 * @param rsaPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
```

```
    public static boolean verifyRSAPSSSignature(
        PublicKey rsaPublic, PSSParameterSpec pssSpec,
        byte[] input, byte[] encSignature)
        throws GeneralSecurityException
    {
        Signature signature = Signature.getInstance("RSAPSS", "BC");

        signature.setParameter(pssSpec);

        signature.initVerify(rsaPublic);

        signature.update(input);

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

Note, in both cases the `setParameter()` method is called before the init() methods.

The following example uses our two example methods to create and verify a signature that would otherwise be specified as “SHA256withRSAandMGF1”. The first argument to the `PSSParameterSpec` constructor provides the signature message digest, the next argument provides the name of the mask generation function with the argument after giving the message digest to use for it. Finally there is the argument specifying the trailer - in this case 1 indicating that the trailer value 0xBC should be used[8].

```
/**
 * An example of using RSA PSS with a PSSParameterSpec based on SHA-256.
 */
public class RSAPSSParamsExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        KeyPair rsaKp = generateRSAKeyPair();
        PSSParameterSpec pssSpec = new PSSParameterSpec(
            "SHA-256",
            "MGF1", new MGF1ParameterSpec("SHA-256"), 32,
            1);

        byte[] pssSignature = generateRSAPSSSignature(
            rsaKp.getPrivate(), pssSpec, Strings.toByteArray("hello, world!"));

        System.out.println("RSA PSS verified: "
```

[8] We are unaware of any other acceptable value for this field.

```
                                + verifyRSAPSSSignature(
                                        rsaKp.getPublic(), pssSpec,
                                        Strings.toByteArray("hello, world!"),
                                        pssSignature));
    }
}
```

You can see that the parameter specification provides everything required as the signature algorithm in the verify and generate methods is simply called RSAPSS.

Assuming all is well, if you run the example you should see the following output:

```
RSA PSS verified: true
```

Other RSA Signature Algorithms

Bouncy Castle also supports three more RSA signature types: two based on the ISO9796-2 [21] and one from the X9.31 - which is also listed in FIPS PUB 186-4 [2] and originally defined in X9.31 [22].

Standard	Algorithm Name Template
ISO9796-2	<digest>withRSA/ISO9796-2
ISO9796-2	<digest>withRSAandMGF1/ISO9796-2
X9.31	<digest>withRSA/X9.31

As you can see, other than the type name following the "/", the name templates follow the usual Java convention. For example, a SHA-256 based signature using the PSS variant of ISO9792-2 is "SHA256withRSAandMGF1/ISO9796-2". Likewise creating an instance of a `Signature` class using SHA-256 with X9.31 uses the name "SHA256withRSA/X9.31".

Note also that at the moment there is no support for using PSSParameterSpec with the ISO9796-2 PSS variant.

SM2

SM2 is a signature algorithm based on elliptic curve cryptography published by the Chinese Commercial Cryptography Administration Office. At the moment there are only draft RFCs available for the SM2 [42] and the message digest it uses SM3 [43]. Another source of information on SM2 and SM3 is the Peking University project hosted at http://gmssl.org [113]. Both the draft RFCs and the Peking University website include references to the original Chinese standards.

The SM2 signature algorithm specifies the use of the SM3 message digest. SM2 also differs from other EC based signing algorithms in a number of respects. The most fundamental difference is that the algorithm preprocesses the message by prepending a constant string based on the identity of the signer and the signer's public key. We will have a look at the preprocessing step first.

Message Preprocessing Step

The first step to signing or verifying a message is to calculate the message prefix. The message prefix is based on an identity string for the user ID, the encoding of the field elements A and B from the EC curve parameters of the curve being used, the encodings of the elements of the base point G from the curve being used, and the encodings of the elements making up the public key Q of the signer.

1. Calculate $ZA = H(bitLength(ID) \parallel ID \parallel A \parallel B \parallel xG \parallel yG \parallel xQ \parallel yQ)$
2. Calculate $m' = ZA \parallel m$

where $bitLength(ID)$ is a two byte integer representing the length of ID in bits and $H()$ represents the SM3 digest.

Where no ID is specified, SM2 uses a default ID string represented by the byte values in the hex string "31323334353637383132333435363738".

Signature Generation

Once a message m has been preprocessed into m' we are ready to calculate a signature using the following 4 steps.

1. Generate a random K such that $0 < K < N$
2. Calculate $(x, y) = KG$
3. Calculate $R = (H(m') + x) \mod N$ if $R == 0$ or $R + K == N$ go back to step 1.
4. Calculate $S = ((1 + d)^{-1} * (K - dR)) \mod N$ if $S == 0$ go back to step 1.

As usual, despite the algorithm differences, the SM2 signature algorithm looks quite familiar when carried out using the JCA.

```
/**
 * Generate an encoded SM2 signature based on the SM3 digest using the
 * passed in EC private key and input data.
 *
 * @param ecPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateSM2Signature(
    PrivateKey ecPrivate, byte[] input)
```

```
        throws GeneralSecurityException
    {
        Signature signature = Signature.getInstance("SM3withSM2", "BC");

        signature.initSign(ecPrivate);

        signature.update(input);

        return signature.sign();
    }
```

Like ECDSA, SM2 produces a byte array representing the signature by encoding R and S as two ASN.1 INTEGER types in an ASN.1 SEQUENCE. Although we are yet to encounter a faulty signature for this one, the usual caveats concerning the fact INTEGER is a signed type will still apply.

Signature Verification

After bounds checking of R and S and the preprocessing of the message m to produce m' we have 3 steps to determining whether a signature matches the message and public key provided.

1. Calculate $T = (R + S) \mod N$. if $T == 0$ return invalid.
2. Calculate $(x, y) = SG + TQ$.
3. verified = $(R == ((H(m') + x) \mod N))$

```
    /**
     * Return true if the passed in SM3withSM2 signature verifies against
     * the passed in EC public key and input.
     *
     * @param ecPublic the public key of the signature creator.
     * @param input the input that was supposed to have been signed.
     * @param encSignature the encoded signature.
     * @return true if the signature verifies, false otherwise.
     */
    public static boolean verifySM2Signature(
        PublicKey ecPublic, byte[] input, byte[] encSignature)
        throws GeneralSecurityException
    {
        Signature signature = Signature.getInstance("SM3withSM2", "BC");

        signature.initVerify(ecPublic);

        signature.update(input);

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

Using SM2ParameterSpec

In the examples we looked at before, the `Signature` object was using the default SM2 ID value internally. Bouncy Castle also supports a parameter specification - the `SM2ParameterSpec` to allow the ID string used for signature calculation to be set explicitly.

The following method shows how to set the ID for signature generation on line 18.

```
 1  /**
 2   * Generate an encoded SM2 signature based on the SM3 digest using the
 3   * passed in EC private key and input data.
 4   *
 5   * @param ecPrivate the private key for generating the signature with.
 6   * @param sm2Spec the SM2 specification carrying the ID of the signer.
 7   * @param input the input to be signed.
 8   * @return the encoded signature.
 9   */
10  public static byte[] generateSM2Signature(
11      PrivateKey ecPrivate, SM2ParameterSpec sm2Spec, byte[] input)
12      throws GeneralSecurityException
13  {
14      Signature signature = Signature.getInstance("SM3withSM2", "BC");
15
16      signature.setParameter(sm2Spec);
17
18      signature.initSign(ecPrivate);
19
20      signature.update(input);
21
22      return signature.sign();
23  }
```

And the next method shows the process for setting the ID for signature verification.

```
  /**
   * Return true if the passed in SM3withSM2 signature verifies against
   * the passed in EC public key and input.
   *
   * @param ecPublic the public key of the signature creator.
   * @param sm2Spec the SM2 specification carrying the expected ID of the signer.
   * @param input the input that was supposed to have been signed.
   * @param encSignature the encoded signature.
   * @return true if the signature verifies, false otherwise.
   */
  public static boolean verifySM2Signature(
      PublicKey ecPublic, SM2ParameterSpec sm2Spec,
      byte[] input, byte[] encSignature)
      throws GeneralSecurityException
  {
      Signature signature = Signature.getInstance("SM3withSM2", "BC");

      signature.setParameter(sm2Spec);

      signature.initVerify(ecPublic);

      signature.update(input);

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

Finally here is an example of putting both methods together with an ID value included.

```
/**
 * An example of using SM2 with an SM2ParameterSpec to specify the ID string
 * for the signature.
 */
public class SM2ParamSpecExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        KeyPair ecKp = generateECKeyPair("sm2p256v1");

        SM2ParameterSpec sm2Spec = new SM2ParameterSpec(
                                     Strings.toByteArray("Signer@Octets.ID"));

        byte[] sm2Signature = generateSM2Signature(
                                  ecKp.getPrivate(), sm2Spec,
                                  Strings.toByteArray("hello, world!"));

        System.out.println("SM2 verified: "
              + verifySM2Signature(
                  ecKp.getPublic(), sm2Spec,
                  Strings.toByteArray("hello, world!"), sm2Signature));
    }
}
```

If you run this example and all goes well you see the output

```
SM2 verified: true
```

As you would expect if you try changing the ID string for either generation or verification without making the corresponding change to the other side of the process, you will see the signature verification step will fail.

Bouncy Castle Calculator Interfaces for Signatures

Bouncy Castle's PKIX package provides general interfaces for operators representing generators and verifiers of signatures. The interfaces are both stream based and provide methods on for retrieving the ASN.1 AlgorithmParameters associated with the algorithm concerned, an output stream to write the data to be processed to, as well as a method for retrieving the created signature, or verifying an existing signature when the stream has been closed.

These interfaces are used throughout the PKIX APIs for introducing signature and verification operations and implementations for this interface exist both for the JCA and the BC low-level APIs. You can also write your own backed by other cryptography APIs if you need to.

The ContentSigner and ContentVerifier Interfaces

The `ContentSigner` interface is given below:

```java
/**
 * General interface for an operator that is able to create a signature from
 * a stream of output.
 */
public interface ContentSigner
{
    /**
     * Return the algorithm identifier describing the signature
     * algorithm and parameters this signer generates.
     *
     * @return algorithm oid and parameters.
     */
    AlgorithmIdentifier getAlgorithmIdentifier();

    /**
     * Returns a stream that will accept data for the purpose of calculating
     * a signature. Use TeeOutputStream if you want to accumulate the data
     * on the fly as well.
     *
     * @return an OutputStream
     */
    OutputStream getOutputStream();

    /**
     * Returns a signature based on the current data written to the stream,
     * since the start or the last call to getSignature().
     *
     * @return bytes representing the signature.
     */
    byte[] getSignature();
}
```

The `ContentVerifier` follows a similar pattern.

```java
/**
 * General interface for an operator that is able to verify a signature based
 * on data in a stream of output.
 */
public interface ContentVerifier
{
    /**
     * Return the algorithm identifier describing the signature
     * algorithm and parameters this verifier supports.
     *
     * @return algorithm oid and parameters.
     */
    AlgorithmIdentifier getAlgorithmIdentifier();

    /**
     * Returns a stream that will accept data for the purpose of calculating
     * a signature for later verification. Use TeeOutputStream if you want
     * to accumulate the data on the fly as well.
     *
     * @return an OutputStream
     */
    OutputStream getOutputStream();

    /**
     * Return true if the expected value of the signature matches the data
     * passed into the stream.
     *
     * @param expected expected value of the signature on the data.
     * @return true if the signature verifies, false otherwise
     */
    boolean verify(byte[] expected);
}
```

As the type of signature being processed is often unknown until the AlgorithmIdentifier for the signature has been analysed, you would normally create one of these from a `ContentVerifierProvider`. In a situation where a high-level PKIX class needs to create a `ContentVerifierProvider` you will normally find a method taking a `ContentVerifierProvider` as an argument. For this reason if you need to provide your own implementation of one of these as you are interfacing with other software or hardware to do the actual work of verifying a signature, you would normally implement the `ContentVerifierProvider` as well.

The `ContentVerifierProvider` interface is also based on the `AlgorithmIdentifier` class and has three methods.

```
/**
 * General interface for providers of ContentVerifier objects.
 */
public interface ContentVerifierProvider
{
    /**
     * Return whether or not this verifier has a certificate associated with it.
     *
     * @return true if there is an associated certificate, false otherwise.
     */
    boolean hasAssociatedCertificate();

    /**
     * Return the associated certificate if there is one.
     *
     * @return a holder containing the associated certificate if there is one,
     * null if there is not.
     */
    X509CertificateHolder getAssociatedCertificate();

    /**
     * Return a ContentVerifier that matches the passed in algorithm identifier,
     *
     * @param verifierAlgorithmIdentifier the algorithm and parameters required.
     * @return a matching ContentVerifier
     * @throws OperatorCreationException if the required ContentVerifier cannot
     * be created.
     */
    ContentVerifier get(AlgorithmIdentifier verifierAlgorithmIdentifier)
        throws OperatorCreationException;
}
```

Summary

In this chapter we have looked at the security levels and fundamentals around working with public key Signatures. This covers the creation and verification of digital signatures using various schemes such as DSA, DSTU 4145, GOST and the most well known algorithm of all, RSA.

Finally, we looked at the operator interfaces used for the Bouncy Castle PKIX package. This has interfaces that include ContentSigner and ContentVerifier allowing for provider independent signature operations.

Chapter 7: Key Transport, Key Agreement, and Key Exchange

This chapter shows how public key cryptography is applied to key transport, key agreement, and key exchange. We will look at key transport using the same API for doing encryption based on symmetric ciphers that we saw in Chapter 2. After that we will look at key agreement and key exchange.

Algorithm Security Strengths

The general security strengths for algorithms involved in supporting key transport, key agreement, and key exchange follow the same guidelines as for signature algorithms that we saw in Chapter 6. For RSA and Elliptic Curve you can follow the same values given in Chapter 6 for RSA signatures and ECDSA.

There is one new algorithm here which we did not cover in the signature chapter: Diffie-Hellman over prime numbers, which mostly uses the same kind of parameters as DSA. The security strengths for key agreement using Diffie-Hellman applicable to NIST standards are also described in Table 2 of NIST SP 800-57 [12]. For the range of security strengths we are interested in the key size table looks like:

Key Agreement Security Strengths

Public Key Algorithm	Key Size	Security Strength (bits)
DH	1024	<= 80
DH	2048	112
DH	3072	128
DH	7680[9]	192
DH	15360[9]	256

We will also see that digest functions are used in many of the techniques described in this chapter. The same rule concerning the relative security strength of a digest compared to the public key algorithm that was expressed in Chapter 6 also applies here.

Key Transport

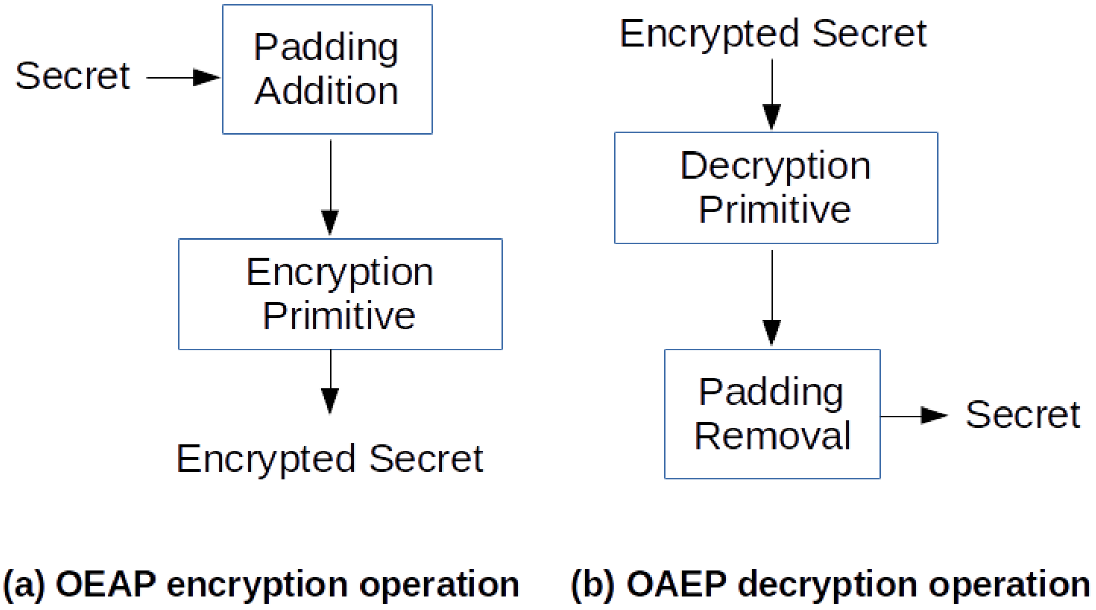
For our purposes here, key transport is a term used to describe the process of getting a symmetric key safely to another party after generating it locally. Without considering post-quantum algorithms, RSA and ElGamal are the only two algorithms that directly offer this ability as their construction allows the

[9] these key sizes are not currently used in the NIST standards.

encryption of a single plain text block at one end and then the recovery of the original plain text block at the other.

RSA-OAEP

The RSA encryption/decryption steps perform the opposite operations to the RSA signing/verification. Given a plaintext message M, the encryption operation for RSA can then be described as $Enc == PAD(M)^E \mod N$ and the decryption operation is $PAD(M) = Enc^D \mod N$, where N is the RSA modulus, E is the public exponent, and D is the private exponent.



Schematics of RSA-OAEP for encryption and decryption

The OAEP padding function was introduced by Bellare and Rogaway in "Optimal Asymmetric Encryption - How to Encrypt with RSA" [116] and later adopted for PKCS #1 [99] starting with Version 2.0. Ideally with an algorithm like RSA the encrypted value would be a random number only slightly smaller than the modulus - at this point any attacks based around known ciphertext (either through knowing the pre-padded message, or knowing something about the padding itself). OAEP does not go quite all the way in doing this, but it gets very close and still provides a non-malleable approach towards encrypting a message and also allows the encryptions to be tagged with a label if desired.

OAEP Key Wrapping

Wrapping a symmetric key for encryption using OAEP involves taking the symmetric key bytes as M, constructing a padded block using OAEP encoding and then applying the RSA encryption operation to produce the ciphertext. The encoding is similar in construction to the PSS mechanism we looked at earlier. In addition to the message M, the OAEP padding also takes a label L - which can be used to tag the encryption or be zero length if not used. Given the two inputs and a digest function $Digest()$ with output length $hLen$ and an RSA modulus $modBits$ in length the encoding algorithm for OAEP padding can be described as follows:

1. Set $maxLen = \lceil (modBits - 1)/8 \rceil$
2. Set $lHash = Digest(L)$
3. Set $DB = lHash \parallel PS \parallel 0x01 \parallel M$, where PS is a string of enough zeroes, possibly empty, to bring the length of DB to $maxLen - hLen$.
4. Generate a random octet string $seed$ of length $hLen$.
5. Set $dbMask = MGF(seed, maxLen - hLen)$.
6. Set $maskedDB = DB \oplus dbMask$.
7. Set $seedMask = MGF(maskedDB, hLen)$.
8. Set $maskedSeed = seed \oplus seedMask$.
9. Set $EM = 0x00 \parallel maskSeed \parallel maskedDB$.
10. Return EM.

From the JCE standpoint, the name of the digest function and the name of the MGF function used are captured in the padding field for the cipher name. As we are dealing with RSA the mode is specified as NONE[10]. In the case of the example that follows our digest function is SHA-256 using the MGF1 mask generation function. The same digest will be used by the mask generation function.

```java
/**
 * Generate a wrapped key using the RSA OAEP algorithm,
 * returning the resulting encryption.
 *
 * @param rsaPublic the public key to base the wrapping on.
 * @param secretKey the secret key to be encrypted/wrapped.
 * @return true if the signature verifies, false otherwise.
 */
public static byte[] keyWrapOAEP(
    PublicKey rsaPublic, SecretKey secretKey)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance(
        "RSA/NONE/OAEPwithSHA256andMGF1Padding", "BC");

    cipher.init(Cipher.WRAP_MODE, rsaPublic);

    return cipher.wrap(secretKey);
}
```

[10] For historical reasons you will also see the mode name ECB. While in a sense this is true, NONE really is a lot more appropriate as ECB makes RSA sound like a regular block cipher, which RSA is definitely not.

As the `Cipher` object in the example has been initialized in wrap mode, only the `Cipher.wrap()` method in enabled. Using `Cipher.update()` or `Cipher.doFinal()` will cause an exception.

OAEP Key Unwrapping

The unwrapping process involves applying the RSA decryption operation to the ciphertext and then reversing the OAEP padding function to produce the original message M which provides us with the bytes making up the symmetric key.

1. Set $maxLen = \lceil (modBits - 1)/8 \rceil$
2. Set $lHash = Digest(L)$
3. Separate EM into $Y \parallel maskSeed \parallel maskedDB$ where Y is a single octet, $maskSeed$ is $hLen$ octets long, and $maskedDB$ is the remaining octets.
4. Set $seedMask = MGF(maskedDB, hLen)$.
5. Set $seed = maskedSeed \oplus seedMask$.
6. Set $dbMask = MGF(seed, maxLen - hLen)$.
7. Set $DB = maskedDB \oplus dbMask$.
8. Separate DB into $lHash' \parallel PS \parallel Z \parallel M$ where $lHash'$ is $hLen$ octets long, PS is a string of zero or more octets with the value 0x00, Z is a single octet, and M is the remaining octets.
9. If $lHash \neq lHash'$ or $Y \neq 0x00$ or $Z \neq 0x01$ return error.
10. Return M.

For the JCE, in the simple case, we just use the `Cipher` class with the same configuration as we had before and use the `Cipher.unwrap()` method. Here is the corresponding unwrap method for the wrap method we saw earlier.

```java
/**
 * Return the secret key that was encrypted in wrappedKey.
 *
 * @param rsaPrivate the private key to use for the unwrap.
 * @param wrappedKey the encrypted secret key.
 * @param keyAlgorithm the algorithm that the encrypted key is for.
 * @return the unwrapped SecretKey.
 */
public static SecretKey keyUnwrapOAEP(
    PrivateKey rsaPrivate, byte[] wrappedKey, String keyAlgorithm)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance(
        "RSA/NONE/OAEPwithSHA256andMGF1Padding", "BC");

    cipher.init(Cipher.UNWRAP_MODE, rsaPrivate);
```

```java

      return (SecretKey)cipher.unwrap(
                            wrappedKey, keyAlgorithm, Cipher.SECRET_KEY);
  }
```

Putting Everything Together

The following example shows the use of the methods we have defined for RSA-OAEP key wrapping and key unwrapping to wrap and unwrap an AES key.

```java
/**
 * Simple example showing secret key wrapping and unwrapping based on OAEP.
 */
public class OAEPExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        SecretKey aesKey = createTestAESKey();

        KeyPair kp = generateRSAKeyPair();

        byte[] wrappedKey = keyWrapOAEP(kp.getPublic(), aesKey);

        SecretKey recoveredKey = keyUnwrapOAEP(
                                    kp.getPrivate(),
                                    wrappedKey, aesKey.getAlgorithm());

        System.out.println(
            Arrays.areEqual(aesKey.getEncoded(), recoveredKey.getEncoded()));
    }
}
```

If all is done correctly, running the example should produce the following output showing that the original key bytes have been correctly recovered:

```
true
```

Using OAEPParameterSpec

As you would have seen in the description of encoding algorithm for OAEP padding, there is provision for a label L to be included in the encryption. The JCE allows you to set this value, as well as to configure the OAEP padding in general by using the `javax.crypto.spec.OAEPParameterSpec` and the `javax.crypto.spec.PSource.PSpecified` classes.

Allowing for the use of the `OAEPParameterSpec` requires us to make allowance for passing the algorithm parameter specification to the `Cipher.init()` method. We can do this by adjusting the key wrapping method as follows:

```
/**
 * Generate a wrapped key using the RSA OAEP algorithm according
 * to the passed in OAEPParameterSpec and return the resulting encryption.
 *
 * @param rsaPublic the public key to base the wrapping on.
 * @param oaepSpec the parameter specification for the OAEP operation.
 * @param secretKey the secret key to be encrypted/wrapped.
 * @return true if the signature verifies, false otherwise.
 */
public static byte[] keyWrapOAEP(
    PublicKey rsaPublic, OAEPParameterSpec oaepSpec, SecretKey secretKey)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("RSA", "BC");

    cipher.init(Cipher.WRAP_MODE, rsaPublic, oaepSpec);

    return cipher.wrap(secretKey);
}
```

and making the equivalent change to the key unwrapping method as below:

```
/**
 * Return the secret key that was encrypted in wrappedKey.
 *
 * @param rsaPrivate the private key to use for the unwrap.
 * @param oaepSpec the parameter specification for the OAEP operation.
 * @param wrappedKey the encrypted secret key.
 * @param keyAlgorithm the algorithm that the encrypted key is for.
 * @return the unwrapped SecretKey.
 */
public static SecretKey keyUnwrapOAEP(
    PrivateKey rsaPrivate, OAEPParameterSpec oaepSpec,
    byte[] wrappedKey, String keyAlgorithm)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("RSA", "BC");

    cipher.init(Cipher.UNWRAP_MODE, rsaPrivate, oaepSpec);

    return (SecretKey)cipher.unwrap(
                         wrappedKey, keyAlgorithm, Cipher.SECRET_KEY);
}
```

Note that in both the above, as the `OAEPParameterSpec` provides all the required configuration for using RSA-OAEP, we only need to specify the base algorithm name "RSA" to the `Cipher.getInstance()` method. The passing of `oaepSpec` to the `Cipher.init()` method does the rest.

The following example code shows the use of the `OAEPParameterSpec` to configure the RSA cipher to use RSA-OAEP based on SHA-256, with MGF1 based on SHA-256 as the mask generation function as well. The only real difference between this example of RSA-OAEP and the previous one we looked at, which did not use `OAEPParameterSpec`, is the inclusion of a label defined by the bytes making up the string "My Label".

```
/**
 * Simple example showing secret key wrapping and unwrapping based on OAEP
 * and using the OAEPParameterSpec class to configure the encryption.
 */
public class OAEPParamsExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        SecretKey aesKey = createTestAESKey();

        KeyPair kp = generateRSAKeyPair();
        OAEPParameterSpec oaepSpec = new OAEPParameterSpec(
                                        "SHA-256",
                                        "MGF1", MGF1ParameterSpec.SHA256,
                                            new PSource.PSpecified(
                                              Strings.toByteArray("My Label")));

        byte[] wrappedKey = keyWrapOAEP(kp.getPublic(), aesKey);

        SecretKey recoveredKey = keyUnwrapOAEP(
                                    kp.getPrivate(), oaepSpec,
                                    wrappedKey, aesKey.getAlgorithm());

        System.out.println(
            Arrays.areEqual(aesKey.getEncoded(), recoveredKey.getEncoded()));
    }
}
```

As with the previous RSA-OAEP example, running the example should produce the following output:

```
true
```

showing that the original key bytes have been correctly recovered.

If you are curious you can change the `oaepSpec` value between the wrap and the unwrap and you will see that the unwrap fails with an exception. Finally, if you wanted to get the `oeapSpec` value back to exactly what it needs to be to produce encryption when we were not using `OAEPParameterSpec`, you can use `PSource.PSpecified.DEFAULT` as the label - this constant provides a label source equating to the empty string.

ElGamal-OAEP

Like RSA, the ElGamal[11] algorithm also provides the ability to process a single block of data. Based on the Diffie-Hellman algorithm it produces a block equal to twice the size of the key containing the encrypted data.

ElGamal uses regular Diffie-Hellman keys. Diffie-Hellman domain parameters are the same as those we saw for DSA in Chapter 6 - in this case the P and G parameters are of the most interest. Once a set of domain parameters have been chosen a private value X where $0 < X < P$[12] and the public value Y is generated by calculating G^X.

Given a message M and a public key containing Y, P, and G the encryption step takes a regular Diffie-Hellman key pair and goes through the following steps:

1. Generate a random K such that $0 < K < P - 1$
2. Calculate $Gamma = G^K \mod P$
3. Calculate $Phi = (PAD(M) * Y^K) \mod P$
4. Return $(Gamma, Phi)$

Where the $(Gamma, Phi)$ pair are returned as unsigned integers, padded with leading zeroes, if necessary, so that each one has an encoding that is the same length of the key. The $PAD()$ function is the OAEP function we saw defined earlier.

On decryption, once $Gamma$ and Phi have been recovered from the encoding, there is a single step to recover M using the private value X and the domain parameter P.

[11] With apologies to Taher Elgamal, the algorithm's creator, we have followed the usual capitalization of the algorithm name.

[12] Normally X is chosen in a limited range of the possible values for security and efficiency reasons. We will look at this more closely with keys for Diffie-Hellman key agreement later.

1. Calculate $PAD(M) = (Gamma^{P-1-X} * Phi) \mod P$

Compatibility wise this algorithm is usually trouble free, although you do occassionally run into implementations that do not extend the encoding of the $(Gamma, Phi)$ tuple correctly.

The following example shows the use of ElGamal for doing OAEP based on a SHA-256 digest. The same OAEP parameters as we used with RSA earlier. Other than primary algorithm names (DH for the key pair generation, ElGamal at the start of the cipher names), you can see it is virtually identical to the code we saw for RSA.

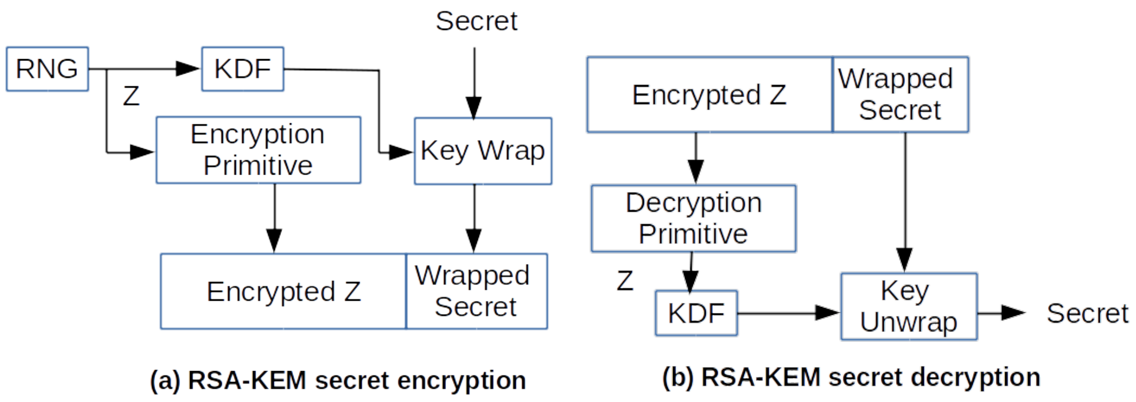
```java
/**
 * Simple example showing secret key wrapping and unwrapping based on
 * ElGamal OAEP.
 */
public class OAEPExampleWithElGamal
{
    /**
     * Generate a 2048 bit DH key pair using provider based parameters.
     *
     * @return a DH KeyPair
     */
    public static KeyPair generateDHKeyPair()
        throws GeneralSecurityException
    {
        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("DH", "BC");

        keyPair.initialize(2048);

        return keyPair.generateKeyPair();
    }

    /**
     * Generate a wrapped key using the OAEP algorithm,
     * returning the resulting encryption.
     *
     * @param dhPublic the public key to base the wrapping on.
     * @param secretKey the secret key to be encrypted/wrapped.
     * @return true if the signature verifies, false otherwise.
     */
    public static byte[] keyWrapOAEP(
        PublicKey dhPublic, SecretKey secretKey)
        throws GeneralSecurityException
    {
        Cipher cipher = Cipher.getInstance(
            "ElGamal/NONE/OAEPwithSHA256andMGF1Padding", "BC");

        cipher.init(Cipher.WRAP_MODE, dhPublic);

        return cipher.wrap(secretKey);
    }

    /**
     * Return the secret key that was encrypted in wrappedKey.
     *
     * @param dhPrivate the private key to use for the unwrap.
     * @param wrappedKey the encrypted secret key.
     * @param keyAlgorithm the algorithm that the encrypted key is for.
     * @return the unwrapped SecretKey.
     */
    public static SecretKey keyUnwrapOAEP(
        PrivateKey dhPrivate, byte[] wrappedKey, String keyAlgorithm)
        throws GeneralSecurityException
    {
        Cipher cipher = Cipher.getInstance(
            "ElGamal/NONE/OAEPwithSHA256andMGF1Padding", "BC");

        cipher.init(Cipher.UNWRAP_MODE, dhPrivate);

        return (SecretKey)cipher.unwrap(
                              wrappedKey, keyAlgorithm, Cipher.SECRET_KEY);
    }

    public static void main(String[] args)
        throws GeneralSecurityException
    {
        SecretKey aesKey = createTestAESKey();

        KeyPair kp = generateDHKeyPair();

        byte[] wrappedKey = keyWrapOAEP(kp.getPublic(), aesKey);

        SecretKey recoveredKey = keyUnwrapOAEP(
                                    kp.getPrivate(),
                                    wrappedKey, aesKey.getAlgorithm());

        System.out.println(
            Arrays.areEqual(aesKey.getEncoded(), recoveredKey.getEncoded()));
    }
}
```

Note that the code uses a 2048 bit key. If you try dumping the ciphertext in this case, you will find that the cipher text is twice the length it was when you used RSA.

Assuming all goes well, running the example should produce the following output showing that the original key bytes have been correctly recovered:

```
true
```

RSA-KEM



(a) RSA-KEM secret encryption

(b) RSA-KEM secret decryption

Schematics of RSA-KEM for encryption and decryption - note how it differs from OAEP

The RSA-KEM key transport algorithm is described in NIST SP 800-56B [10] and also in RFC 5990 [82], which details its use in Cryptographic Message Syntax (CMS). It is the closest you can get to an ideal use of the RSA algorithm at its highest level of security which is shown by the fact RSA-KEM has a tighter security proof than RSA-OAEP. The only downside of the algorithm is it produces a slightly longer encryption than the corresponding RSA-OAEP encryption would be. The reason for this is the payload sent in the RSA block making up the RSA-KEM message is used to create a symmetric key which is then used to wrap the key being transported.

At the moment, RSA-KEM is only implemented in the BCFIPS API.

RSA-KEM Key Wrapping

Key wrapping is a two stage process. We encrypt a random value used to derive a key encryption key (KEK) and then wrap the secret key we want to protect with the KEK. Given our modulus N, our public exponent E, and a length $kekLen$ for the KEK; we perform the following steps:

1. Generate a random Z such that $0 < Z < N - 1$
2. Calculate $C = Z^E \mod N$
3. Derive $KEK = KDF(Z, kekLen)$
4. Calculate $WK = Wrap(KEK, K)$
5. Set $EK = C \parallel WK$.
6. Return EK.

The function $KDF()$ is usually a key derivation function based on a message digest. The function is usually based around one of the functions we looked at in Chapter 3. The $Wrap()$ function is one of the symmetric key wrapping functions we looked at in Chapter 4.

At the basic level, in Bouncy Castle RSA-KEM is supported using the JCE's `Cipher` class and a simple example of use is as follows:

```
/**
 * Generate a wrapped key using the RSA-KTS-KEM-KWS algorithm,
 * returning the resulting encryption.
 *
 * @param rsaPublic the public key to base the wrapping on.
 * @param ktsSpec key transport parameters.
 * @param secretKey the secret key to be encrypted/wrapped.
 * @return true if the signature verifies, false otherwise.
 */
public static byte[] keyWrapKEMS(
    PublicKey rsaPublic, KTSParameterSpec ktsSpec, SecretKey secretKey)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");

    cipher.init(Cipher.WRAP_MODE, rsaPublic, ktsSpec);

    return cipher.wrap(secretKey);
}
```

You can see from the arguments to the `keyWrapKEMS()` method that there are two things to consider when wrapping a secret key using RSA-KEM. One is the overall security of the RSA key used, the other is the overall security of the parameters chosen, with both the choice of symmetric algorithm and the KDF function chosen also contributing to the security of the key encryption mechanism. We will have a closer look at the parameters a bit further on.

RSA-KEM Key Unwrapping

As with RSA-KEM key wrapping, RSA-KEM key unwrapping is also a two stage process. We need to first decrypt the key material for the key encryption key, and then decrypt the wrapped key to recover the original secret key we were sending.

Given our the RSA-KEM output WK, the modulus N and our private exponent D we perform the following steps:

1. Split $EK = C \parallel WK$ where C is the RSA encrypted block, WK is the wrapped key.
2. Calculate $Z = C^D \mod N$
3. Derive $KEK = KDF(Z, kekLen)$
4. Calculate $K = Unwrap(KEK, WK)$
5. Return K.

At the JCE level, this again translates into a use of the `Cipher` class:

```java
/**
 * Return the secret key that is encrypted in wrappedKey.
 *
 * @param rsaPrivate the private key to use for the unwrap.
 * @param ktsSpec key transport parameters.
 * @param wrappedKey the encrypted secret key.
 * @param keyAlgorithm the algorithm that the encrypted key is for.
 * @return the unwrapped SecretKey.
 */
public static SecretKey keyUnwrapKEMS(
    PrivateKey rsaPrivate, KTSParameterSpec ktsSpec,
    byte[] wrappedKey, String keyAlgorithm)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");

    cipher.init(Cipher.UNWRAP_MODE, rsaPrivate, ktsSpec);

    return (SecretKey)cipher.unwrap(
                              wrappedKey, keyAlgorithm, Cipher.SECRET_KEY);
}
```

Note: as the code implies, for RSA-KEM to work both sides must have agreed to the same parameters.

Putting Everything Together

The KTSParameterSpec offers the constants `KDF2`, which refers to the X9.63 KDF, and `KDF3`, which refers to the algorithm formerly referred to as the NIST Concatenation KDF.

The following example uses the RSA-KEM method with the default KDF, which is based on SHA-256 using the NIST concatenation KDF. In this case the parameters also specify that the KEK is to be 256 bits and used with AESKWP as the key wrapping function.

```java
/**
 * Simple example showing secret key wrapping and unwrapping based on RSA-KEMS.
 */
public class KEMSExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        SecretKey aesKey = new SecretKeySpec(
                                Hex.decode("000102030405060708090a0b0c0d0e0f"),
                                "AES");

        KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA", "BCFIPS");

        keyPair.initialize(2048);

        KeyPair kp = keyPair.generateKeyPair();

        KTSParameterSpec ktsSpec =
                            new KTSParameterSpec.Builder(
                                "AESKWP", 256,
                                Strings.toByteArray("OtherInfo Data")).build();

        byte[] wrappedKey = keyWrapKEMS(kp.getPublic(), ktsSpec, aesKey);

        SecretKey recoveredKey = keyUnwrapKEMS(
                                    kp.getPrivate(), ktsSpec,
                                    wrappedKey, aesKey.getAlgorithm());

        System.out.println(
            Arrays.areEqual(aesKey.getEncoded(), recoveredKey.getEncoded()));
    }
}
```

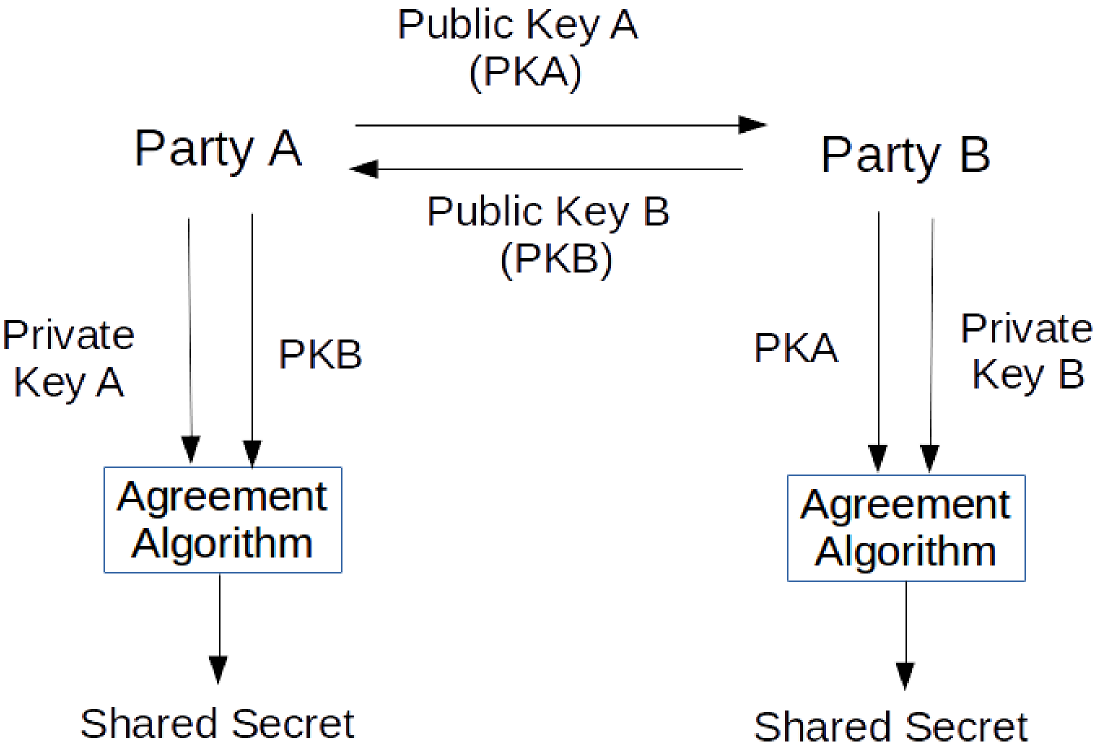
As we saw in Chapter 3, the OtherInfo data provides additional material to the KDF function for producing the KEK. We will also see later that OtherInfo can be used to introduce the output of other algorithms into the mix as well. Bouncy Castle also provides a helper class for constructing OtherInfo from an ASN.1 encoding - the `DEROtherInfo` class in the `org.bouncycastle.crypto.util` package.

Running the example should produce the following output indicating that the bytes making up `aesKey` have successfully made it through the wrapping and unwrapping process:

```
true
```

Ordinarily, with RSA-OAEP for example, you would expect `wrappedKey` to be 256 bytes long. If you check the length of `wrappedKey` produced by the example code, you will find the byte array is 280 bytes long. In this case, the additional 24 bytes of data represent the AESKWP wrapping of the secret key involved.

Key Agreement and Key Exchange



The operations for basic key agreement between two parties

The algorithms we are about to look at now have their foundation in Diffie-Hellman-Merkle[13] key exchange. More commonly known as Diffie-Hellman, based on the original paper by Whitfield Diffie and Martin Hellman in 1976, it is regarded as one of the first examples of a practical public key cryptography scheme. Funnily enough, as with RSA, it has also turned out that the scheme was developed independently in the United Kingdom at GCHQ by James Ellis, Clifford Cocks, and Malcolm Williamson in 1969. The primary NIST standard for key agreement is NIST SP 800-56A "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography" [9], however many of the EC signature schemes we have looked at such as those from Russia, China,

[13] Ralph Merkle was the original inventor of public key cryptography, the addition of his name to the algorithm name in 2002, at the request of the other authors, was in recognition of this.

and the Ukraine, also have associated key agreement algorithms. Diffie-Hellman's first appearance in an IETF RFC was probably in 1999, in RFC 2631 [46] which was related to work going on in the development of what became X9.42 [34]. There have been a number of updates and alternate proposals both around prime fields and elliptic curve since then.

Diffie-Hellman over Finite Fields

Diffie-Hellman is based around a set of parameters (P, Q, G) where the parameters have the following characteristics.

- P: a prime.
- Q: a prime divisor of $(P - 1)$.
- G: a generator of a subgroup of order Q in the multiplicative group of Galois Field $GF(P)$, such that $1 < G < P$.

Often the Q parameter is ignored and there is an example of this in PKCS #3 [120]. This is unfortunate as the Q value provides a way of better validating a public key than the basic check detailed in Section 5.1 of RFC 7919 [98] and it is also required if you wish to implement MQV as described in NIST SP 800-56A [9].

The key agreement operation is quite simple. Considering the case for only two parties, the protocol proceeds as follows:

1. A chooses a random secret x where $1 \leq x \leq P - 2$
2. B chooses a random secret y where $1 \leq y \leq P - 2$
3. A sends B G^x, B sends A G^y
4. A computes $Z = (G^y)^x$, if $Z \leq 1$ or $Z = (P - 1)$ then return error
5. B computes $Z = (G^x)^y$, if $Z \leq 1$ or $Z = (P - 1)$ then return error
6. A returns Z
7. B returns Z

The shared value Z can then be used in some fashion to facilitate encrypted communications between A and B. It is also useful to know that the random secret for either party is referred to as the private key X and that the public value G^X is referred to as the public key Y. We will see this used in the discussions further on.

One thing which is important to recognise about this protocol is it need not be restricted to two parties. A consequence of this is when the protocol is executed between remote parties, it should be clear that it is possible to do a man-in-the-middle attack if neither party takes any steps to verify the origins of the other party's public key.

If you use key agreement, it is important to use it together with an authentication mechanism in order to prevent a man-in-the-middle attack.

A number of recommended parameter sets for the use with Diffie-Hellman have been described in the IETF standards such as in RFC 5114 [67]. You can also generate your own parameter sets in the JCE, using the `AlgorithmParameterGenerator`. As with DSA, the process of generating Diffie-Hellman parameters is slow. It is not something you want to do very often. The following method shows an example of how to generate parameters for the purpose of creating 2048 bit keys:

```
/**
 * Return a generated set of DH parameters suitable for creating 2048
 * bit keys.
 *
 * @return a DHParameterSpec holding the generated parameters.
 */
public static DHParameterSpec generateDHParams()
    throws GeneralSecurityException
{
    AlgorithmParameterGenerator paramGen =
        AlgorithmParameterGenerator.getInstance("DH", "BC");

    paramGen.init(2048);

    AlgorithmParameters params = paramGen.generateParameters();

    return params.getParameterSpec(DHParameterSpec.class);
}
```

The example method returns a `DHParameterSpec` which is a JCE class that supports the Diffie-Hellman parameters as described in PKCS #3. As this is the case, the `DHParameterSpec` only supports P, G, and an additional value L. The L value is used to specify the bit length of the private value in the Diffie-Hellman operation. The purpose of L is to make the calculations more efficient - for example with a 2048 bit private value the calculation (if only described by the private value) is believed to have a security rating of 1024 bits, the square-root of 2^{2048}. Given that a 2048 bit Diffie-Hellman key is currently thought to offer 112 bits of security the private value can be as small as 224 bits. This is discussed further in RFC 5114, Section 4 [67] and also RFC 4419 [61], Section 6.2 which suggests a rule of the private value being at least twice the key size of the symmetric key to be generated from the agreement. Left to nothing but chance, the longest possible private value may be as much as 2047 bits, clearly overkill in this case.

The JCE also supports a `DHGenParameterSpec` class which takes both the size of the base prime and a proposed size for the private value. For example to generate our 2048 bit parameters and additionally specify a private value size of 256 bits, you could change the initialization of `paramGen` to the following:

```
    paramGen.init(new DHGenParameterSpec(2048, 256));
```

The Bouncy Castle APIs also support the `DHDomainParameterSpec`. This can be used to retrieve an extension of DHParameterSpec that also includes the Q value as well as validation parameters (in the case of BCFIPS).

Key Pair Generation

Key pair generation involves the creation of a public value (called Y) and a private value (called X). The private value X is randomly generated where $1 \leq X \leq P-2$ and Y is calculated as $Y = G^X \mod P$. The private X value is then used to initialise a key agreement operation and the Y is sent to the other parties we are trying to calculate a shared key with.

As with other algorithms, in Java the most basic way of generating a key pair is to create a KeyPairGenerator and simply specify the size as in:

```java
/**
 * Generate a 2048 bit DH key pair using provider based parameters.
 *
 * @return a DH KeyPair
 */
public static KeyPair generateDHKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DH", "BC");

    keyPairGen.initialize(2048);

    return keyPairGen.generateKeyPair();
}
```

If the key size one that does not already have parameters provided for in the Bouncy Castle provider, the key size will result in the KeyPairGenerator generating suitable parameters for the requested key size from scratch - this may take some time. You can also specify your own parameters for generating a key if you wish using the DHParameterSpec by making use of the KeyPairGenerator.initialize() method that takes an AlgorithmParameterSpec to pass a DHParameterSpec you created earlier.

```java
/**
 * Generate a DH key pair using our own specified parameters.
 *
 * @param dhSpec the DH parameters to use for key generation.
 * @return a DH KeyPair
 */
public static KeyPair generateDHKeyPair(DHParameterSpec dhSpec)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DH", "BC");

    keyPairGen.initialize(dhSpec);

    return keyPairGen.generateKeyPair();
}
```

In both cases, the KeyPair returned by the example method will consist of implementations of DHPublicKey and DHPrivateKey from the javax.crypto.interfaces package.

Basic Key Agreement

The Java KeyAgreement class divides the protocol up into three classes of methods:

- init() which takes the caller's private key and, optionally, any other parameters.
- doPhase() which takes the another party's public key and a boolean lastPhase - when lastPhase is true it means the protocol has finished.
- generateSecret() which is called after the protocol has finished. These methods return the shared secret in a couple of different forms.

In the most general case the approach to using KeyAgreement is to call init() with your private key and parameters (if you have them), call doPhase() with each party's public key passing false for lastPhase until the last public key is used, and, finally, call generateSecret() to retrieve the shared secret, either as a byte array, or as a secret key.

The most basic way to retrieve the shared secret is as a byte array. The following method demonstrates Diffie-Hellman between two parties with the shared secret being returned as a byte array. For the sake of example, the method given also takes the left most 32 bytes of shared secret returned by the instance of KeyAgreement and uses them as the return value.

```
    /**
     * Generate an agreed secret byte value of 32 bytes in length.
     *
     * @param aPriv Party A's private key.
     * @param bPub Party B's public key.
     * @return the first 32 bytes of the generated secret.
     */
    public static byte[] generateSecret(PrivateKey aPriv, PublicKey bPub)
        throws GeneralSecurityException
    {
        KeyAgreement agreement = KeyAgreement.getInstance("DH", "BC");

        agreement.init(aPriv);

        agreement.doPhase(bPub, true);

        return Arrays.copyOfRange(agreement.generateSecret(), 0, 32);
    }
```

If you remove the call to `Arrays.copyOfRange()` you will find the return value is a lot bigger than 256 bits.

The second way to return a shared secret is as a key.

```
/**
 * Generate an agreed AES key value of 256 bits in length.
 *
 * @param aPriv Party A's private key.
 * @param bPub Party B's public key.
 * @return the generated AES key (256 bits).
 */
public static SecretKey generateAESKey(PrivateKey aPriv, PublicKey bPub)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DH", "BC");

    agreement.init(aPriv);

    agreement.doPhase(bPub, true);

    return agreement.generateSecret("AES");
}
```

This example method is still just using the straight Diffie-Hellman calculation. We will see what this means when we look at the output from the example below.

An Initial Example

Before we look at an example, in order to assist with demonstrating a couple of things about the use of Diffie-Hellman we need to define some utility methods which will allow us have some control over the values of the public and private keys that are being produced by the JCE.

If you have a look at the following class you will see that it provides a hardcoded set of private values via a Bouncy Castle `FixedSecureRandom` class which it combines with the default Diffie-Hellman parameters from the provider to produce keys with a 256 bit private value. The two arguments to the `FixedSecureRandom` mean that the `insecureRandom` field can be used to produce up to two keys.

```java
public class InsecureDHUtils
{
    // Set up constant secure random to produce constant private keys with
    private static SecureRandom insecureRandom = new FixedSecureRandom(
        new FixedSecureRandom.Source[] {
        new FixedSecureRandom.BigInteger(256,
            Hex.decode(
                "914bb9b3d677c4ce87e5f671a88c85c0" +
                    "615228c6f7d6fbfdee89092f69609128")),
        new FixedSecureRandom.BigInteger(256,
            Hex.decode(
                "837028bb0ccdaf1cf9a390b1f84bf916" +
                    "999ba4760d4297124ca991a1e616b676")) });


    private static DHParameterSpec dhSpec;

    /**
     * Initialize our Diffie-Hellman parameters for key pair generation
     */
    static void insecureUtilsInit()
        throws GeneralSecurityException
    {
        // Grab a regular parameter set and tailor it for keys with a 256 bit
        // private value.
        KeyPair kp = generateDHKeyPair();
        dhSpec = ((DHPublicKey)kp.getPublic()).getParams();
        dhSpec = new DHParameterSpec(dhSpec.getP(), dhSpec.getG(), 256);
    }

    /**
     * Generate a key pair using our constant random source.
     * @return a Diffie-Hellman key pair.
     */
    static KeyPair insecureGenerateDHKeyPair()
        throws GeneralSecurityException
    {
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DH", "BC");

        keyPairGen.initialize(dhSpec, insecureRandom);

        return keyPairGen.generateKeyPair();
    }
}
```

Combined with the `generateSecret()` and `generateAESKey()` methods we looked at earlier, the utility methods can be used to produce an example of the use of Diffie-Hellman between two parties using key agreement to generate both a shared secret and an AES key as follows:

```
/**
 * Basic Diffie-Hellman example showing the use of the KeyAgreement class
 * for generating byte arrays and secret keys.
 * <b>Note</b>: this example is generating constant keys, do not use the
 * key generation code in real life unless that is what you intend!
 */
public class BasicDHExample
{
    // Generate and print a shared secret using the A and B party key pairs.
    private static byte[] generateSecretValue(KeyPair aKp, KeyPair bKp)
        throws GeneralSecurityException
    {
        byte[] aValue = generateSecret(aKp.getPrivate(), bKp.getPublic());
        byte[] bValue = generateSecret(bKp.getPrivate(), aKp.getPublic());

        System.out.println("aS: " + Hex.toHexString(aValue));
        System.out.println("bS: " + Hex.toHexString(bValue));

        return aValue;
    }

    // Generate and print an AES key using the A and B party key pairs.
    private static SecretKey generateAESKeyValue(KeyPair aKp, KeyPair bKp)
        throws GeneralSecurityException
    {
        SecretKey aKey = generateAESKey(aKp.getPrivate(), bKp.getPublic());
        SecretKey bKey = generateAESKey(bKp.getPrivate(), aKp.getPublic());

        System.out.println("aK: " + Hex.toHexString(aKey.getEncoded()));
        System.out.println("bK: " + Hex.toHexString(bKey.getEncoded()));

        return aKey;
    }

    public static void main(String[] args)
        throws Exception
    {
        // Set up parameter spec for key pair generation
        insecureUtilsInit();

        // Generate the constant key pairs for party A and party B
        KeyPair aKp = insecureGenerateDHKeyPair();
        KeyPair bKp = insecureGenerateDHKeyPair();

        // key agreement generating a shared secret
        byte[] retGenSec = generateSecretValue(aKp, bKp);
        // key agreement generating an AES key
        SecretKey retAESKey = generateAESKeyValue(aKp, bKp);

        // compare the two return values.
        System.err.println(Arrays.areEqual(retGenSec, retAESKey.getEncoded()));
    }
}
```

When you run this example it will print out 5 lines and, if all has gone well, they will read

```
aS: 8412f73f3e704cdf8af44d2a229933a70f020c1bbc0b5d378ee25a882003d18b
bS: 8412f73f3e704cdf8af44d2a229933a70f020c1bbc0b5d378ee25a882003d18b
aK: 8412f73f3e704cdf8af44d2a229933a70f020c1bbc0b5d378ee25a882003d18b
bK: 8412f73f3e704cdf8af44d2a229933a70f020c1bbc0b5d378ee25a882003d18b
true
```

There are two things worth noting about this output: all four values are the same in spite of the fact that distinct instances of `KeyAgreement` have been used and the value returned by the `generateSecret()` method, which simply returns the results of the public/private key calculations as a byte array, has the same first 32 bytes as the AES key encoding. This last one is a bit troubling as clearly the key is revealing the answer to the Diffie-Hellman Agreement Calculation we saw earlier, a calculation which you clearly only get to do once in these circumstances. This is a bit of an issue if you want to be able to issue a certificate for your Diffie-Hellman public key and not throw it away after a single use.

There are a couple of options available to help allow multiple use of a Diffie-Hellman key pair. The first is to make use of additional keying material and a KDF to generate the key. The second is to make use of Ephemeral Keys as well. We will look at introducing additional keying material and a KDF first.

Adding Keying Material and a KDF

Introducing a KDF into the process allows us to combine the shared secret with some additional agreed on material and generate a new shared secret that has a value that should not be reversible to the original shared secret. This will hide the values related to the actual key agreement, and if the additional key material is chosen appropriately allow for the re-use of the same Diffie-Hellman keys to produce a series of unique shared keys.

The KDF used is specified by the algorithm name passed to `KeyAgreement.getInstance()`. There does not appear to be a standard format for these so here at Bouncy Castle we have adopted the convention used with the other compound names and use names of the format:

- DHwith<digest>KDF for uses of the X9.63 KDF
- DHwith<digest>CKDF for uses of the NIST concatenation KDF

For example, "DHwithSHA256KDF" specifies Diffie-Hellman using the X9.63 KDF based on the SHA-256 message digest.

The key material, as it will generally vary per invocation, is managed by using a Bouncy Castle specific `AlgorithmParameterSpec` - the class `org.bouncycastle.jcajce.spec.UserKeyingMaterialSpec`. The name of the class comes from the name the parameter is referred to in RFC 5652 [74] for the key agreement recipient in Cryptographic Message Syntax (CMS).

The `UserKeyingMaterialSpec` just takes a byte array for contents.

```
/**
 * Generate an agreed AES key value of 256 bits in length.
 *
 * @param aPriv Party A's private key.
 * @param bPub Party B's public key.
 * @return the generated AES key (256 bits).
 */
public static SecretKey generateAESKey(
    PrivateKey aPriv, PublicKey bPub, byte[] keyMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DHwithSHA256KDF", "BC");

    agreement.init(aPriv, new UserKeyingMaterialSpec(keyMaterial));

    agreement.doPhase(bPub, true);

    return agreement.generateSecret("AES");
}
```

How you format the byte array used to construct the `UserKeyingMaterialSpec` is up to you. For example, both NIST SP 800-56A [9] and NIST SP 800-56B [10] provide detailed recommendations on how to construct user keying material and you can also make use of the `DEROtherInfo` class included in Bouncy Castle.

The next example makes use of our new `generateAESKey()` method to incorporate additional key material using a KDF. For comparison purposes, it still uses the same method for generating a byte array from a shared secret as before.

```java
/**
 * Basic Diffie-Hellman example showing the use of the KeyAgreement class
 * for generating byte arrays and secret keys using additional key material.
 * <b>Note</b>: this example is generating constant keys, do not use the
 * key generation code in real life unless that is what you intend!
 */
public class UKMDHExample
{
    // Generate and print a shared secret using the A and B party key pairs.
    private static byte[] generateSecretValue(KeyPair aKp, KeyPair bKp)
        throws GeneralSecurityException
    {
        byte[] aValue = generateSecret(aKp.getPrivate(), bKp.getPublic());
        byte[] bValue = generateSecret(bKp.getPrivate(), aKp.getPublic());

        System.out.println("aS: " + Hex.toHexString(aValue));
        System.out.println("bS: " + Hex.toHexString(bValue));

        return aValue;
    }

    // Generate and print an AES key using the A and B party key pairs.
    // Note: this time around we have introduced some bytes of key material.
    private static SecretKey generateAESKeyValue(KeyPair aKp, KeyPair bKp)
        throws GeneralSecurityException
    {
        byte[] keyMaterial = Strings.toByteArray("For an AES key");

        SecretKey aKey = generateAESKey(
            aKp.getPrivate(), bKp.getPublic(), keyMaterial);
        SecretKey bKey = generateAESKey(
            bKp.getPrivate(), aKp.getPublic(), keyMaterial);

        System.out.println("aK: " + Hex.toHexString(aKey.getEncoded()));
        System.out.println("bK: " + Hex.toHexString(bKey.getEncoded()));

        return aKey;
    }

    public static void main(String[] args)
        throws Exception
    {
        // Set up parameter spec for key pair generation
        insecureUtilsInit();

        // Generate the key pairs for party A and party B
        KeyPair aKp = insecureGenerateDHKeyPair();
        KeyPair bKp = insecureGenerateDHKeyPair();

        // key agreement generating a shared secret
        byte[] retGenSec = generateSecretValue(aKp, bKp);
        // key agreement generating an AES key
        SecretKey retAESKey = generateAESKeyValue(aKp, bKp);

        // compare the two return values.
        System.err.println(Arrays.areEqual(retGenSec, retAESKey.getEncoded()));
    }
}
```

Try running the example and you should see it produce the following output:

```
aS: 8412f73f3e704cdf8af44d2a229933a70f020c1bbc0b5d378ee25a882003d18b
bS: 8412f73f3e704cdf8af44d2a229933a70f020c1bbc0b5d378ee25a882003d18b
aK: 1898ad4d20a7c33c54f6ab58b9b9d7f363f94fdbf48b5e72db487a80c84f2abc
bK: 1898ad4d20a7c33c54f6ab58b9b9d7f363f94fdbf48b5e72db487a80c84f2abc
false
```

As you can see, the shared value for the AES key has now changed. If you tweak the value passed to the `UserKeyingMaterialSpec` further you will see the value of the AES key changes again - despite the constant keys.

Adding Ephemeral Keys

The next thing that can be done to improve the key agreement process is to make use of ephemeral keys. So far we have been considering the case of multiple use keys. These are also referred to as static keys, due to their long term existence. Ephemeral keys, on the other hand, are only ever used once. The best thing about using ephemeral keys, if generated properly, is that they introduce forward secrecy into the system as a compromise of the static key by itself will not result in the ability to decrypt any message that has been sent. Likewise compromising an ephemeral key will not result in the compromise of its associated message unless the static key is known as well. Even then you will only get that one message, as recovering any other message will also involve compromising the associated ephemeral key used with it.

There are two ways of combining ephemeral keys into a Diffie-Hellman based protocol. One of these is referred to as the Unified Model and exists in two pass and one-pass versions. The second one is referred to as MQV, after the algorithm's creators: Alfred Menezes, Minghua Qu, and Scott Vanstone.

MQV also exists in two pass and one-pass versions. In the case of a one-pass scheme only one party in the protocol has an ephemeral key pair, the other party uses their static key where they would have used an ephemeral key before.

The Unified Model

The unified model introduces a second Diffie-Hellman operation which makes use of ephemeral keys. Seen mostly from the point of view of party A, the scheme, in two pass mode, requires the following operations:

1. A generates ephemeral key pair (Y_e, X_e)
2. A sends B (Y_s, Y_e)
3. B sends A Y_{sb} and Y_{eb}
4. A computes $Z_s = (Y_{sb})^{X_s}$, if $Z \leq 1$ or $Z = (P - 1)$ then return error
5. A computes $Z_e = (Y_{eb})^{X_e}$, if $Z \leq 1$ or $Z = (P - 1)$ then return error
6. return $Z = Z_e \parallel Z_s$

Where Y represents a public key, X the private key, with a subscript starting with s indicating a static key and e indicating an ephemeral key.

A one-pass version of this protocol is also possible if either A or B replace their ephemeral key pair with their static key pair. You can see that if no authentication mechanism is used in conjunction with the scheme it is still possible for a man-in-the-middle attack to take place, in both, or either, steps involving the agreement step.

Using the KeyAgreement Class

The next thing to look at is how to make use of the Unified Diffie-Hellman model using the `KeyAgreement` class. For the purposes of this protocol the ephemeral keys are treated as parameters to the agreement. In Bouncy Castle these extra keys and the keying material are configured using the `DHUParameterSpec` which can be found in the `org.bouncycastle.jcajce.spec` package.

The Java method below shows what is required for generating a basic AES key using the Unified Diffie-Hellman model and an X9.63 KDF based on SHA-256.

```
    /**
     * Generate an agreed AES key value of 256 bits in length
     * using the Unified Diffie-Hellman model.
     *
     * @param aPriv Party A's private key.
     * @param aPubEph Party A's ephemeral public key.
     * @param aPrivEph Party A's ephemeral private key.
     * @param bPub Party B's public key.
     * @param bPubEph Party B's ephemeral public key.
     * @return the generated AES key (256 bits).
     */
    public static SecretKey dhuGenerateAESKey(
        PrivateKey aPriv, PublicKey aPubEph, PrivateKey aPrivEph,
        PublicKey bPub, PublicKey bPubEph, byte[] keyMaterial)
        throws GeneralSecurityException
    {
        KeyAgreement agreement =
            KeyAgreement.getInstance("DHUwithSHA256KDF", "BC");

        agreement.init(aPriv,
            new DHUParameterSpec(aPubEph, aPrivEph, bPubEph, keyMaterial));

        agreement.doPhase(bPub, true);

        return agreement.generateSecret("AES");
    }
```

As you can see, the Unified model is specified by starting the algorithm name with “DHU” rather than just “DH”.

The following example shows the use of the `dhuGenerateAESKey()` method to generate a shared AES key between 2 parties.

```
/**
 * Basic Unified Diffie-Hellman example showing use of two key pairs
 * per party in the protocol, with one set being regarded as ephemeral.
 */
public class UnifiedDHExample
{
    public static void main(String[] args)
        throws Exception
    {
        // Generate the key pairs for party A and party B
        KeyPair aKpS = generateDHKeyPair();
        KeyPair aKpE = generateDHKeyPair();    // A's ephemeral pair
        KeyPair bKpS = generateDHKeyPair();
        KeyPair bKpE = generateDHKeyPair();    // B's ephemeral pair

        // key agreement generating an AES key
        byte[] keyMaterial = Strings.toByteArray("For an AES key");

        SecretKey aKey = dhuGenerateAESKey(
            aKpS.getPrivate(),
            aKpE.getPublic(), aKpE.getPrivate(),
            bKpS.getPublic(), bKpE.getPublic(), keyMaterial);
        SecretKey bKey = dhuGenerateAESKey(
            bKpS.getPrivate(),
            bKpE.getPublic(), bKpE.getPrivate(),
            aKpS.getPublic(), aKpE.getPublic(), keyMaterial);

        // compare the two return values.
        System.out.println(
            Arrays.areEqual(aKey.getEncoded(), bKey.getEncoded()));
    }
}
```

In this case, as it is using random keys, our best sign that it is working is the key comparison at the end. Assuming all goes well, the example will produce the following indicating that both keys were calculated to have the same value:

```
true
```

You can see, by looking at the example, that the static key pairs of either party could also be used in the protocol to make up for the lack of one of the ephemeral key pairs. This will also mean that the static keys acting as a replacement would also be passed in to be used as configuration parameters to the `dhuGenerateAESKey()` method as well. This is the one-pass protocol that we mentioned earlier. The key feature of it, as the name suggests, is that the second party in the protocol only needs to have published their public key for the initiating party to still be able to introduce an ephemeral key into the agreement process. This makes it possible to send an offline message which can be handled by the second party at a later time without the need for any interaction.

MQV

MQV is another key agreement protocol which also makes use of ephemeral key pairs. Named from the surnames of its originators Alfred Menezes, Minghua Qu, and Scott Vanstone, the protocol was first described in the paper "Some New Key Agreement Protocols Providing Mutual Implicit Authentication" [125] in 1995. MQV differs from the Unified Model in that it also incorporates an "implicit signature" based on the static private keys of both parties. This means that any attempt to man-in-the-middle the protocol will fail as it will be detected - the shared secrets will not agree in value. This "implicit signature" means MQV offers a stronger basis for authentication "out of the box" than the Unified model does.

The MQV protocol was originally written for Elliptic Curve Cryptography but it can be applied to Finite Field Cryptography, and details for doing so are provided in NIST SP 800-56A [9], amongst others. Assuming validation of the inputs, the following steps, from the point of view of party A, are required to excute MQV:

1. A generates ephemeral key pair (Y_e, X_e)
2. B generates ephemeral key pair (Y_{eb}, X_{eb})
3. B sends A Y_{sb} and Y_{eb}
4. Set $w = \lceil bitLength(Q)/2 \rceil$
5. Set $T_a = (Y_e \mod 2^w) + 2^w$
6. Set $S_a = (X_e + T_a X_s) \mod Q$
7. Set $T_b = (Y_{eb} \mod 2^w) + 2^w$
8. Set $Z = ((Y_{eb}(Y_{sb}^{T_b}))^{S_a}) \mod P$
9. if $Z \leq 1$ or $Z = (P - 1)$ then return error
10. return Z

As you might guess from the algorithm description, one catch with using MQV in Java is that the Q value for the domain parameters needs to be available. In Bouncy Castle this means making use of parameters defined using a `DHDomainParametersSpec` class.

Using the KeyAgreement Class

From the Java point of view MQV is performed in the same way as the Unified Model, but also lacks a JCE parameter specification class. In Bouncy Castle the extra keys and the keying material are configured using the `MQVParameterSpec` which can be found in the `org.bouncycastle.jcajce.spec` package.

The `MQVParameterSpec` is used in the same way as the `DHUParameterSpec`. The naming convention for MQV algorithms is also similar to that used with regular Diffie-Hellman, you just start the algorithm name with "MQV" instead of "DH". The following example method shows the code required for generating a basic AES key using the MQV protocol and an X9.63 KDF based on SHA-256.

```
  /**
   * Generate an agreed AES key value of 256 bits in length
   * using MQV.
   *
   * @param aPriv Party A's private key.
   * @param aPubEph Party A's ephemeral public key.
   * @param aPrivEph Party A's ephemeral private key.
   * @param bPub Party B's public key.
   * @param bPubEph Party B's ephemeral public key.
   * @return the generated AES key (256 bits).
   */
  public static SecretKey mqvGenerateAESKey(
      PrivateKey aPriv, PublicKey aPubEph, PrivateKey aPrivEph,
      PublicKey bPub, PublicKey bPubEph, byte[] keyMaterial)
      throws GeneralSecurityException
  {
      KeyAgreement agreement =
          KeyAgreement.getInstance("MQVwithSHA256KDF", "BC");

      agreement.init(aPriv,
          new MQVParameterSpec(aPubEph, aPrivEph, bPubEph, keyMaterial));

      agreement.doPhase(bPub, true);

      return agreement.generateSecret("AES");
  }
```

Putting the example into a simple use case gives us the following code:

```
/**
 * Basic Diffie-Hellman MQV example showing use of two key pairs
 * per party in the protocol, with one set being regarded as ephemeral.
 */
public class MQVDHExample
{
    public static void main(String[] args)
        throws Exception
    {
        // Generate the key pairs for party A and party B
        KeyPair aKpS = generateDHKeyPair();
        KeyPair aKpE = generateDHKeyPair();    // A's ephemeral pair
        KeyPair bKpS = generateDHKeyPair();
        KeyPair bKpE = generateDHKeyPair();    // B's ephemeral pair

        // key agreement generating an AES key
        byte[] keyMaterial = Strings.toByteArray("For an AES key");

        SecretKey aKey = mqvGenerateAESKey(
            aKpS.getPrivate(),
            aKpE.getPublic(), aKpE.getPrivate(),
            bKpS.getPublic(), bKpE.getPublic(), keyMaterial);
        SecretKey bKey = mqvGenerateAESKey(
            bKpS.getPrivate(),
            bKpE.getPublic(), bKpE.getPrivate(),
            aKpS.getPublic(), aKpE.getPublic(), keyMaterial);

        // compare the two return values.
        System.out.println(
            Arrays.areEqual(aKey.getEncoded(), bKey.getEncoded()));
    }
}
```

Running the example should produce the following output indicating that both keys were calculated to have the same value:

```
true
```

Hopefully, you can also see how either one of the ephemeral keys in the example can be replaced with the same party's static key in order to produce a one-pass version of the MQV protocol.

Diffie-Hellman over Elliptic Curve

Most of what we said about Diffie-Hellman so far applies equally well to performing the protocol using Elliptic Curve Cryptography (ECC). Elliptic Curve (EC) offers the advantages of similar performance, but with much smaller key sizes for an equivalent level of security.

While there are now specific parameters sets for EC key agreement in some standards, in general EC Diffie-Hellman can be performed with keys generated using the same EC parameter sets as ECDSA. In example code in the following sections we have assumed you have access to the key pair generator methods defined for ECDSA in Chapter 6.

Basic Key Agreement

The basic steps for EC, assuming a co-factor of 1, key agreement goes as follows

1. A sends their public key W_a to B

2. B sends their public key W_b to A
3. A computes $(x, y) = d_a W_b$. If $((x, y)$ is at $\infty)$ return invalid
4. B computes $(x, y) = d_b W_a$. If $((x, y)$ is at $\infty)$ return invalid
5. A returns x
6. B returns x

The only catch in this otherwise straightforward procedure is the co-factor of the curve forming the EC parameters involved is not always 1. Where the co-factor (normally given as h) is not 1 and not taken into account, it is possible for someone to use a specially crafted public key which will result in information about the other party's private key being leaked[14]. In order to protect against this a party receiving a public key needs to make sure the other party's public key is always forced into the appropriate subgroup for the curve parameters we are using.

There are two approaches to doing this the co-factor. The first, favoured in the NIST standards, is to just multiply in the co-factor, so steps 4 and 5 are replaced with:

1. A computes $(x, y) = hd_a W_b$. If $((x, y)$ is at $\infty)$ return invalid
2. B computes $(x, y) = hd_b W_a$. If $((x, y)$ is at $\infty)$ return invalid

The second approach is to multiply through by the co-factor and then multiply through by its inverse. In this case steps 4 and 5 need to be expanded into the 5 steps below.

1. set $l = h^{-1} \mod N$
2. A computes $P_b = hW_b$
3. B computes $P_a = hW_a$
4. A computes $(x, y) = (d_a l \mod N)P_b$. If $((x, y)$ is at $\infty)$ return invalid
5. B computes $(x, y) = (d_b l \mod N)P_a$. If $((x, y)$ is at $\infty)$ return invalid

Where N is the order of the curve associated with the EC parameter set being used.

Note that in both cases, where the co-factor is 1, the algorithms are equivalent. In Bouncy Castle, EC algorithm names starting with "ECC" represent algorithms that adopt the approach favored by NIST, where the co-factor is simply multiplied in. Algorithms which *clear* the co-factor by multiplying it in and then multiplying in its multiplicative inverse start with "EC".

There are equivalent algorithms for EC key agreement to the ones defined for key agreement over Finite Fields, as well as a few extras - you can find the full list in Appendix C. As this is the case, we will provide a basic example for the simple case, but we will not provide full examples for all the different cases of key agreement as you can use the earlier ones as a guide.

The following code demonstrates how to generate an AES key using co-factor ECDH with a X9.63 KDF based on SHA-256.

[14] For the curious, this is known as a small subgroup attack.

```
  /**
   * Generate an agreed AES key value of 256 bits in length.
   *
   * @param aPriv Party A's private key.
   * @param bPub Party B's public key.
   * @param keyMaterial user keying material to mix with secret.
   * @return the generated AES key (256 bits).
   */
  public static SecretKey ecGenerateAESKey(
      PrivateKey aPriv, PublicKey bPub, byte[] keyMaterial)
      throws GeneralSecurityException
  {
      KeyAgreement agreement =
          KeyAgreement.getInstance("ECCDHwithSHA256KDF", "BC");

      agreement.init(aPriv, new UserKeyingMaterialSpec(keyMaterial));

      agreement.doPhase(bPub, true);

      return agreement.generateSecret("AES");
  }
```

Here is a simple example showing the `ecGenerateAESKey()` key method being used between two parties to generate a shared AES key:

```
public class ECCDHExample
{
    public static void main(String[] args)
        throws Exception
    {
        // Generate the key pairs for party A and party B
        KeyPair aKp = generateECKeyPair();
        KeyPair bKp = generateECKeyPair();

        // key agreement generating a shared secret
        byte[] keyMaterial = Strings.toByteArray("For an AES key");

        SecretKey aKey = ecGenerateAESKey(
            aKp.getPrivate(), bKp.getPublic(), keyMaterial);
        SecretKey bKey = ecGenerateAESKey(
            bKp.getPrivate(), aKp.getPublic(), keyMaterial);

        // compare the two return values.
        System.out.println(
            Arrays.areEqual(aKey.getEncoded(), bKey.getEncoded()));
    }
}
```

As you can see the pattern is identical to what we saw with Diffie-Hellman over Finite Fields. The only real change is in the way the keys are generated. In keeping with now established tradition, running this example should show the two keys generated are the same and produce the following output:

```
true
```

Adding Ephemeral Keys

Eliptic Curve based agreement also supports the Unified Model and MQV. Both of these approaches are documented in NIST SP 800-56A [9]. Elliptic Curve based MQV is also defined for use in CMS in RFC 5753 [76].

One other thing to be aware of is that some methods for performing ECMQV are currently subject to patents as an IPR disclosure associated with RFC 5753 [41] shows.

ECDH - The Unified Model

Under ECC the Unified Model follows the same process as we saw earlier. From party A's point of view the process follows the following steps:

1. A sends their public keys (W_sa, W_ea) to B
2. B sends their public keys (W_sb, W_eb) to A
3. A computes $(x_s, y_s) = d_saW_sb$. If $((x_s, y_s)$ is at $\infty)$ return invalid
4. A computes $(x_e, y_e) = d_eaW_eb$. If $((x_e, y_e)$ is at $\infty)$ return invalid
5. A returns $x_e \parallel x_s$

In NIST SP 80-56A the above is adapted to include multiplication with the co-factor as we saw before. The same could be done where the co-factor was *cleared* instead.

The following sample method shows the use of the co-factor ECDH under the Unified Model, assuming a full set of keys is passed in. Like the previous example of the Unified Model it is also basing the key derivation around an X9.63 KDF based on SHA-256.

```
/**
 * Generate an agreed AES key value of 256 bits in length
 * using the EC Unified Diffie-Hellman model.
 *
 * @param aPriv Party A's private key.
 * @param aPubEph Party A's ephemeral public key.
 * @param aPrivEph Party A's ephemeral private key.
 * @param bPub Party B's public key.
 * @param bPubEph Party B's ephemeral public key.
 * @return the generated AES key (256 bits).
 */
public static SecretKey ecdhuGenerateAESKey(
    PrivateKey aPriv, PublicKey aPubEph, PrivateKey aPrivEph,
    PublicKey bPub, PublicKey bPubEph, byte[] keyMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement =
        KeyAgreement.getInstance("ECCDHUwithSHA256CKDF", "BC");

    agreement.init(aPriv,
        new DHUParameterSpec(aPubEph, aPrivEph, bPubEph, keyMaterial));

    agreement.doPhase(bPub, true);

    return agreement.generateSecret("AES");
}
```

Note that the `DHUParameterSpec` class is the same one we used before. You can also use this method for the one-pass approach by invoking `ecdhuGenerateAESKey()` with the ephemeral keys of one party replaced with their static keys.

ECMQV

Elliptic Curve MQV also incorporates an implicit signature. The algorithm follows the following steps:

1. A generates ephemeral key pair (W_e, d_e)
2. B generates ephemeral key pair (W_{eb}, d_{eb})
3. B sends A W_{sb} and W_{eb}
4. Set $w = \lceil bitLength(N)/2 \rceil$
5. Set $(x, y) = W_e$
6. Set $\overline{\mathrm{W}} = (x \mod 2^w) + 2^w$
7. Set $s = (d_s\overline{\mathrm{W}} + d_e) \mod N$
8. Set $(x, y) = W_{eb}$

```
 9. Set $\overline{\mathrm{W_b}} = (x \mod 2^w) + 2^w$
10. Set $(x, y) = (hs\overline{\mathrm{W_b}} \mod N)W_{sb} + (hs \mod N)W_{eb}$
11. Return $x$
```

with steps 5 to 7 representing the implicit signature calculation.

The following example method shows the code required for generating a basic AES key using the ECMQV protocol and an X9.63 KDF based on SHA-256.

```java
/**
 * Generate an agreed AES key value of 256 bits in length
 * using ECMQV.
 *
 * @param aPriv Party A's private key.
 * @param aPubEph Party A's ephemeral public key.
 * @param aPrivEph Party A's ephemeral private key.
 * @param bPub Party B's public key.
 * @param bPubEph Party B's ephemeral public key.
 * @return the generated AES key (256 bits).
 */
public static SecretKey ecmqvGenerateAESKey(
    PrivateKey aPriv, PublicKey aPubEph, PrivateKey aPrivEph,
    PublicKey bPub, PublicKey bPubEph, byte[] keyMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement =
        KeyAgreement.getInstance("ECMQVwithSHA256KDF", "BC");

    agreement.init(aPriv,
        new MQVParameterSpec(aPubEph, aPrivEph, bPubEph, keyMaterial));

    agreement.doPhase(bPub, true);

    return agreement.generateSecret("AES");
}
```

Note that the `MQVParameterSpec` class is the same one we used before. You can also use this method for one-pass ECMQV by invoking `ecmqvGenerateAESKey()` with the ephemeral keys of one party replaced with their static keys. It is the one-pass approach for ECMQV that is defined for use in CMS in RFC 5753 [76].

Key Confirmation

Key Confirmation is described in NIST SP 800-56A [9], Section 5.9. It provides another way of either introducing, or strengthening an authentication mechanism being used in conjunction with a key agreement mechanism. The basic technique is not difficult to understand - the KDF used for generating a shared secret key is also used to generate a shared MAC key. This is done by splitting the derived key material generated by the KDF into two parts as in:

$$MacKey \parallel SecretKey = DerivedKeyMaterial$$

There is a lengthy discussion in NIST SP 800-56A, Sections 5.9.1 to 5.9.3, on what data is suitable for inclusion in the data processed by the MAC. Possible data can include things like public keys, the IDs of the participants, and/or a previously agreed secret. We will not reproduce the full discussion here, but there are two things worth noting. The first is that the strength of the MAC chosen should be at least as strong, in a bits of security strength, as the shared secret being generated is expected to be. The second, as you might expect, is that the MAC key should be discarded as soon as it is used, and should not be used for any other purpose.

At the moment, only the BCFIPS API supports key confirmation directly. A class `ByteMacData` is provided in `org.bouncycastle.crypto.util` in case you wish to follow the MAC data construction used in NIST SP 800-56A. When key confirmation is being used, the `KeyAgreement` class returns an `AgreedKeyWithMacKey` which is an object that provides access to both the derived shared secret key and the MAC key as well.

Generating an AgreedKeyWithMacKey

The BCFIPS API supports the return of two keys from the `KeyAgreement` class by extending the syntax allowed in the algorithm string passed to `KeyAgreement.generateSecret()`. The basic syntax follows the format:

“macAlgorithm[macKeySizeInBits]/cipherAlgorithm[cipherKeySizeInBits]”

The following method generates a 256 bit AES key like the methods we looked at earlier did. In this case the `generateSecret()` method is being passed a string saying “HmacSHA512[256]/AES[256]”, so rather than returning a simple `SecretKey` the method will return an `AgreedKeyWithMacKey` which includes a SHA-512 HMAC key of 256 bits in length as well.

```
/**
 * Generate an agreed AES key of 256 bits with an associated
 * 256 bit SHA-512 MAC key using ECMQV.
 *
 * @param aPriv Party A's private key.
 * @param aPubEph Party A's ephemeral public key.
 * @param aPrivEph Party A's ephemeral private key.
 * @param bPub Party B's public key.
 * @param bPubEph Party B's ephemeral public key.
 * @return an AgreedKeyWithMacKey containing the two keys.
 */
public static AgreedKeyWithMacKey keyConfGenerateAESKey(
    PrivateKey aPriv, PublicKey aPubEph, PrivateKey aPrivEph,
    PublicKey bPub, PublicKey bPubEph, byte[] keyMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement =
        KeyAgreement.getInstance("ECMQVwithSHA256KDF", "BCFIPS");

    agreement.init(aPriv,
        new MQVParameterSpec(aPubEph, aPrivEph, bPubEph, keyMaterial));

    agreement.doPhase(bPub, true);

    return (AgreedKeyWithMacKey)
                agreement.generateSecret("HmacSHA512[256]/AES[256]");
}
```

The order of the two keys reflects the order of the items as they are extracted from the derived key material. The initial 256 bits of derived key material being used as the source of the HMAC key, and the next 256 bits of derived key material being used for the AES key.

Calculating the MAC

Once the `AgreedKeyWithMacKey` has been returned, we should check the MAC before proceeding any further. In a FIPS environment we should also erase the MAC key as soon as we use it. The MAC key implements the interface `ZeroizableSecretKey` in order to make this possible.

The following method will verify a MAC calculated for a SHA-512 HMAC, the same MAC algorithm specified in our generate method. You can see the call to the `zeroize()` method on line 19.

```
    /**
     * Calculate a MAC tag value for the passed in MAC data.
     * Note: we zeroize the MAC key before returning the tag value.
     *
     * @param macKey key to initialize the MAC with.
     * @param macData data to calculate the MAC tag from.
     * @return the MAC tag value
     */
    public static byte[] calculateMAC(
        ZeroizableSecretKey macKey, byte[] macData)
        throws GeneralSecurityException
    {
        Mac mac = Mac.getInstance("HmacSHA512", "BCFIPS");

        mac.init(macKey);

        byte[] rv = mac.doFinal(macData);

        macKey.zeroize();

        return rv;
    }
```

As you would expect, once the MAC key has been erased it is no longer of any use.

Putting Everything Together

The following example shows the use of our agreement and MAC methods and also demonstrates a way of constructing the input data for the MAC using the Bouncy Castle `ByteMacData` builder class. You can see the use of the builder on lines 20 to 27 of the example.

```
 1  /**
 2   * Key Confirmation example showing use of two key pairs
 3   * per party in the protocol, with one set being regarded as ephemeral.
 4   */
 5  public class KeyConfExample
 6  {
 7      public static void main(String[] args)
 8          throws Exception
 9      {
10          // Generate the key pairs for party A and party B
11          KeyPair aKpS = generateECKeyPair();
```

```
        KeyPair aKpE = generateECKeyPair();    // A's ephemeral pair
        KeyPair bKpS = generateECKeyPair();
        KeyPair bKpE = generateECKeyPair();    // B's ephemeral pair

        // key agreement generating an AES key
        byte[] keyMaterial = Strings.toByteArray("For an AES key");

        // byte mac data
        ByteMacData byteMacData = new ByteMacData.Builder(
                ByteMacData.Type.BILATERALU,
                Strings.toByteArray("Party A"),    // party A ID
                Strings.toByteArray("Party B"),    // party B ID
                aKpE.getPublic().getEncoded(),        // ephemeral data A
                bKpE.getPublic().getEncoded())        // ephemeral data B
            .withText(Strings.toByteArray("hello, world!")) // optional shared
            .build();

        // A side.
        AgreedKeyWithMacKey aKey = keyConfGenerateAESKey(
            aKpS.getPrivate(),
            aKpE.getPublic(), aKpE.getPrivate(),
            bKpS.getPublic(), bKpE.getPublic(), keyMaterial);

        byte[] aTag = calculateMAC(aKey.getMacKey(), byteMacData.getMacData());

        // B side.
        AgreedKeyWithMacKey bKey = keyConfGenerateAESKey(
            bKpS.getPrivate(),
            bKpE.getPublic(), bKpE.getPrivate(),
            aKpS.getPublic(), aKpE.getPublic(), keyMaterial);

        byte[] bTag = calculateMAC(bKey.getMacKey(), byteMacData.getMacData());

        // compare the two return values.
        System.err.println("keys equal: "
            + Arrays.areEqual(aKey.getEncoded(), bKey.getEncoded()));
        System.err.println("tags equal: "
            + Arrays.areEqual(aTag, bTag));
    }
}
```

The builder setup is in accordance with the discussions in NIST SP 800-56A, Sections 5.9.1 to 5.9.3. You will also find the definitions for constants such as `BILATERALU` there as well. Fundamentally the constants are used to act as differentiators to help avoid accidental clashes when the MAC tag value is calculated for different situations.

If all goes well, at the end of the example, you will have equal agreed keys and equal MAC tag values and should see the following output:

```
keys equal: true
tags equal: true
```

If you try changing a few of the input values, you should see that it does not take much to cause a failure or interference in the agreement to be detected.

Summary

In this chapter we have looked into how public key cryptography can be used for transporting keys, performing key agreement and exchanging keys. This covered a wide range of options for algorithm choice with Elliptic Curve, Diffie-Hellman and their derived variants for each of the protocols. This chapter also covers extensions to many of the protocols improving their reliability.

We looked at range of options for wrapping and unwrapping keys using algorithms like RSA-OAEP, ElGamal-OAEP and RSA-KEM including worked examples for each.

For key agreement and key exchange, Diffie-Hellman features strongly in NIST, and as such we covered fully worked examples in this chapter using these algorithms.

Finally, we also looked at a series of improvements to the key agreement protocols using ephemeral keys. There are 2 main approaches covered in this chapter, the Unified Model, and MQV. In addition the key confirmation protocols defined in NIST SP 800-56A provide additional options for strengthening the key agreement mechanisms.

Chapter 8: X.509 Certificates and Attribute Certificates

While public key algorithms solve the “numerical” problems associated with being able to securely transport keys or provide people with a mechanism for verifying signatures you have digitally signed, the reality is that one bundle of numbers looks very much like another and there needs to be a way of associating an entity with a particular public key.

One of the most common ways of associating public keys with identities and asserting the validity of that association is by using the X.509 [25] certificate structure. X.509 Certificates are used to associated a key with an identity. This identity is the subject of the certificate. Another identity, the signer of the certificate who is also regarded as the certifying authority (CA), also known as the certificate issuer, is also stored in the certificate so that it is (hopefully) possible to trace the certificate’s provenance.

In this chapter we will look at how public key certificates are created as well as how additional attributes can be associated with them after creation. In dealing with this, the first thing we need to look at is how certificates are associated with identities in the first place.

The X.500 Distinguished Name

The fundamental unit of identity for certificates is the X.500 distinguished name. The main ASN.1 structure in X.500 is the Name and we will see that it is used to populate the identity fields in X.509 certificates as well as providing a way of representing identities in extensions and other fields.

```
Name ::= CHOICE {
              RDNSequence }

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET SIZE (1..MAX) OF AttributeTypeAndValue

AttributeTypeAndValue ::= SEQUENCE {
                             type  OBJECT IDENTIFIER,
                             value ANY }
```

> Note that the Name structure's ASN.1 definition means it is a CHOICE item. For this reason X.500 names cannot be implicitly tagged, they must always be tagged explicitly, even in a situation where an ASN.1 module may appear to suggest otherwise.

In Bouncy Castle the X.500 name is represented by the `X500Name` class which is defined under the hierarchy `org.bouncycastle.asn1.x500`. You will probably be wondering why an entire package hierarchy is devoted to such a simple structure. The reality is that while the structure is simple, its uses and applications are quite diverse - to the point where, as pointed out in Peter Gutmann's excellent "X.509 Style Guide" [37], you could easily include an MPEG movie showing someone playing with their cat!

At the moment the X500 package hierarchy consists of 2 packages, the core `x500` package and a sub-package `x500.style`. The core package contains the `X500Name` class as well as the `X500NameBuilder` class and also component classes representing the items that the SEQUENCE making up a Name is made of: `RDN` being RelativeDistinguishedName (RDN) and `AttributeTypeAndValue` representing the AttributeTypeAndValue (ATV) structure. The style package contains template classes that provide ways of configuring the `X500Name` objects to conform to different standards. In the situation where you find yourself trying to conform to a profile for Distinguished Names where the BC classes do not quite fit, the style classes can be extended, or copied and edited, in order to produce something that is appropriate.

The JCA relies on `X500Principal` for supporting X.500 Name structures. Unlike the BC's `X500Name` class, an `X500Principal` can only be created from either a string or a byte encoding - there is no builder class. For this reason, even if you are restricting yourself to using `X500Principal` it can still sometimes be better to use the X500NameBuilder for constructing a name and then use `X500Name.getEncoded()` to create a new `X500Principal` from the byte encoding as in:

```
X500NameBuilder x500Bldr = new X500NameBuilder();

x500Bldr.addRDN(BCStyle.CN, "Example X.500 Name");

X500Principal x500Principal = new X500Principal(x500Bldr.build().getEncoded())
```

This especially applies in the situation where you adding multiple ATVs to a single RDN.

> ⚠ If you are adding multiple ATVs to an RDN remember X.500 Names are generally DER encoded. As the ATV uses a SET this will mean it is also sorted according to the encoded value of the members.

You can convert an `X500Principal` to an `X500Name` by using the `X500Name`'s `getInstance()` method.

```
X500Name x500Name = X500Name.getInstance(x500Principal.getEncoded())
```

Where possible, avoid using simple string representations, or use them carefully when you do - there are a couple of widely used variations in how X.500 names are converted into strings. Both the `X500Name` class and the `X500Principal` class support the conversion of strings to X.500 Name objects, but what you get is a matter of how the different classes interpret the input. We will have a look at some of the issues associated with that now.

X.500 Meets the String

The first issue you may notice with X.500 Names is that there are multiple ways of producing a string representation of one. Part of this is how shorthands for the different ASN.1 OBJECT IDENTIFIERs (OIDs) are used to represent the type of the attributes (for example "CN" actually is a short hand for the OID "2.5.4.3"). The second problem is there are actually two ways of ordering the SEQUENCE making up the Name, depending on which convention is used. The first convention, the ISO one, converts the sequence in order, so the first part of the string representing an RDN is the first RDN, and so on. The second convention, used by the IETF and specified in RFC 4519 [63], does the conversion by converting the sequence in reverse order, so the first part of the string representing an RND is actually the last RDN in the sequence.

How to choose? These days the IETF ordering is probably the most common, that said if you are producing, or consuming, X.500 Names as strings, it may save future embarrassment to check if the profile under which the work is being done has chosen a specific ordering.

In Bouncy Castle's case the default ordering is the ISO one. The JCA's `X500Principal` class follows the IETF one.

Bouncy Castle's `X500Name` class does allow for the definition of a local style though, so, for example, you can use the following method to specify that a `X500Name` should be converted to a string using the approach taken in RFC 4519.

```
/**
 * Convert an X500Name to use the IETF style.
 */
public static X500Name toIETFName(X500Name name)
{
    return X500Name.getInstance(RFC4519Style.INSTANCE, name);
}
```

X.500 Meets Equality

There are also multiple ways of comparing X.500 names for equality, the most general one compares RDNs for equality regardless of their position in the sequence only checking that the RDN is present rather than in the same spot in both cases. The elements of the RDN containing text may also be compared case insensitive, or case sensitive, depending on what they are.

Bouncy Castle's `X500Name` class uses this approach by default. The JCA's `X500Principal` class converts the Names being compared to a canonical string representation which it then compares internally.

Public Key Certificates

As with X.500, X.509 itself is quite generous in terms of what it makes possible, so people generally adopt a particular profile for creating and evaluating certificates. For the purposes of the Internet the most popular is RFC-5280 [72]. While profiles may vary in terms of how things are interpreted the format is always that described in X.509.

Certificate and TBSCertificate

Before we start looking at the creation of certificates it is worth having a quick look at the ASN.1 structures involved in order to get some idea of what is going on underneath the API. While there are few things in this world that will fill a developer with the kind trepidation that an ASN.1 module will, it is worth getting your head around the concepts as it makes it a lot easier to debug situations that have gone wrong if you have some idea of what is really going on "inside the box".

The below ASN.1 fragment provides the core definitions for an X.509 certificate. You can find more in depth details as to what this means in the discussion in Appendix A if you need it.

```
Certificate  ::=  SEQUENCE  {
     tbsCertificate       TBSCertificate,
     signatureAlgorithm   AlgorithmIdentifier,
     signature            BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
     version         [0]  Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version MUST be v2 or v3
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version MUST be v2 or v3
     extensions      [3]  Extensions OPTIONAL
```

```
                          -- If present, version MUST be v3 --  }

Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

CertificateSerialNumber  ::=  INTEGER

Validity ::= SEQUENCE {
     notBefore      Time,
     notAfter       Time  }

Time ::= CHOICE {
     utcTime        UTCTime,
     generalTime    GeneralizedTime }

UniqueIdentifier  ::=  BIT STRING

SubjectPublicKeyInfo  ::=  SEQUENCE  {
     algorithm            AlgorithmIdentifier,
     subjectPublicKey     BIT STRING  }

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
     extnID      OBJECT IDENTIFIER,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue   OCTET STRING
                 -- contains the DER encoding of an ASN.1 value
                 -- corresponding to the extension type identified
                 -- by extnID
     }
```

The first and most important thing to be aware of is the TBSCertificate structure contains the body of the certificate, including the public key (encoded in the SubjectPublicKeyInfo). The second thing is that the signature stored in Certificate is calculated over the DER encoding of the TBSCertificate (TBS in this case actually stands for “To Be Signed”). The use of DER, or Distinguished Encoding Rules, is important here as it means that the certificate should always evaluate the same way when its signature hash is calculated.

The second thing to be aware of is the validity period, certificates are only issued to be valid for a specific period of time. This determination is made by the CA and will reflect a range of considerations including the strength of the signing algorithm, how long the entity the certificate is for (the subject) should have a certificate in the first place, and the strength of the public key and the number of usages it, or its associated private key might get.

Finally, there is the version. Its very rare to see v2 certificates - the issuerUniqueID and subjectUniqueID were added for v2, and it turned out that perhaps it was one of those ideas best left on the drawing board. Normally you will only encounter v1 and v3 certificates, and if you see a v1 certificate it will normally be self-signed and being used as a trust anchor. Certificates with a version of v3 generally have extensions, in day to day use they are the ones that we most often do things with, such as verifying documents or timestamps, or encrypting secret keys. We will have a closer look at the common extensions in certificates further on.

As you can see the TBSCertificate also contains the identity information we need, both to help us identify the issuer, the subject, and also the serial number so that we can tell which certificate that came from the issuer we are looking at. Apart from being handy to know, this also makes it possible for the issuer to revoke the certificate, a topic which we will look at later in this chapter.

Creating a Basic Public Key Certificate

Before we look at generating certificates, a couple of utility methods would be useful to define. One to provide an appropriate date:

```
/**
 * Calculate a date in seconds (suitable for the PKIX profile - RFC 5280)
 *
 * @param hoursInFuture hours ahead of now, may be negative.
 * @return a Date set to now + (hoursInFuture * 60 * 60) seconds
 */
public static Date calculateDate(int hoursInFuture)
{
     long secs = System.currentTimeMillis() / 1000;

     return new Date((secs + (hoursInFuture * 60 * 60)) * 1000);
}
```

and one to provide us with serial numbers:

```
private static long serialNumberBase = System.currentTimeMillis();

/**
 * Calculate a serial number using a monotonically increasing value.
 *
 * @return a BigInteger representing the next serial number in the sequence.
 */
public static synchronized BigInteger calculateSerialNumber()
{
    return BigInteger.valueOf(serialNumberBase++);
}
```

You will see that the Bouncy Castle APIs generate an X509CertificateHolder class. These have the advantage of being provider independent as they simply act as a carrier for the certificate. We will look at converting the X509CertificateHolder into one of Java's X509Certificate objects after we have looked at how to build a basic certificate.

The following example shows how to generate a self-signed version 1 certificate. Version 1 certificates are still used extensively, but only as trust anchors, as it is otherwise impossible to restrict their use. We will see examples of how a certificate's use can be limited when we look at certificates that carry extensions.

```
/**
 * Build a sample self-signed V1 certificate to use as a trust anchor, or
 * root certificate.
 *
 * @param keyPair the key pair to use for signing and providing the
 *                public key.
 * @param sigAlg the signature algorithm to sign the certificate with.
 * @return an X509CertificateHolder containing the V1 certificate.
 */
public static X509CertificateHolder createTrustAnchor(
        KeyPair keyPair, String sigAlg)
        throws OperatorCreationException
{
    X500NameBuilder x500NameBld = new X500NameBuilder(BCStyle.INSTANCE)
            .addRDN(BCStyle.C, "AU")
            .addRDN(BCStyle.ST, "Victoria")
            .addRDN(BCStyle.L, "Melbourne")
            .addRDN(BCStyle.O, "The Legion of the Bouncy Castle")
            .addRDN(BCStyle.CN, "Demo Root Certificate");

    X500Name name = x500NameBld.build();
22
23          X509v1CertificateBuilder certBldr = new JcaX509v1CertificateBuilder(
24                                                      name,
25                                                      calculateSerialNumber(),
26                                                      calculateDate(0),
27                                                      calculateDate(24 * 31),
28                                                      name,
29                                                      keyPair.getPublic());
30
31          ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
32                                      .setProvider("BC").build(keyPair.getPrivate());
33
34          return certBldr.build(signer);
35      }
```

On lines 14 to 19 we can see the use of the X500NameBuilder to create an X.500 name to provide the details of the identity to be linked to the certificate. The content of the certificate is then built on lines 23 to 29 and a signer is created using the private key and the signature algorithm name passed in to the method.

As the certificate is self signed, both the subject and the issuer in the certificate body are set to the same name. If you are given one these, you really have to take the presenter's word for it.

To call the method, we just need to decide on what public key algorithm to use and what signature algorithm to use to sign the certificate with. In the following code fragment we can see how we would call the method to create a self-signed certificate signed with SHA-256 using ECDSA.

```
KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");
KeyPair trustKp = kpGen.generateKeyPair();

X509CertificateHolder trustCert =
                          createTrustAnchor(trustKp, "SHA256withECDSA");
```

Converting an X509CertificateHolder to an X509Certificate

The `org.bouncycastle.cert.jcajce.JcaX509CertificateConverter` class is provided to convert `X509CertificateHol` objects into regular JCA X509Certificate objects.

The `JcaX509CertificateConverter` class can be used to generate a certificate according to the current provider priority in the JVM, or you can specify a particular provider. In this example we have specified the Bouncy Castle provider by using its name.

```
X509CertificateHolder certHldr = ...

X509Certificate cert = new JcaX509CertificateConverter()
                                                   .setProvider("BC")
                                                   .getCertificate(certHldr);
```

To go back the other way, simply pass the encoding of the `X509Certificate` to the `X509CertificateHolder`, or pass the `X509Certificate` to a `JcaX509CertificateHolder` which will do the encoding step for you.

```
X509Certificate cert = ...
X509CertificateHolder certHldr = new JcaX509CertificateHolder(cert);
```

As we will see next, you can also convert an `X509CertificateHolder` using a `CertificateFactory` class.

The CertificateFactory Class

The `CertificateFactory` class in the `java.security.cert` package can also be used to convert an `X509CertificateHolder`, or a number of other forms of an X.509 certificate into one of Java's `X509Certificate` objects. We will see further uses of this class in the next chapter as well as `CertificateFactory` is also used for Certificate Revocation Lists (CRLs), and certificate path construction and validation.

The `CertificateFactory` class is also provider based and requires the use of a `getInstance()` method to instantiate one. It has two methods associated with producing public key certificates: `generateCertificate()` and `generateCertificates()`. These methods can be used to read in various encodings of certificates, usually including PEM and BER/DER, and then produce either a single certificate (in the case of `generateCertificate()`) or a collection of certificates (in the case of `generateCertificates()`).

The `getEncoded()` method of `X509CertificateHolder` produces a BER encoding of the X.509 certificate wrapped in the class. You can use this method to also provide input for the `generateCertificate()` method on the `CertificateFactory` in the following way:

```
/**
 * Simple method to convert an X509CertificateHolder to an X509Certificate
 * using the java.security.cert.CertificateFactory class.
 */
public static X509Certificate convertX509CertificateHolder(
                                         X509CertificateHolder certHolder)
        throws GeneralSecurityException, IOException
{
    CertificateFactory cFact = CertificateFactory.getInstance("X.509", "BC");

    return (X509Certificate)cFact.generateCertificate(
                                       new ByteArrayInputStream(
                                               certHolder.getEncoded()));
}
```

Creating a CA Certificate with Extensions

The principal innovation with the release of X.509 version 3 was the introduction of certificate extensions. Before we proceed it would be a good idea just to revise how an extension is encoded using the ASN.1 Extension structure.

```
Extension  ::=  SEQUENCE  {
     extnID      OBJECT IDENTIFIER,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue   OCTET STRING
                 -- contains the DER encoding of an ASN.1 value
                 -- corresponding to the extension type identified
                 -- by extnID
     }
```

The OBJECT IDENTIFIER extnID is used to identify the contents of extnValue, critical is used to indicate whether or not the CA felt this extension was important enough that it must be understood by anyone wishing to use the certificate it is in, and then extnValue is an OCTET STRING holding a DER encoding of whatever the extension value actually is. Note that as critical has a DEFAULT value of false, if it is false the value will be absent when the Extension structure is DER encoded.

Extensions provide two benefits, they allow extra information about the certificate's subject to be added, such as other names they/it might be known by and they also make it possible for the CA to place some constraints on what purposes the certificate can be used for. This is an important thing to remember when we look at CertificationRequests later, while it is possible to request an extension to be added, what extensions are added to a certificate are determined by the CA signing the certificate.

This also allows for CAs to safely create other CAs to act on their behalf. A certificate that cannot be used for a CA terminates the chain of certificates, or certificate path, that leads from the trust anchor at the top of the path. A certificate terminating the chain is called an end-entity certificate. End-entity certificates are most commonly tied directly with identity of specific services, organisations, or individuals.

While the `X509CertificateHolder` presents extensions as objects, if you are using a JCA `X509Certificate` there will either be a specific method associated with the extension, or you will need to use the `X509Certificate.getExtensionValue()` to retrieve the extension value. Note that the extension value returned will be the DER encoded OCTET STRING containing the extension value, not the extension value itself. You can unwrap the extension value and get the content bytes representing the extension value with the following method:

```java
/**
 * Extract the DER encoded value octets of an extension from a JCA
 * X509Certificate.
 *
 * @param cert the certificate of interest.
 * @param extensionOID the OID associated with the extension of interest.
 * @return the DER encoding inside the extension, null if extension missing.
 */
public static byte[] extractExtensionValue(
        X509Certificate cert,
        ASN1ObjectIdentifier extensionOID)
{
    byte[] octString = cert.getExtensionValue(extensionOID.getId());

    if (octString == null)
    {
        return null;
    }

    return ASN1OctetString.getInstance(octString).getOctets();
}
```

where extensionOID is the OBJECT IDENTIFIER associate with the extension.

Common extensions are defined in RFC 5280 and can require calculations. If you are using JCA keys, the JcaX509ExtensionUtils class can be used to do these calculations for you and help create the appropriate extension value.

Consider the following example, which creates an intermediate certificate which can be used to sign other certificates. The method make use of the JcaX509ExtensionUtils class and creates a number of extensions in order to provide extra information about the issuer, a general identifier for the certificate's public key, and some details about what the CA that signed the certificate expected the certificate to be used for.

```
/**
 * Build a sample V3 intermediate certificate that can be used as a CA
 * certificate.
 *
 * @param signerCert certificate carrying the public key that will later
 *                    be used to verify this certificate's signature.
 * @param signerKey private key used to generate the signature in the
 *                   certificate.
 * @param sigAlg the signature algorithm to sign the certificate with.
 * @param certKey public key to be installed in the certificate.
 * @param followingCACerts
 * @return an X509CertificateHolder containing the V3 certificate.
 */
public static X509CertificateHolder createIntermediateCertificate(
        X509CertificateHolder signerCert, PrivateKey signerKey,
        String sigAlg, PublicKey certKey, int followingCACerts)
        throws CertIOException, GeneralSecurityException,
               OperatorCreationException
{
    X500NameBuilder x500NameBld = new X500NameBuilder(BCStyle.INSTANCE)
            .addRDN(BCStyle.C, "AU")
            .addRDN(BCStyle.ST, "Victoria")
            .addRDN(BCStyle.L, "Melbourne")
            .addRDN(BCStyle.O, "The Legion of the Bouncy Castle")
            .addRDN(BCStyle.CN, "Demo Intermediate Certificate");

    X500Name subject = x500NameBld.build();

    X509v3CertificateBuilder certBldr = new JcaX509v3CertificateBuilder(
            signerCert.getSubject(),
            calculateSerialNumber(),
            calculateDate(0),
            calculateDate(24 * 31),
            subject,
            certKey);

    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();

    certBldr.addExtension(Extension.authorityKeyIdentifier,
            false, extUtils.createAuthorityKeyIdentifier(signerCert))
            .addExtension(Extension.subjectKeyIdentifier,
                    false, extUtils.createSubjectKeyIdentifier(certKey))
            .addExtension(Extension.basicConstraints,
                    true, new BasicConstraints(followingCACerts))
            .addExtension(Extension.keyUsage,
                    true, new KeyUsage(
                                          KeyUsage.digitalSignature
                                        | KeyUsage.keyCertSign
                                        | KeyUsage.cRLSign));

    ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
            .setProvider("BC").build(signerKey);

    return certBldr.build(signer);
}
```

Given the `PrivateKey` for the trust anchor `trustAnchorKey` and its associated certificate `trustCert` we could call the method above to create a single level CA certificate as follows:

```
KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");
KeyPair caKp = kpGen.generateKeyPair();

X509CertificateHolder caCert =
        createIntermediateCertificate(trustCert,
                trustAnchorKey,
                "SHA256withECDSA", caKp.getPublic(), 0);
```

Note that the call passed 0 as the value of `followingCACerts`. We will see what that means when we look at the BasicConstraints extension below.

Common Certificate Extensions

Having seen extensions getting added, it is worth devoting a bit of time to understanding what the ones used in the example and the common ones actually mean. The extensions covered here should only be taken as a guide, if you need to be able to interpret them at any depth you should check the relevant sections of RFC 5280 [72]..

Basic Constraints

The basicConstraints extension is identified by the constant Extension.basicConstraints which has OID value “2.5.29.19” (id-ce-basicConstraints). Its ASN.1 definition looks like this:

```
BasicConstraints ::= SEQUENCE {
    cA                 BOOLEAN DEFAULT FALSE,
    pathLenConstraint INTEGER (0..MAX) OPTIONAL }
```

The BasicConstraints extension helps you to determine if the certificate containing it is allowed to sign other certificates, and if so what depth this can go to. So, for example, if cA is TRUE and the pathLenConstraint is 0, then the certificate, as far as this extension is concerned, is allowed to sign other certificates, but none of the certificates so signed can be used to sign other certificates and lengthen the chain. This is a very important extension to set if you are trying to avoid people somehow issuing extra certificates under one you have signed.

In the first example the BasicConstraints extension is added in the lines:

```
        .addExtension(Extension.basicConstraints,
                        true, new BasicConstraints(followingCACerts))
```

As a pathLenConstraint is provided in the form of the followingCACerts variable, this tells us the certificate is being issued as a CA in its own right, with the provision that no more than followingCACerts of CAs can follow it, with a value of zero meaning that only an end-entity certificate can follow.

To recover a BasicConstraints using a X509CertificateHolder class you can use:

```
BasicConstraints basicConstraints = BasicConstraints.fromExtensions(
                                                     certHldr.getExtensions());
```

to recover the object.

In the case of Java's X509Certificate class, a method `X509Certificate.getBasicConstraints()` is provided which returns an int. The int value represents the pathLenConstraint unless cA is FALSE in which case the return value of getBasicConstraints() is -1.

Authority Key Identifier

The authorityKeyIdentifier extension is identified by the constant `Extension.authorityKeyIdentifier` which has the OID value "2.5.29.35" (id-ce-authorityKeyIdentifier). Its ASN.1 definition looks like this:

```
AuthorityKeyIdentifier ::= SEQUENCE {
    keyIdentifier [0] KeyIdentifier OPTIONAL,
    authorityCertIssuer [1] GeneralNames OPTIONAL,
    authorityCertSerialNumber [2] CertificateSerialNumber OPTIONAL }

KeyIdentifier ::= OCTET STRING
```

The object of this extension is to identify the public key that can be used to verify the signature on the certificate, or put another way, to verify the signature on the certificate which ties the public key on the certificate to the subject of the certificate and any associated extensions. Because of this if you are willing to accept the public key identified by this extension as being authoritative, it makes sense for you to accept the public key at its associated attributes stored in the certificate as being valid as well. On the other hand, if you can't make sense of this extension or verify the signer of the certificate, accepting anything in the certificate is more of an act of blind faith, rather than an act based on the validity of the signing algorithim used.

To recover a AuthorityKeyIdentifier using a X509CertificateHolder class you can use:

```
AuthorityKeyIdentifier authorityKeyIdentifier =
                AuthorityKeyIdentifier.fromExtensions(certHldr.getExtensions());
```

to recover the object.

In the case of Java's X509Certificate class, there is no method for directly retrieving the AuthorityKeyIdentifier as an object. If you want to create an AuthorityKeyIdentifier object from one of these you can make use of the extractExtensionValue method we defined earlier as follows:

```
X509Certificate cert = ...

AuthorityKeyIdentifier authorityKeyIdentifier =
     AuthorityKeyIdentifier.getInstance(
                 extractExtensionValue(cert, Extension.authorityKeyIdentifier));
```

Subject Key Identifier

The subjectKeyIdentifier extension is identified by the constant `Extension.subjectKeyIdentifer` which has the OID value "2.5.29.14" (id-ce-subjectKeyIdentifier). Its ASN.1 definition looks like this:

```
SubjectKeyIdentifier ::= KeyIdentifier
```

The subjectKeyIdentifier is simply a string of octets which is used to provide an identifier for the public key the certificate contains. For example if you were looking for the issuing certificate for a particular certificate and the AuthorityKeyIdentifier for the particular certificate had the keyIdentifier field set you would expect to find the value stored in keyIdentifier in the SubjectKeyIdentifier extension of the issuing certificate. RFC 5280, section 4.2.1.2, gives 2 common methods for calculating the key identifier value.

To recover a SubjectKeyIdentifier using a X509CertificateHolder class you can use:

```
SubjectKeyIdentifier subjectKeyIdentifier = SubjectKeyIdentifier.fromExtensions(
                                                     certHldr.getExtensions());
```

to recover the object.

In the case of Java's X509Certificate class, there is no method for directly retrieving the SubjectKeyIdentifier as an object. If you want to create an SubjectKeyIdentifier object from one of these you can make use of the extractExtensionValue method we defined earlier as follows:

```
X509Certificate cert = ...

SubjectKeyIdentifier subjectKeyIdentifier =
     SubjectKeyIdentifier.getInstance(
                 extractExtensionValue(cert, Extension.subjectKeyIdentifier));
```

Subject Alternative Name

The subjectAltName extension is identified by the constant `Extension.subjectAlternativeName` which has the OID value "2.5.29.17" (id-ce-subjectAltName). Its ASN.1 definition looks like this:

```
SubjectAltName ::= GeneralNames

GeneralNames ::= SEQUENCE SIZE (1..MAX) OF GeneralName
```

The subjectAltName is used to store alternate, or alias, subject names to associate with a certificate. If you want to associate an email address with a certificate, strictly speaking, this is the best place to put it.

> It is worth noting that the PKIX profile allows for certificates where the subject name field in the TBSCertificate is an empty sequence with the actual subject details stored in the subjectAltName extension.

To recover the GeneralNames object for a subjectAltName extension using a X509CertificateHolder class you can use:

```
GeneralNames subjectAltName = GeneralNames.fromExtensions(
                   certHldr.getExtensions(), Extension.subjectAlternativeName);
```

to recover the object.

In the case of Java's X509Certificate class, the method X509Certificate.getSubjectAlternativeNames() is provided which returns an immutable collection of List objects representing the various objects that can be found in the extension. Each List has two elements, with the first being the tag associated with the GeneralName and the second element being an interpretation of the value of the GeneralName, be it a String, or a DER encoded byte array. Depending on what you are planning to do with this one, it may be simpler just to create a GeneralNames object using the extractExtensionValue() method we used earlier.

Issuer Alternative Name

The issuerAltName extension is identified by the constant Extension.issuerAlternativeName which has the OID value "2.5.29.18" (id-ce-issuerAltName). Its ASN.1 definition looks like this:

```
IssuerAltName ::= GeneralNames
```

Like the subjectAltName the issuerAltName is used to store alternate names that can be associated with the certificate issuer. Unlike the subjectAltName extension this extension cannot act as a substitute for the contents of the issuer filed in the TBSCertificate structure. To recover the `GeneralNames` object for a issuerAltName extension using a `X509CertificateHolder` class you can use:

```
GeneralNames issuerAltName = GeneralNames.fromExtensions(
                     certHldr.getExtensions(), Extension.issuerAlternativeName);
```

to recover the object.

In the case of Java's `X509Certificate` class, this extension is handled the same way as the SubjectAltName extension and a method `X509Certificate.getIssuerAlternativeNames()` is provided which returns an immutable collection of List objects representing tuples describing the various objects that can be found in the extension. Likewise with the SubjectAltName extension you may find you are better off handling this extension as a `GeneralNames` object directly.

CRL Distribution Point

The CRL distribution point extension provides a means to identify how CRL information is obtained. The extension should be non-critical and is identified by the OID value "2.5.29.31" (id-ce-cRLDistributionPoints).

The syntax for the extension is described in RFC 5280 [72] as follows:

```
id-ce-cRLDistributionPoints OBJECT IDENTIFIER ::=  { id-ce 31 }

CRLDistributionPoints ::= SEQUENCE SIZE (1..MAX) OF DistributionPoint

DistributionPoint ::= SEQUENCE {
    distributionPoint       [0]     DistributionPointName OPTIONAL,
    reasons                 [1]     ReasonFlags OPTIONAL,
    cRLIssuer               [2]     GeneralNames OPTIONAL }

DistributionPointName ::= CHOICE {
    fullName                [0]     GeneralNames,
    nameRelativeToCRLIssuer [1]     RelativeDistinguishedName }

ReasonFlags ::= BIT STRING {
    unused                  (0),
    keyCompromise           (1),
    cACompromise            (2),
    affiliationChanged      (3),
    superseded              (4),
    cessationOfOperation    (5),
    certificateHold         (6),
    privilegeWithdrawn      (7),
    aACompromise            (8) }
```

The CRLDistributionPoints structure is represented in the BC library by the CRLDistPoint class. You can recover a CRL distribution point extension value from an `X509CertificateHolder` using the following:

```
CRLDistPoint crlDistPoints =
                    CRLDistPoint.getInstance(
                        certHldr.getExtensions().getExtension(
                            Extension.cRLDistributionPoints).getParsedValue());
```

As you can see the extension can contain information about where CRL information can be obtained as well as what reasons the CRL will be covering. There is quite a lengthy discussion in RFC 5280 concerning this extension which we will not reproduce here, but we will make the following point: a little caution is warranted about blindly processing this one - accepting a direction to go to an arbitrary URL potentially downloading a huge file of unknown size is not something to be taken lightly. For anything Internet facing you really want to abstract this process in some fashion so you can be protected from the fact that not all the certificates you receive are likely to be well intentioned.

Key Usage

The keyUsage extension is identified by the constant Extension.keyUsage which has the OID value “2.5.29.15” (id-ce-keyUsage). Its ASN.1 definition looks like this:

```
KeyUsage ::= BIT STRING {
    digitalSignature      (0),
    nonRepudiation        (1),
    keyEncipherment       (2),
    dataEncipherment      (3),
    keyAgreement          (4),
    keyCertSign           (5),
    cRLSign               (6),
    encipherOnly          (7),
    decipherOnly          (8) }
```

The keyUsage extension is the most general way of restricting the uses of the key contained in the certificate. Which bits must, or must not be set, depends largely on the profile the certificate is been used with and what purpose it has. As an example, a certificate used for client-side SSL might have only the digitalSignature bit set as the public key in the certificate should only be used for signature verification operations. RFC 5280, section 4.2.1.3 discusses the range of possibilities as well as providing some pointers to other RFCs that place specific interpretations on keyUsage.

To recover a KeyUsage using a X509CertificateHolder class you can use:

```
KeyUsage keyUsage = KeyUsage.fromExtensions(certHldr.getExtensions());
```

to recover the object.

The Java `X509Certificate` class provides specific handling for this extension with a specific method `X509Certificate.getKeyUsage()` which returns an array of booleans representing each bit in the KeyUsage bitstring. Note that in this case the order of the booleans reflects their numbering in the ASN.1 definition, not the actual order in which the bits are set in the `BIT STRING`.

Extended Key Usage

The extKeyUsage extension is identified by the constant `Extension.extendedKeyUsage` which has the OID value "2.5.29.37" (id-ce-extKeyUsage). Its ASN.1 definition looks like this:

```
ExtKeyUsageSyntax ::= SEQUENCE SIZE (1..MAX) OF KeyPurposeId
KeyPurposeId ::= OBJECT IDENTIFIER
```

If this extension is present the certificate is meant to be used for only one of the purposes listed in it, unless the special KeyPurposeId anyExtendedKeyUsage is included in the ExtKeyUsageSyntax sequence. There are a range of key purpose IDs allocated off the arc `id-kp` ("1.3.6.1.5.5.7.3").

Commonly this extension is used with end entity certificates to lock the certificate down to a specific purpose in a more rigid way than allowed by the key usage extension. For example a certificate used for verifying time-stamps must also include the ExtendedKeyUsage id-kp-timeStamping, which is defined as:

```
    id-kp-timeStamping OBJECT IDENTIFIER ::= {iso(1)
                    identified-organization(3) dod(6)
                    internet(1) security(5) mechanisms(5) pkix(7)
                    kp (3) timestamping (8)}
```

indicating that the certificate public key can only be used for verifying time-stamps. The ExtendedKeyUsage extension in this case would also be expected to be marked as critical and the presence of the ExtendedKeyUsage extension is clearly a stronger assertion than is possible by just setting the digitalSignature bit in the KeyUsage extension.

To recover an ExtendedKeyUsage using a X509CertificateHolder class you can use:

```
ExtendedKeyUsage extKeyUsage = ExtendedKeyUsage.fromExtensions(
                                                     certHldr.getExtensions());
```

In the case of a Java `X509Certificate` class, this extension is also represented by a specific method `X509Certificate.getExtendedKeyUsage()` which returns a list of strings representing the OBJECT IDENTIFIER values that have been set in the extension.

If an extended key usage and a key usage extension are used together, make sure they are compatible. RFC 5280 directs that a certificate cannot be used for any purpose if the extended key usage extension and key usage extension conflict.

Note also that if you need to use anyExtendedKeyUsage (say, because an application you are working with requires it), the extension should not be marked as critical.

Creating End-Entity Certificates

As we mentioned earlier, end-entity certificates represent the end of the certificate chain. If you want a certificate which ties back to a particular user, or the activity of a particular user it will normally be an end-entity certificate that is associated with them.

The following example creates an end-entity certificate which is authorised to be used for the verification of digital signatures. Note that the example includes a BasicConstraints extension, but that this time the extension value is not created with a path length, but with a boolean value of false.

```
/**
 * Create a general end-entity certificate for use in verifying digital
 * signatures.
 *
 * @param signerCert certificate carrying the public key that will later
 *                   be used to verify this certificate's signature.
 * @param signerKey private key used to generate the signature in the
 *                  certificate.
 * @param sigAlg the signature algorithm to sign the certificate with.
 * @param certKey public key to be installed in the certificate.
 * @return an X509CertificateHolder containing the V3 certificate.
 */
public static X509CertificateHolder createEndEntity(
        X509CertificateHolder signerCert, PrivateKey signerKey,
        String sigAlg, PublicKey certKey)
        throws CertIOException, GeneralSecurityException,
               OperatorCreationException
{
    X500NameBuilder x500NameBld = new X500NameBuilder(BCStyle.INSTANCE)
            .addRDN(BCStyle.C, "AU")
            .addRDN(BCStyle.ST, "Victoria")
            .addRDN(BCStyle.L, "Melbourne")
            .addRDN(BCStyle.O, "The Legion of the Bouncy Castle")
            .addRDN(BCStyle.CN, "Demo End-Entity Certificate");

    X500Name subject = x500NameBld.build();

    X509v3CertificateBuilder   certBldr = new JcaX509v3CertificateBuilder(
            signerCert.getSubject(),
            calculateSerialNumber(),
            calculateDate(0),
            calculateDate(24 * 31),
            subject,
            certKey);

    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();

    certBldr.addExtension(Extension.authorityKeyIdentifier,
            false, extUtils.createAuthorityKeyIdentifier(signerCert))
            .addExtension(Extension.subjectKeyIdentifier,
                    false, extUtils.createSubjectKeyIdentifier(certKey))
            .addExtension(Extension.basicConstraints,
                    true, new BasicConstraints(false))
```

```
                .addExtension(Extension.keyUsage,
                        true, new KeyUsage(KeyUsage.digitalSignature));

        ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
                                        .setProvider("BC").build(signerKey);

        return certBldr.build(signer);
    }
```

Note that in this case the constructor for `BasicConstraints` is just being passed the boolean value of false. This is to indicate that the certificate cannot be used as a CA certificate, even though it may be used for verifying a signature. If we are presented with a certificate signed by a certificate configured like the one in the example above, we should reject it out of hand as it is clear that the signing certificate was never created with the intention of making it possible for the certificate path to become longer.

Using ExtendedKeyUsage

In the second example we are creating an end-entity certificate that includes an ExtendedKeyUsage extension. As we saw from the ASN.1 definition earlier, ExtendedKeyUsage is defined as being one or more KeyPurposeIds, where a KeyPurposeId is an OBJECT IDENTIFIER. RFC 5280 currently provides the following definitions for KeyPurposeId:

```
anyExtendedKeyUsage OBJECT IDENTIFIER ::= { id-ce-extKeyUsage 0 }

id-pkix  OBJECT IDENTIFIER  ::=
           { iso(1) identified-organization(3) dod(6) internet(1)
                   security(5) mechanisms(5) pkix(7) }

id-kp OBJECT IDENTIFIER ::= { id-pkix 3 }

id-kp-serverAuth             OBJECT IDENTIFIER ::= { id-kp 1 }

id-kp-clientAuth             OBJECT IDENTIFIER ::= { id-kp 2 }

id-kp-codeSigning            OBJECT IDENTIFIER ::= { id-kp 3 }

id-kp-emailProtection        OBJECT IDENTIFIER ::= { id-kp 4 }

id-kp-timeStamping           OBJECT IDENTIFIER ::= { id-kp 8 }

id-kp-OCSPSigning            OBJECT IDENTIFIER ::= { id-kp 9 }
```

Sometimes you need to define KeyUsage and ExtendedKeyUsage together. RFC 5280, Section 4.2.1.12. also provides details about which bits in the KeyUsage extension are regarded as compatible.

You will also find KeyPurposeIds defined outside of RFC 5280, possibly going as far as finding proprietary definitions with specific meaning in an application you are working with.

The following example creates an end-entity certificate including an ExtendedKeyUsage extension.

```java
/**
 * Create a special purpose end entity cert which is associated with a
 * particular key purpose.
 *
 * @param signerCert certificate carrying the public key that will later
 *                   be used to verify this certificate's signature.
 * @param signerKey private key used to generate the signature in the
 *                  certificate.
 * @param sigAlg the signature algorithm to sign the certificate with.
 * @param certKey public key to be installed in the certificate.
 * @param keyPurpose the specific KeyPurposeId to associate with this
 *                   certificate's public key.
 * @return an X509CertificateHolder containing the V3 certificate.
 */
public static X509CertificateHolder createSpecialPurposeEndEntity(
        X509CertificateHolder signerCert, PrivateKey signerKey,
        String sigAlg, PublicKey certKey, KeyPurposeId keyPurpose)
        throws OperatorCreationException, CertIOException,
               GeneralSecurityException
{
    X500NameBuilder x500NameBld = new X500NameBuilder(BCStyle.INSTANCE)
            .addRDN(BCStyle.C, "AU")
            .addRDN(BCStyle.ST, "Victoria")
            .addRDN(BCStyle.L, "Melbourne")
            .addRDN(BCStyle.O, "The Legion of the Bouncy Castle")
            .addRDN(BCStyle.CN, "Demo End-Entity Certificate");

    X500Name subject = x500NameBld.build();

    X509v3CertificateBuilder   certBldr = new JcaX509v3CertificateBuilder(
            signerCert.getSubject(),
            calculateSerialNumber(),
            calculateDate(0),
            calculateDate(24 * 31),
            subject,
            certKey);

    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();

    certBldr.addExtension(Extension.authorityKeyIdentifier,
            false, extUtils.createAuthorityKeyIdentifier(signerCert))
            .addExtension(Extension.subjectKeyIdentifier,
                    false, extUtils.createSubjectKeyIdentifier(certKey))
            .addExtension(Extension.basicConstraints,
                    true, new BasicConstraints(false))
            .addExtension(Extension.extendedKeyUsage,
                          true, new ExtendedKeyUsage(keyPurpose));

    ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
                                   .setProvider("BC").build(signerKey);

    return certBldr.build(signer);
}
```

As we can see, the certificate is defined as an end-entity certificate by the BasicConstraints extension that has been provided and has the ExtendedKeyUsage extension also included.

Given the `PrivateKey` for the CA `caPrivKey` and its associated certificate `caCert` we could call the method above to a certificate specifically for signing timestamps as follows:

```
    KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");
    KeyPair specEEKp = kpGen.generateKeyPair();

    X509CertificateHolder specEECert =
            createSpecialPurposeEndEntity(caCert, caPrivKey,
                                           "SHA256withECDSA",
                                           specEEKp.getPublic(),
                                           KeyPurposeId.id_kp_timeStamping);
```

Note the example sets critical on the ExtendedKeyUsage extension to true. Had our code fragment above used `KeyPurposeId.anyExtendedKeyUsage` as the key purpose for the extension, the critical value would need to be changed to false.

Attribute Certificates

The most common profile for X.509 Attribute Certificates is described in RFC 5755 [77]. They differ from the certificates we looked at earlier in that they do not have a public key attached to them.

Instead they allow attributes such as those that specify group membership, role, security clearance, and other authorization information to be associated with another entity, possibly an existing public key certificate. For most purposes, where the attribute certificate is associated with a public key certificate, it can help to think of attribute certificates like a visa, and the original public key certificate as a passport.

This property of attribute certificates makes them quite useful, rather than everyone having their own CA issuing X.509 certificates as digital identities, individual groups can issue attribute certificates which are associated with an already confirmed digital identity signed by someone else. In addition, granting someone short term access to something, for example, is possible without affecting the public key certificate when the access expires.

The structure of attribute certificates is defined as follows:

```
AttributeCertificate ::= SEQUENCE {
  acinfo               AttributeCertificateInfo,
  signatureAlgorithm   AlgorithmIdentifier,
  signatureValue       BIT STRING
}

AttributeCertificateInfo ::= SEQUENCE {
  version                 AttCertVersion, -- version is v2
  holder                  Holder,
  issuer                  AttCertIssuer,
  signature               AlgorithmIdentifier,
  serialNumber            CertificateSerialNumber,
  attrCertValidityPeriod  AttCertValidityPeriod,
  attributes              SEQUENCE OF Attribute,
  issuerUniqueID          UniqueIdentifier OPTIONAL,
  extensions              Extensions OPTIONAL
}

AttCertVersion ::= INTEGER { v2(1) }


AttCertValidityPeriod  ::= SEQUENCE {
  notBeforeTime  GeneralizedTime,
  notAfterTime   GeneralizedTime
}
```

You can see how the basic structure of the outer layer follows the same pattern as that of an X.509 certificate, where the `AttributeCertificateInfo` is the local equivalent of the `TBSCertificate` structure used in X.509. Even the `AttributeCertificateInfo` has several fields which are, at the end of the day, the same as the `TBSCerficate`.

There are two big differences though. Rather than a subject, the AttributeCertificateInfo structure has a holder, which describes how to match the public key certificate the attribute certificate was issued for. Secondly, rather than a public key, the AttributeCertificateInfo structure has an attributes field which contains the additional attributes that the issuer wished to make available to the holder of the public key certificate the attribute certificate was issued for.

We can see how the Holder structure allows for things other than public key certificates to be used by looking at its definition, which is given in RFC 5755 as follows:

```
Holder ::= SEQUENCE {
  baseCertificateID   [0] IssuerSerial OPTIONAL,
      -- the issuer and serial number of
      -- the holder's Public Key Certificate
  entityName          [1] GeneralNames OPTIONAL,
      -- the name of the claimant or role
  objectDigestInfo    [2] ObjectDigestInfo OPTIONAL
      -- used to directly authenticate the holder,
      -- for example, an executable
}

ObjectDigestInfo ::= SEQUENCE {
  digestedObjectType  ENUMERATED {
    publicKey            (0),
    publicKeyCert        (1),
    otherObjectTypes     (2) },
  -- otherObjectTypes MUST NOT
  -- be used in this profile
  otherObjectTypeID   OBJECT IDENTIFIER OPTIONAL,
  digestAlgorithm     AlgorithmIdentifier,
  objectDigest        BIT STRING
}
```

It is worth noting that the standard allows for three methods of matching the holder to be defined and you can see from the ObjectDigestInfo structure it can be used for public key certificates as well as other objects. Originally it was expected that the structure would lend extra flexibility to interpreters of the Holder structure, however it turned out that the flexibility created opportunities for ambiguity as well and in order to remove the risks of confusion RFC 5755 specifies that **only one** option in the Holder sequence should be present.

The AttCertIssuer structure was also designed with flexibility in mind. A look at the definition provided in RFC 5755 also reveals, that as with the Holder, for the purposes of the RFC it is better to have a well defined subset of fields populated in the structure.

```
AttCertIssuer ::= CHOICE {
  v1Form   GeneralNames,  -- MUST NOT be used in this
                          -- profile
  v2Form   [0] V2Form     -- v2 only
}

V2Form ::= SEQUENCE {
  issuerName           GeneralNames  OPTIONAL,
  baseCertificateID     [0] IssuerSerial  OPTIONAL,
  objectDigestInfo      [1] ObjectDigestInfo  OPTIONAL
    -- issuerName MUST be present in this profile
    -- baseCertificateID and objectDigestInfo MUST NOT
    -- be present in this profile
}

IssuerSerial  ::=  SEQUENCE {
  issuer         GeneralNames,
  serial         CertificateSerialNumber,
  issuerUID      UniqueIdentifier OPTIONAL
}
```

As the `UniqueIdentifier` is deprecated in the X.509 profile given by RFC 5280, it would be very unlikely you would see the value populated in the `IssuerSerial` structure as well.

The last element in the attribute certificate structure we will look at is the `Attribute` structure that makes the sequence of attributes that the certificate is produced to carry.

```
Attribute ::= SEQUENCE {
  type      AttributeType,
  values    SET OF AttributeValue
    -- at least one value is required
}

AttributeType ::= OBJECT IDENTIFIER

AttributeValue ::= ANY DEFINED BY AttributeType
```

As you can see the Attribute structure is fairly general in terms of what it can contain.

Building an Attribute Certificate

The following example shows the construction of a simple attribute certificate that just includes a URI for a user's role using the `RoleSyntax` defined in RFC 5755.

```
public static X509AttributeCertificateHolder createAttributeCertificate(
        X509CertificateHolder issuerCert, PrivateKey issuerKey, String sigAlg,
        X509CertificateHolder holderCert, String holderRoleUri)
        throws OperatorCreationException
{
    X509v2AttributeCertificateBuilder acBldr =
            new X509v2AttributeCertificateBuilder(
                  new AttributeCertificateHolder(holderCert),
                  new AttributeCertificateIssuer(issuerCert.getSubject()),
                  calculateSerialNumber(),
                  calculateDate(0),
                  calculateDate(24 * 7));

    GeneralName roleName = new GeneralName(
                     GeneralName.uniformResourceIdentifier, holderRoleUri);

    acBldr.addAttribute(
             X509AttributeIdentifiers.id_at_role, new RoleSyntax(roleName));

    ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
            .setProvider("BC").build(issuerKey);

    return acBldr.build(signer);
}
```

The builder follows a similar pattern as we have already seen for certificates, the only substantive variation is the construction and assignment of the attributes the final certificate is meant to carry. In practice the attributes might even include the use of proprietary OIDs as well as proprietary ASN.1 structures.

Invoking the example method could look as follows:

```
X509AttributeCertificateHolder attrCert =
        createAttributeCertificate(issuerCert,
                issuerSigningKey,
                "SHA256withECDSA", holderCert, "id://DAU123456789");
```

We would just need to be sure the signing algorithm was appropriate for the issuer key and that the role, or what ever other detail might be passed to build attributes from, was appropriate for the attribute certificate we were trying to construct.

Summary

In this chapter we have looked at one of the fundamental building blocks of secure and authenticated communication, X.509 Certificates.

We look at how public keys and identities are attached using the X.509 certificate structure, and how to create a certificate chain.

Finally this chapter considers how to use Attribute Certificates with existing public key certificates.

Chapter 9: Certificate Revocation and Certificate Paths

Occasionally it is necessary for an issuer to withdraw its signature from a certificate. This process is also called certificate revocation and there are both static and on-line methods for supporting this.

Also occasionally, a party issued with a certificate may try to use it for something they should not. In order to prevent this happening there are algorithms which have been standardised for doing certificate path validation. These algorithms can be used to check the provenance of all the certificates involved in issuing an end-entity certificate we have been given and also to confirm that the end-entity certificate is "fit for purpose".

We will look at the process of "changing one's mind" first.

Certificate Revocation Lists (CRLs)

In a perfect world, certificates would be issued and then stay valid till they reached their expiry date. Unfortunately, this is not always the case so a mechanism exists for establishing whether or not a certificate issuer has since changed their mind about the validity of a particular certificate.

Certificate Revocation Lists, or CRLs for short, were the original solution proposed for certificate issuers to be able to keep their users up to date as to what the state of the certificates that had been issued was. As with certificates, the definition of a CRL is provided in RFC 5280 [72] and the top level structure, the `CertificateList` is defined as follows:

```
CertificateList  ::=  SEQUENCE  {
      tbsCertList          TBSCertList,
      signatureAlgorithm   AlgorithmIdentifier,
      signatureValue       BIT STRING  }

 TBSCertList  ::=  SEQUENCE  {
      version                 Version OPTIONAL,
                                   -- if present, MUST be v2
      signature               AlgorithmIdentifier,
      issuer                  Name,
      thisUpdate              Time,
      nextUpdate              Time OPTIONAL,
      revokedCertificates     SEQUENCE OF SEQUENCE  {
           userCertificate         CertificateSerialNumber,
           revocationDate          Time,
           crlEntryExtensions      Extensions OPTIONAL
                                    -- if present, version MUST be v2
                                }  OPTIONAL,
      crlExtensions           [0]  EXPLICIT Extensions OPTIONAL
                                    -- if present, version MUST be v2
                                }
```

As you can see from the structure, the intention is that CRLs take their authority from the person who signs them. It contains some issuer details, a list of revocations, details about when the CRL was issued (thisUpdate), possibly a time at which an update should be requested (nextUpdate), and some extensions.

Common CRL Extensions

Like Certificates, CRLs can also have extensions which are used to convey extra information about how to interpret the CRL. The list following is not exhaustive and should just be taken as a guide. If you need to interpret these extensions with a good depth of understanding it is worth making sure you read the relevant sections of RFC 5280 [72].

Authority Key Identifier

The authority key identifier extension provides a means for identifying the public key that can be used for verifying the signature on the CRL. CRL issuers conforming to RFC 5280 [72] are required to include the authority key identifier extension.

The extension has the same syntax as the equivalent extension for a certificate.

Issuer Alternative Name

The issuer alternative name extension is a non-critical extension which allows additional identities to be associated with the CRL. The extension has the same syntax as the equivalent extension for a certificate.

CRL Number

The CRL number is also non-critical and is used to provider a unique identifier indicating whether this CRL supersedes another one. CRL issuers conforming to RFC 5280 [72] are required to include the CRL number extension.

The CRL Number extension is identified by the constant Extension.basicConstraints which has the OID value "2.5.29.20" and is made up of a single integer. The actual definition looks like:

```
id-ce-cRLNumber OBJECT IDENTIFIER ::= { id-ce 20 }

CRLNumber ::= INTEGER (0..MAX)
```

A conformant CRL issuer should be increasing the CRLNumber value with each new CRL issued, so that a CRL with a higher number always represents a more recent revision of the CRL. It is important to take note of the fact that where the CRL is a delta CRL the value CRLNumber must share the same numbering sequence as the CRLs the delta is for.

Delta CRL Indicator

The Delta CRL indicator shows that a CRL should be treated as a list of changes from an earlier CRL, the idea being that if a Delta CRL and the earlier full CRL are combined the result would represent what would have been a full CRL at the time the Delta CRL was issued.

The Delta CRL indicator has the OID value “2.5.29.27” and the extension has the following syntax:

```
id-ce-deltaCRLIndicator OBJECT IDENTIFIER ::= { id-ce 27 }

BaseCRLNumber ::= CRLNumber
```

where the BaseCRLNumber is the CRL Number of the earlier full CRL that the Delta CRL shows changes from.

The Delta CRL Indicator should be marked as a critical extension.

Common CRL Entry Extensions

As you can see from the definition of `TBSCertList` the `revokedCertificates` field also has space for optional extensions. RFC 5280 currently defines three extensions for a CRL entry. It should be noted that pre-RFC 5280 there was a HoldInstructionCode (OID value “2.5.29.23”) as well, but it has since been removed.

CRLReason

The CRLReason extension has the OID value “2.5.29.21” and the extension has the following syntax:

```
id-ce-cRLReasons OBJECT IDENTIFIER ::= { id-ce 21 }

-- reasonCode ::= { CRLReason }

CRLReason ::= ENUMERATED {
    unspecified             (0),
    keyCompromise           (1),
    cACompromise            (2),
    affiliationChanged      (3),
    superseded              (4),
    cessationOfOperation    (5),
    certificateHold         (6),
         -- value 7 is not used
    removeFromCRL           (8),
    privilegeWithdrawn      (9),
    aACompromise           (10) }
```

As the name suggests the CRLReason extension indicates the reason that the certificate wound up in the CRL in the first place. If the CRLReason extension is not present you should assume a reason code of 0, as in `unspecified`. RFC 5280 also maintains that the extension should be absent if the reason would be `unspecified` and only used if some other reason needs to be indicated.

InvalidityDate

The InvalidityDate extension is used to provide an exact date at which it is known, or suspected, that a certificate became invalid. This extension removes the need to back date a CRL entry when the actually time of invalidity is before the revocation date. RFC 5280 recommends including this extension whenever such information is available.

The InvalidityDate extension has the OID value "2.5.29.24" and the extension has the following syntax:

```
id-ce-invalidityDate OBJECT IDENTIFIER ::= { id-ce 24 }

InvalidityDate ::=  GeneralizedTime
```

Details about how GeneralizedTime is constructed can be found in Appendix A: "ASN.1 and Bouncy Castle".

CertificateIssuer

The CertificateIssuer extension is used to indicate who the real issuer of the certificate was. There is a bit of twist to this one when you find it though. RFC 5280, section 5.3.3, states that where this extension is not present, the certificate issuer defaults to that of the previous entry. Or put another way, once a

CertificateIssuer extension shows up, every subsequent CRL entry should be assumed to be associated with the new real issuer until the extension shows up again.

The CertificateIssuer extension has the OID value “2.5.29.29” and the extension has the following syntax:

```
id-ce-certificateIssuer   OBJECT IDENTIFIER ::= { id-ce 29 }

CertificateIssuer ::=     GeneralNames
```

Owing to what affect the extension’s presence has on interpreting the CRL, RFC 5280 states the extension must be marked as critical if used.

Creating a New CRL

The following code creates a CRL containing a single revocation.

```java
/**
 * Create a CRL containing a single revocation.
 *
 * @param caKey the private key signing the CRL.
 * @param sigAlg the signature algorithm to sign the CRL with.
 * @param caCert the certificate associated with the key signing the CRL.
 * @param certToRevoke the certificate to be revoked.
 * @return an X509CRLHolder representing the revocation list for the CA.
 */
public X509CRLHolder createCRL(
        PrivateKey caKey,
        String sigAlg,
        X509CertificateHolder caCert,
        X509CertificateHolder certToRevoke)
        throws IOException, GeneralSecurityException, OperatorCreationException
{
    X509v2CRLBuilder crlGen = new X509v2CRLBuilder(caCert.getSubject(),
            calculateDate(0));

    crlGen.setNextUpdate(calculateDate(24 * 7));

    // add revocation
    ExtensionsGenerator extGen = new ExtensionsGenerator();

    CRLReason crlReason = CRLReason.lookup(CRLReason.privilegeWithdrawn);

    extGen.addExtension(Extension.reasonCode, false, crlReason);


    crlGen.addCRLEntry(certToRevoke.getSerialNumber(),
                                            new Date(), extGen.generate());

    // add extensions to CRL
    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();

    crlGen.addExtension(Extension.authorityKeyIdentifier, false,
            extUtils.createAuthorityKeyIdentifier(caCert));

    ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
                                    .setProvider("BC").build(caKey);

    return crlGen.build(signer);
}
```

Our first step is creating a builder for the CRL, the X509v2CRLBuilder class, in this case the setNextUpdate() method is also called specifying that the CRL has a lifetime of 7 days. After that we add a revocation, providing a CRL entry with a CRLReason extension on it.

When there are no revoked certificates, the revoked certificates list must be absent from the CRL structure.

Converting an X509CRLHolder to an X509CRL

The org.bouncycastle.cert.jcajce.JcaX509CRLConverter class is provided to convert X509CRLHolder objects into regular JCA X509CRL objects. The JcaX509CRLConverter class can be used to generate a X509CRL according to the current provider priority in the JVM, or you can specify a particular provider. In this example we have specified the Bouncy Castle provider by using its name.

```
X509CRLHolder crlHldr = ...

X509CRL crl = new JcaX509CRLConverter()
                                      .setProvider("BC")
                                      .getCRL(crlHldr);
```

To go back the other way, simply pass the encoding of the X509CRL to the X509CRLHolder, or pass the X509CRL to a JcaX509CRLHolder which will do the encoding step for you.

```
X509CRL crl = ...
X509CRLHolder crlHldr = new JcaX509CRLHolder(crl);
```

The `CertificateFactory` class has two methods associated with producing public key certificates: `generateCRL()` and `generateCRLs()`. These methods can be used to read in various encodings of certificates, usually including PEM and BER/DER, and then produce either a single certificate (in the case of `generateCRL()`) or a collection of certificates (in the case of `generateCRLs()`).

The `getEncoded()` method of `X509CertificateHolder` produces a BER encoding of the X.509 certificate wrapped in the class. You can use this method to also provide input for the `generateCertificate()` method on the `CertificateFactory` in the following way:

```
/**
 * Simple method to convert an X509CRLHolder to an X509CRL
 * using the java.security.cert.CertificateFactory class.
 */
public static X509CRL convertX509CRLHolder(
                                         X509CertificateHolder crlHolder)
        throws GeneralSecurityException, IOException
{
    CertificateFactory cFact = CertificateFactory.getInstance("X.509", "BC");

    return (X509CRL)cFact.generateCRL(
                                       new ByteArrayInputStream(
                                               crlHolder.getEncoded()));
}
```

Creating an Updated CRL

As you have no doubt realised, CRLs grow larger over time. Without some API help, creating an updated CRL could be very painful. The following example shows how to take an existing CRL and update it.

```
/**
 * Create an updated CRL from a previous one and add a new revocation.
 *
 * @param caKey the private key signing the CRL.
 * @param sigAlg the signature algorithm to sign the CRL with.
 * @param caCert the certificate associated with the key signing the CRL.
 * @param previousCaCRL the previous CRL for this CA.
 * @param certToRevoke the certificate to be revoked.
 * @return an X509CRLHolder representing the updated revocation list for the
 * CA.
```

```
     */
    public X509CRLHolder updateCRL(
            PrivateKey caKey,
            String sigAlg,
            X509CertificateHolder caCert,
            X509CRLHolder previousCaCRL,
            X509CertificateHolder certToRevoke)
            throws IOException, GeneralSecurityException, OperatorCreationException
    {
        X509v2CRLBuilder crlGen = new X509v2CRLBuilder(caCert.getIssuer(),
                calculateDate(0));

        crlGen.setNextUpdate(calculateDate(24 * 7));

        // add new revocation
        ExtensionsGenerator extGen = new ExtensionsGenerator();

        CRLReason crlReason = CRLReason.lookup(CRLReason.privilegeWithdrawn);

        extGen.addExtension(Extension.reasonCode, false, crlReason);

        crlGen.addCRLEntry(certToRevoke.getSerialNumber(),
                                            new Date(), extGen.generate());

        // add previous revocations
        crlGen.addCRL(previousCaCRL);

        // add extensions to CRL
        JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();

        crlGen.addExtension(Extension.authorityKeyIdentifier, false,
                extUtils.createAuthorityKeyIdentifier(caCert));

        ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
                                    .setProvider("BC").build(caKey);

        return crlGen.build(signer);
    }
```

You can see that this is very similar to creation of a CRL from scratch, with one major difference, the line that says

```
        crlGen.addCRL(previousCaCRL);
```

which adds all the entries contained in `previousCaCRL` to the CRL builder we are currently using.

Obtaining Revocation Information from a Certificate Issuer

CRLs are made available by certificate issuers in two ways. The first is directly, in which case a URL is normally provided that the physical CRL can be downloaded from. The second is indirectly, usually via a service such as Online Certificate Status Protocol (OCSP) which we will look at next. The information for how to do this is published somewhere, often in the certificates themselves via a certificate extension, such as the CRL Distribution Point.

From a local point of view we should make sure we have a safe method of obtaining safe CRL information even if it does mean rejecting some "out of the wild" certificates without being sure they are invalid. The reason for this is it is important to check with reasonable probability of a correct answer that a CA has not revoked a certificate[15].

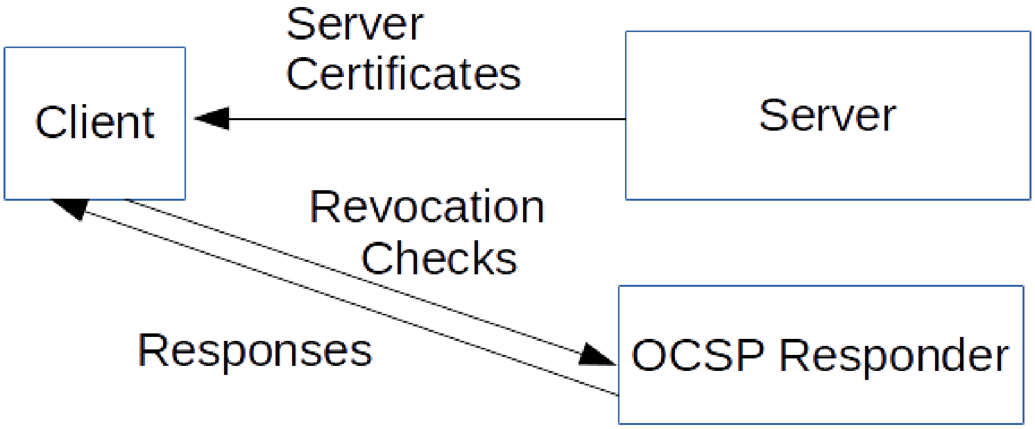
We want to make sure of this as the alternative is to use whatever source the certificate we are checking suggests. Of course, as this is determined by the certificate issuer, it is unlikely to result in a fake certificate being accepted, but if the certificate issuer does not mean us well, who knows what trouble we might cause ourselves.

Online Certificate Status Protocol

There are a couple of obvious problems with CRLs if you expect to have a large turnover in certificates. Constantly issuing updates (especially with a narrow time window) and making sure that all parties of interest have received them can rapidly become unmanageable. Online Certificate Status Protocol (OCSP), which is currently defined in RFC 6960 [86] was developed to look after this problem.

[15] There is an approach called 'soft failure', where if a certificate cannot be checked (say due to an OCSP server being down), that the certificate is considered valid. At the end of the day whether you want to do this is determined by what you know about the issuer and other risks. However, it is worth keeping in mind that some people have compared the usefulness of "soft failure" to the usefulness a seat belt specifically designed to fail only when you have a crash.

The Basic Protocol



OCSP - Server presents certificates, an OCSP responder responds with assurances of validity to the client

OCSP is described in terms of clients sending requests and then the CAs responsible for satisfying those requests replying via responders - the servers at addresses indicated by the CA as being responsible for managing a CA's OCSP requests. Responses from a CA are signed and can have a lifetime associated with them - in the same way as a CRL has a "next update" associated with it. Object identifiers associated with OCSP are generally off the arc, or branch, `id-pkix-ocsp` which is defined as "1.3.6.1.5.5.7.48.1".

In the context of Bouncy Castle, both OCSP request and response generation is supported. The high level classes for dealing with OCSP live under the `org.bouncycastle.ocsp` package in the bcpkix distribution. The low-level classes making up the ASN.1 protocol elements can be found in `org.bouncycastle.asn1.ocsp`. The API follows the ASN.1 structures that make up the protocol in both cases though, with the higher level one providing the ability to operate with JCA classes and provide signing and verification functionality. As this is the case we will look at the API in the context of the ASN.1 structures first, before moving on to an example of usage.

Requests

The core of the OCSP request structure is the `Request` sequence, which contains a `CertID` and optional extensions, and the `CertID` which identifies the certificates that the status enquiry is about. The ASN.1 for these two structures is defined below.

```
Request ::=     SEQUENCE {
    reqCert                    CertID,
    singleRequestExtensions    [0] EXPLICIT Extensions
                                       { {re-ocsp-service-locator,
                                              ...}} OPTIONAL }

CertID ::= SEQUENCE {
    hashAlgorithm            AlgorithmIdentifier
                                 {DIGEST-ALGORITHM, {...}},
    issuerNameHash     OCTET STRING, -- Hash of issuer's DN
    issuerKeyHash      OCTET STRING, -- Hash of issuer's public key
    serialNumber       CertificateSerialNumber }
```

The BC API uses the Req class to represent the Request sequence and uses CertificateID to represent the CertID. The Req class is what you would expect, although you generally would not create one directly as you would use an OCSPReqBuilder to build a full request object and the builder will do the creation for you.

The CertificateID, on the other hand, requires some work, as it needs to be able to calculate hashes. SHA-1 is still heavily used for this purpose so unless you are told otherwise, creating a CertificateID to check the status for a certificate with the serial number serialNumber which has been issued by issuer certificate issuerCert will look something like the following:

```
DigestCalculatorProvider digCalcProv =
    new JcaDigestCalculatorProviderBuilder().setProvider("BC").build();

CertificateID   id = new CertificateID(
    digCalcProv.get(CertificateID.HASH_SHA1), issuerCert, serialNumber);
```

Where the DigestCalculator passed to the CertificateID constructor is, as shown, for SHA-1.

The carrier structure for the request information is the OCSPRequest, represented in the BC API by the OCSPReq class. An OCSPRequest can be signed if necessary so the ASN.1 definition follows a similar pattern to what we saw with certificates.

The ASN.1 definition for a full OCSPRequest is as follows:

```
OCSPRequest     ::=     SEQUENCE {
    tbsRequest                  TBSRequest,
    optionalSignature   [0]     EXPLICIT Signature OPTIONAL }

TBSRequest      ::=     SEQUENCE {
    version             [0] EXPLICIT Version DEFAULT v1,
    requestorName       [1] EXPLICIT GeneralName OPTIONAL,
    requestList             SEQUENCE OF Request,
    requestExtensions   [2] EXPLICIT Extensions {{re-ocsp-nonce |
                     re-ocsp-response, ...,
                     re-ocsp-preferred-signature-algorithms}} OPTIONAL }

Signature       ::=     SEQUENCE {
    signatureAlgorithm   AlgorithmIdentifier
                             { SIGNATURE-ALGORITHM, {...}},
    signature            BIT STRING,
    certs            [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL }

Version  ::=  INTEGER  {  v1(0) }
```

The only time you would expect to create a BC `OCSPReq` object directly is as a responder where you are creating it from an encoded `OCSPRequest` structure that someone has sent you. In the case of a client you can build an `OCSPReq` using the `OCSPReqBuilder` class.

Responses

Responders reply with an `OCSPResponse` structure. In the Bouncy Castle OCSP API the `OCSPResponse` structure is represented by the `OCSPResp` class. The ASN.1 description for this is as follows:

```
OCSPResponse ::= SEQUENCE {
  responseStatus         OCSPResponseStatus,
  responseBytes          [0] EXPLICIT ResponseBytes OPTIONAL }

OCSPResponseStatus ::= ENUMERATED {
   successful            (0),  -- Response has valid confirmations
   malformedRequest      (1),  -- Illegal confirmation request
   internalError         (2),  -- Internal error in issuer
   tryLater              (3),  -- Try again later
                               -- (4) is not used
   sigRequired           (5),  -- Must sign the request
   unauthorized          (6)   -- Request unauthorized
}
```

The important thing to note with the above structure is that the response status does not say anything about the validity of any of the certificates you are enquiring about! You need to dig deeper to get information on that.

As you can see, the structure is designed with flexibility in mind and the `ResponseBytes` structure follows a similar pattern to what you will see in a lot of ASN.1 structures, where an OBJECT IDENTIFIER is included to allow for the identification of the payload it carries - in this case embedded in an OCTET STRING.

```
ResponseBytes ::= SEQUENCE {
   responseType            OBJECT IDENTIFIER,
   response                OCTET STRING }
```

The standard response defined in RFC 6960 is the `BasicOCSPResponse` defined by an object identifier called `id-pkix-ocsp-basic` which is given as `{ id-pkix-ocsp 1 }`.

```
BasicOCSPResponse ::= SEQUENCE {
  tbsResponseData          ResponseData,
  signatureAlgorithm       AlgorithmIdentifier,
  signature                BIT STRING,
  certs                [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL }

ResponseData ::= SEQUENCE {
   version             [0] EXPLICIT Version DEFAULT v1,
   responderID             ResponderID,
   producedAt              GeneralizedTime,
   responses               SEQUENCE OF SingleResponse,
   responseExtensions  [1] EXPLICIT Extensions OPTIONAL }
```

You can get access to the payload in an `OCSPResp` object by calling the `getResponseObject()` method. If the internal structure identifies itself as carrying an OCTET STRING associated with id-pkix-ocsp-basic the `getResponseObject()` will create a `BasicOCSPResp` object for you.

As you can see, the `BasicOCSPResponse` allows for signing and as the `responses` field is a SEQUENCE, it can carry a batch of responses to be included in a single response. There is also a `certs` field which can be used to carry certificates that may help the OCSP client verify the signature associated with the `BasicOCSPResponse`.

In the Bouncy Castle API the `BasicOCSPResponse` structure is represented by the `BasicOCSPResp` class. The `BasicOCSPResp` class has methods on it allowing access to most things in the `BasicOCSPResponse` structure, with the two most important methods being the `isSignatureValid()` method which allows verification of the signature associated with `BasicOCSPResponse` and the `getResponses()` method which returns an array of `SingleResp` classes. A special extended key usage `id-kp-OCSPSigning` ({`id-kp 9`}) can be assigned to a certificate which is authorised for verifying OCSP signatures.

The `SingleResp` classes act as wrappers around `SingleResponse` structures. The `SingleResponse` structure has the following syntax:

```
SingleResponse ::= SEQUENCE {
   certID                  CertID,
   certStatus              CertStatus,
   thisUpdate              GeneralizedTime,
   nextUpdate          [0] EXPLICIT GeneralizedTime OPTIONAL,
   singleExtensions    [1] EXPLICIT Extensions OPTIONAL }
```

As you can see, this is a bit like a "mini-CRL" for a single certificate. The `certStatus` field, returned by the `getCertStatus()` method on a `SingleResp` object is the actual certificate status and will tell us whether the certificate is GOOD (represented by a `null`) or has an issue. The `nextUpdate` field is accessed using the `getNextUpdate()` method and can be used if the OCSP response is to be cached (so for OCSP stapling purposes). Note: as the `nextUpdate` is optional, it may be absent in which case the `getNextUpdate()` method will return null.

Normally you would only build an `OCSPResp` directly in a client where you were converting an encoded response. In a responder you would usually build an `OCSPResp` using an `OCSPRespBuilder`.

Extensions

Looking at the ASN.1, both OCSP requests and responses can carry extensions. There are a number listed in RFC 6960, but the two most common ones are detailed here.

Nonce

The `nonce` extension is used to match a response to a request and to prevent replay attacks[16]. As the name of the extension suggests, the value of the `nonce` extension should be something that the client will only use once within a reasonable period. The ASN.1 definition and labeling for it is as follows:

```
id-pkix-ocsp-nonce     OBJECT IDENTIFIER ::= { id-pkix-ocsp 2 }

Nonce ::= OCTET STRING
```

As the value is specified as an OCTET STRING, it can be anything a client wishes to use that meets the "risk of reuse" criteria. When used the extension is added by the client to the request and then echoed back by the responder in the response.

[16] If there is no nonce in a response, an attacker can simply replay a previously captured request. This can result in great embarrassment if it turns out the certificate has since been revoked.

Acceptable Response Types

OCSP allows for a variety of response types, other than the `BasicOCSPResponse`. Where a responder is able to respond with multiple response types, it is a good idea for the client to lock down the response types the responder might choose from.

The extension to make this possible is the `AcceptableResponses` extension, defined and labeled as follows:

```
id-pkix-ocsp-response  OBJECT IDENTIFIER ::= { id-pkix-ocsp 4 }

AcceptableResponses ::= SEQUENCE OF OBJECT IDENTIFIER
```

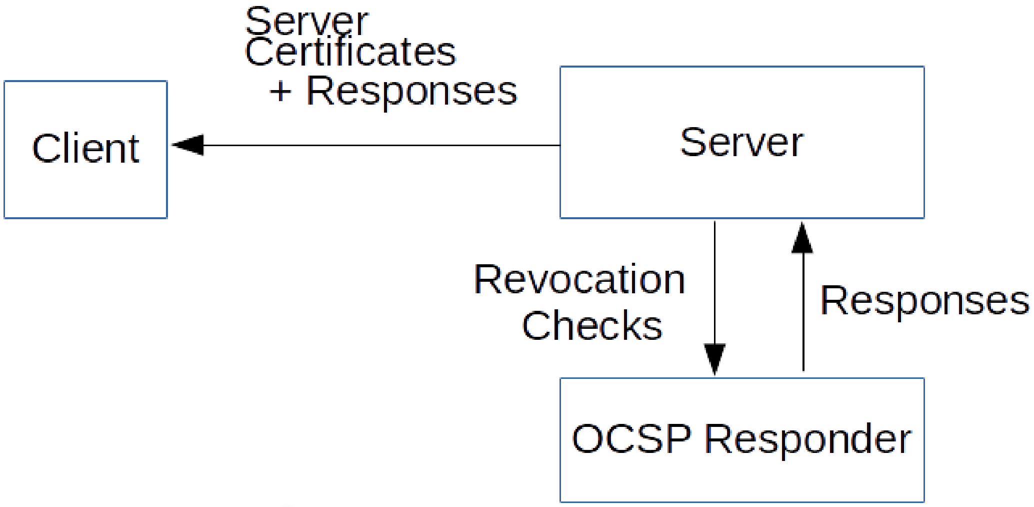
RFC 6960 specifies that a responder must be able to provide responses containing `BasicOCSPResponse`. If you need to tell a responder that your client can only handle the `BasicOCSPResponse`, you can create the extension as:

```
Extension ext =
    new Extension(OCSPObjectIdentifiers.id_pkix_ocsp_response, true,
       new DERSequence(OCSPObjectIdentifiers.id_pkix_ocsp_basic).getEncoded());
```

and then add it to your request using the `OCSPReqBuilder` class, which we will look at below.

OCSP Stapling: Fixing an Issue with OCSP

Before we look at how the Bouncy Castle APIs support OCSP it is worth being aware that while OCSP fixes most of the issues around the use of CRLs, it does create a few problems of its own. CAs who make OCSP responders available and also issue plenty of high issue certificates are going to see an awful lot of requests, furthermore a “side channel” is also created as the end of the connection doing the OCSP checking is broadcasting to the CA that issued the certificate that it is talking to the party that certificate was issued to. This can be a particularly serious problem for a protocol like TLS, where the client browsers will find themselves broadcasting their traffic habits to CAs - possibly not what everyone would regard as a positive feature.



OCSP Stapling - Server sends short lived OCSP responses as well as its certificates

This issue can be addressed using a procedure known as OCSP stapling, which in the case of TLS, where the concept originated, is known as the Multiple Certificate Status Request Extension and described in RFC 6961 [87]. In this case the TLS server collects signed OCSP responses and sends them with its certificate path. The client is then able to verify the signatures on the OCSP responses (which must be signed by the certificates' issuers) and can then be confident that the certificates are valid - at least within the bounds of the validity time of the OCSP responses.

OCSP stapling works as the server is responsible for collecting the responses, so the client's identity is protected. Also, while the OCSP responses have a small lifetime, they are not single use, so the server can reuse the responses until new ones are required - taking load off the CA's OCSP responder. If you are using OCSP in a system of your own design, you might want to think about this.

Generating OCSP Requests

This example shows a method generating an OCSP request to query the validity of a certificate with issuer `issuerCert` and serial number `serialNumber`. The request is also generated to include a nonce extension.

```
/**
 * Generation of an OCSP request concerning certificate serialNumber from
 * issuer represented by issuerCert with a nonce extension.
 *
 * @param issuerCert certificate of issuer of certificate we want to check.
 * @param serialNumber serial number of the certificate we want to check.
 * @return an OCSP request.
 */
public static OCSPReq generateOCSPRequest(
                    X509CertificateHolder issuerCert, BigInteger serialNumber)
        throws OCSPException, OperatorCreationException
    {
        DigestCalculatorProvider digCalcProv =
            new JcaDigestCalculatorProviderBuilder().setProvider("BC").build();

        // Generate the id for the certificate we are looking for
        CertificateID   id = new CertificateID(
            digCalcProv.get(CertificateID.HASH_SHA1), issuerCert, serialNumber);

        // Basic request generation with nonce
        OCSPReqBuilder bldr = new OCSPReqBuilder();

        bldr.addRequest(id);

        // Create details for nonce extension - example only!
        BigInteger nonce = BigInteger.valueOf(System.currentTimeMillis());

        bldr.setRequestExtensions(new Extensions(
            new Extension(OCSPObjectIdentifiers.id_pkix_ocsp_nonce,
                             true, new DEROctetString(nonce.toByteArray()))));

        return bldr.build();
    }
```

The code follows three steps, the `CertificateID` is constructed so we can include the correct information for the certificate we wish to check in the request. Next we create an `OCSPReqBuilder` which we add the `CertificateID` to. Finally we add the nonce extension to the builder and then make the completed request.

Generating OCSP Responses

Putting together an OCSP response requires the use of a `BasicOCSPRespBuilder` and an `OCSPRespBuilder` class. We need the two builders as OCSP allows for response types other than the `BasicOCSPResponse` to be included in the `OCSPResponse`.

The example method that follows has been written to allow the method to generate both passes and fails for certificates passed to it according to the final argument - `revokedSerialNumber`.

```
/**
 * Generation of an OCSP response based on a single revoked certificate.
 *
 * @param request the OCSP request we are asked to check.
 * @param responderKey signing key for the responder.
 * @param pubKey public key for responder.
 * @param revokedSerialNumber the serial number that we regard as revoked.
 * @return an OCSP response.
 */
public static OCSPResp generateOCSPResponse(
    OCSPReq request,
    PrivateKey responderKey, SubjectPublicKeyInfo pubKey,
    BigInteger revokedSerialNumber)
    throws OCSPException, OperatorCreationException
{
    DigestCalculatorProvider digCalcProv =
         new JcaDigestCalculatorProviderBuilder().setProvider("BC").build();

    BasicOCSPRespBuilder basicRespBldr = new BasicOCSPRespBuilder(
                                 pubKey,
                                 digCalcProv.get(CertificateID.HASH_SHA1));

    Extension ext = request.getExtension(
                              OCSPObjectIdentifiers.id_pkix_ocsp_nonce);

    if (ext != null)
    {
        basicRespBldr.setResponseExtensions(new Extensions(ext));
    }

    Req[] requests = request.getRequestList();

    for (int i = 0; i != requests.length; i++)
    {
        CertificateID certID = requests[i].getCertID();

        // this would normally be a lot more general!
        if (certID.getSerialNumber().equals(revokedSerialNumber))
        {
            basicRespBldr.addResponse(certID,
                new RevokedStatus(new Date(), CRLReason.privilegeWithdrawn));
        }
        else
        {
            basicRespBldr.addResponse(certID, CertificateStatus.GOOD);
        }
    }

    ContentSigner signer = new JcaContentSignerBuilder("SHA256WithECDSA")
                                    .setProvider("BC").build(responderKey);

    BasicOCSPResp basicResp = basicRespBldr.build(signer, null, new Date());

    OCSPRespBuilder respBldr = new OCSPRespBuilder();

    return respBldr.build(OCSPRespBuilder.SUCCESSFUL, basicResp);
}
```

Following through the code we see we need to set up the `BasicOCSPRespBuilder` first, adding the nonce from the request if it is present. The processing step is to see if any of the `CertificateID` objects in the request match the revoked serial number we have been given. Next we build the `BasicOCSPResp` signing it as we do (we pass a `null` for the certificate chain as we assume the client knows how to verify the message already). Finally we create the `OCSPResp` object that contains the `OCSPResponse` indicating a successful status as we were able to process the request fully.

Putting the Request Generator and Responder Together

The next two code examples are a general status reporter to give some idea of how you might want to interpret a response with a main driver to do some initial setup.

In the following code fragment we generate a request using the `generateOCSPRequest()` method we defined earlier and then process the result using the `generateOCSPResponse()` method we defined in the previous section.

```
/**
 * Check a certificate against a revoked serial number by using an
 * OCSP request and response.
 *
 * @param caPrivKey the issuer private key.
 * @param caCert the issuer certificate.
 * @param revokedSerialNumber a serial number the responder is to
 *                             treat as revoked.
 * @param certToCheck the certificate to generate the OCSP request for.
 * @return a status message for certToCheck
 */
public static String getStatusMessage(
    PrivateKey caPrivKey, X509CertificateHolder caCert,
          BigInteger revokedSerialNumber, X509CertificateHolder certToCheck)
    throws Exception
{
    OCSPReq request = generateOCSPRequest(
                                    caCert, certToCheck.getSerialNumber());

    OCSPResp response = generateOCSPResponse(
        request,
        caPrivKey, caCert.getSubjectPublicKeyInfo(),
        revokedSerialNumber);

    BasicOCSPResp   basicResponse =
                        (BasicOCSPResp)response.getResponseObject();

    ContentVerifierProvider verifier =
          new JcaContentVerifierProviderBuilder()
                .setProvider("BC").build(caCert.getSubjectPublicKeyInfo());

    // verify the response
    if (basicResponse.isSignatureValid(verifier))
    {
        SingleResp[]      responses = basicResponse.getResponses();

        Extension reqNonceExt = request.getExtension(
                         OCSPObjectIdentifiers.id_pkix_ocsp_nonce);
        byte[] reqNonce = reqNonceExt.getEncoded();
        Extension respNonceExt = basicResponse.getExtension(
                     OCSPObjectIdentifiers.id_pkix_ocsp_nonce);

        // validate the nonce if it is present
        if (respNonceExt != null
            && Arrays.equals(reqNonce, respNonceExt.getEncoded()))
        {
            String message = "";
            for (int i = 0; i != responses.length; i++)
            {
                message += " certificate number "
                    + responses[i].getCertID().getSerialNumber();
                if (responses[i].getCertStatus()
                    == CertificateStatus.GOOD)
                {
                    return message + " status: good";
                }
                else
                {
                    return message + " status: revoked";
                }
            }

            return message;
        }
        else
        {
            return "response nonce failed to validate";
        }
    }
    else
    {
        return "response failed to verify";
    }
}
```

This code fragment just does some initial setup to create the keys and certificates needed for the `getStatusMessage()` method to work and then calls it.

```
public static void main(
    String[] args)
    throws Exception
{
    KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");

    KeyPair caKp = kpGen.generateKeyPair();

    X509CertificateHolder caCert =
                       createTrustAnchor(caKp, "SHA256withECDSA");

    KeyPair certKp = kpGen.generateKeyPair();

    X509CertificateHolder certOfInterest = createIntermediateCertificate(
                                    caCert,
                                    caKp.getPrivate(),"SHA256withECDSA",
                                    certKp.getPublic(), 0);

    System.out.println(
        getStatusMessage(
            caKp.getPrivate(), caCert,
            certOfInterest.getSerialNumber().add(BigInteger.ONE),
            certOfInterest));
}
```

If you run the `main()` as is, you should get back the cerificate's serial number followed by "status: good". You can also cause `getStatusMessage()` to return "status: revoked" by changing `certOfInterest.getSerialNumber().` to `certOfInterest.getSerialNumber()`.

Note that while the example has concentrated on the most general OCSP classes in the API, there are also JCA specific OCSP classes in `org.bouncycastle.ocsp.jcajce` such as `JcaCertificateID`, `JcaBasicOCSPRespBuilder`, and `JcaRespID` which allow the use of `PublicKey` and `X509Certificate` classes.

Certificate Path Validation

When we talk about a certificate path we are talking about the certificates involved in getting us from a particular trust anchor, which we accept at face value, to a certificate we are trying to validate, usually an end-entity certificate. This path is really a collection of smaller paths, representing subsets of the full certificate path as it extends further and further from the trust anchor. Sometimes a certificate path is also referred to as a certificate chain.

We have already touched on two aspects of a certificate's validity: is the certificate signed by a certificate authority we trust, and are we sure that the authority has not withdrawn their signature by issuing a CRL. Other than the trust anchor (usually self signed), all certificates in a path need to satisfy these criteria. The other thing that we need to check is whether everyone, from the trust anchor down, who issued certificates on the path, meant for a given certificate they issued to be used in the manner the certificate is now being used.

For example, we have already seen that the Basic Constraints extension can be used to determine if a certificate can be used as a CA certificate or it is an end-entity certificate and represents the end of a path. Now imagine you are presented with a certificate path where the end-entity certificate is signed by a certificate where the Basic Constraints extension does not assert that the signing certificate is a CA certificate. In this case the path is clearly invalid as the issuer of the certificate that signed the end-entity we were presented with did not sign the signing certificate with the idea in mind that it would be used for signing other certificates.

Other than for the example above, path validation can be, or at least appear to be, horrendously complex. The most common algorithm for path validation is the IETF one presented in RFC 5280 [72], but it is important to keep in mind that both governments and corporates have specified variations on the algorithm presented in the RFC. These variations are referred to as profiles, and before embarking on validating certificates for a particular organisation it is important to make sure you are aware of what profile you are doing the validation for.

Certificate Path Validation in the JCA

Java has its own API for the validation of certificate paths, the CertPath API. This involves a few classes which are created via `getInstance()` methods as the internals of them may vary from provider to provider. There are also some parameter classes for configuring the certificate path analyser as well as a class for carrying the result of an analysis and a class for carrying a certificate path itself.

The CertPath API

All the classes below are packaged in the `java.security.cert` package. We will see how these classes can be used in the examples, but the following should give you a general idea of what you are looking at when you are reading the code.

CertPath
: `CertPath` is a carrier class for certificate chains, or paths. Objects of this class are created using one of the `CertificateFactory.generateCertPath()` methods, they are not created directly.

CertStore
: `CertStore` is a general store for certificates and CRLS. They are created via `CertStore.getInstance()` and require a `CertStoreParameters` object as part of the `getInstance()` method.

CertStoreParameters
: `CertStoreParameters` has two subclasses `CollectionCertStoreParameters` and `LDAPCertStoreParameters`. These classes exist to make certificates and CRLs available to `CertStores`. You can also specify your own if you wish to implement the underlying CertStore to support it.

CertSelector
: The `CertSelector` is a general interface to provide a mechanism for matching certificates. You will usually use the `X509CertSelector` class when you need one of these.

CertPathBuilder
: `CertPathBuilder` is a support class for building validated certificate paths. These are also provider based and use the `getInstance()` factory pattern for construction.

CertPathValidator
: `CertPathValidator` is a support class for validating an existing certificate path. The `CertPathValidator` is also provider based and created using the `getInstance()` factory pattern.

CRLSelector
: The `CRLSelector` is a general interface to provide a mechanism for matching CRLs. Usually you will use the mutable class `X509CRLSelector` when you need one of these.

PKIXParameters
: `PKIXParameters` is the primary class for passing configuration to either a certificate path validator or, through its subclass `PKIXBuilderParameters`, a certificate path builder.

PKIXBuilderParameters
: `PKIXBuilderParameters` is an extension of the `PKIXParameters` class. The class allows you to define a maximum path length that is acceptable and also to provide details about the end-entity certificate you are hoping to validate, or build a path to.

PKIXCertPathValidatorResult

If a certificate path validation fails you will be greeted with an exception. In the event that a validation passes, the result will come back in a `PKIXCertPathValidatorResult`.

PKIXCertPathChecker

The `PKIXCertPathChecker` class provides a means of customising certificate path validation by allowing a developer to introduce their own checks.

PKIXRevocationChecker

The `PKIXRevocationChecker` class is an extension class of PKIXCertPathChecker that appeared in Java 8. The class provides additional support for customising the use of CRLs and OCSP beyond what is usually allowed.

TrustAnchor

`TrustAnchor` provides a mechanism for giving a certificate path builder, or validator, the public key or its containing certificate that you are willing to accept at face value to form the basis for a certificate path.

X509CertSelector

`X509CertSelector` is a mutable class that can be configured to match particular certificates. In the case of the certificate path builder class, it is used to specify the certificate that should represent the end-entity.

X509CRLSelector

`X509CRLSelector` is also a mutable class and can be configured to match particular CRLs.

A Basic Example

The following example shows the process of validation for a simple certificate path based on a trust anchor with a single CA certificate above the end-entity certificate using the Java CertPath API.

```java
/**
 * Basic example of certificate path validation using a CertPathValidator.
 */
public class JcaCertPathExample
{
    public static void main(String[] args)
        throws Exception
    {
        //
        // set up our sample data
        //
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");
        JcaX509CertificateConverter certConverter =
                                   new JcaX509CertificateConverter().setProvider("BC");
        JcaX509CRLConverter crlConverter =
                                           new JcaX509CRLConverter().setProvider("BC");

        KeyPair trustKp = kpGen.generateKeyPair();

        X509CertificateHolder trustHldr =
                           createTrustAnchor(trustKp, "SHA256withECDSA");
        X509Certificate trustCert = certConverter.getCertificate(trustHldr);

        KeyPair caKp = kpGen.generateKeyPair();

        X509CertificateHolder caHldr = createIntermediateCertificate(
                                        trustHldr,
                                        trustKp.getPrivate(),
                                        "SHA256withECDSA", caKp.getPublic(), 0);
        X509Certificate caCert = certConverter.getCertificate(caHldr);

        KeyPair eeKp = kpGen.generateKeyPair();

        X509Certificate eeCert = certConverter.getCertificate(
            createEndEntity(
                caHldr, caKp.getPrivate(), "SHA256withECDSA", eeKp.getPublic()));

        X509CRL trustCRL = crlConverter.getCRL(
            createEmptyCRL(trustKp.getPrivate(), "SHA256withECDSA", trustHldr));
        X509CRL caCRL = crlConverter.getCRL(
            createEmptyCRL(caKp.getPrivate(), "SHA256withECDSA", caHldr));

        //
        // create our sample certificate path
        //
        CertificateFactory certFact = CertificateFactory.getInstance(
                                               "X.509", "BC");

        List<X509Certificate> chain = new ArrayList<X509Certificate>();

        chain.add(caCert);
        chain.add(eeCert);

        CertPath certPath = certFact.generateCertPath(chain);

        //
        // construct the CertStore containing our certificate set.
        //
        List certStoreList = new ArrayList();
        certStoreList.add(trustCRL);
        certStoreList.add(caCert);
        certStoreList.add(caCRL);
        certStoreList.add(eeCert);

        CertStoreParameters params =
            new CollectionCertStoreParameters(certStoreList);

        CertStore certStore = CertStore.getInstance("Collection", params, "BC");

        //
        // create the set of valid trust anchors
        //
        Set<TrustAnchor> trust = new HashSet<TrustAnchor>();
        trust.add(new TrustAnchor(trustCert, null));

        //
        // set up the parameters for the path validation
        //
        PKIXParameters param = new PKIXParameters(trust);

        X509CertSelector certSelector = new X509CertSelector();
        certSelector.setCertificate(eeCert);

        param.setTargetCertConstraints(certSelector);
        param.addCertStore(certStore);
        param.setRevocationEnabled(true);

        try
        {
            //
            // Create a validator and perform the validation.
            //
            CertPathValidator validator =
                                CertPathValidator.getInstance("PKIX", "BC");

            PKIXCertPathValidatorResult result =
                (PKIXCertPathValidatorResult)validator.validate(certPath, param);

            System.out.println("validated: " + result.getPublicKey());
        }
        catch (CertPathValidatorException e)
        {
            System.out.println("validation failed: index ("
                + e.getIndex() + "), reason \"" + e.getMessage() + "\"");
        }
    }
}
```

Customizing a CertPath Validation

The PKIXCertPathChecker class provides the ability to introduce your own processing into a path validation process. The classes need to be added at configuration time and are added to a validation process by passing them to the PKIXParameters.addCertPathChecker() method.

There are a couple of reasons you may want to use this class. You might be working with a validation profile that has introduced its own critical extensions in which case you will need implementations based on the PKIXCertPathChecker to make sure the extensions are handled properly. Alternately, you might be using a path validator that relies on CRLs for doing revocation and you want to handle revocation checking differently, say by using OCSP.

The PKIXCertPathChecker Class

The PKIXCertPathChecker is abstract and has four methods on it which are expected to be provided by subclasses. The main thing to be aware of is that the order in which certificates are presented to the checker is determined by the writer of the path validator. A validator which does forward checking presents the certificates starting at the end-entity and working up the chain towards the trust anchor. Reverse checking simply presents certificates in the opposite direction, with the end-entity being the last one to be presented.

The four methods to be implemented are:

init()

As the name implies, the init() method initialises the state of the checker. The single boolean argument to it indicates whether the checker is being initialised for forward checking (true) or reverse checking (false).

isForwardCheckingSupported()

This method should return true if forward checking is supported. Note: the JCA expects all checkers to support reverse processing.

getSupportedExtensions()

getSupportedExtensions() should return a Set of OIDs as String objects, representing the specific extensions the checker handles. If the checker has some purpose other than extension handling it should return null.

check()

The check() method is where the work is done. It takes two parameters, one representing the certificate to be checked, and one representing the set of critical OIDs the validator has not been able to process. The OIDs are in a `Set` of `String`, and the `Set` is mutable, the idea being that you should remove any OIDs from it that your checker has handled so signaling that that is the case to the validator that invoked the checker.

The following example shows a simple OCSP checker, based on the PKIXCertPathChecker. Prior to Java 1.8, and also with providers that support older JVMs, this was the only approach for supporting “roll your own” OCSP implementations.

```
/**
 * A basic path checker that does an OCSP check for a single CA
 */
public class OCSPPathChecker
    extends PKIXCertPathChecker
{
    private KeyPair         responderPair;
    private X509Certificate caCert;
    private BigInteger      revokedSerialNumber;

    public OCSPPathChecker(
        KeyPair         responderPair,
        X509Certificate caCert,
        BigInteger      revokedSerialNumber)
    {
        this.responderPair = responderPair;
        this.caCert = caCert;
        this.revokedSerialNumber = revokedSerialNumber;
    }

    public void init(boolean forwardChecking)
        throws CertPathValidatorException
    {
        // ignore
    }

    public boolean isForwardCheckingSupported()
    {
        return true;
    }

    public Set getSupportedExtensions()
    {
        return null;
    }

    public void check(Certificate cert, Collection extensions)
        throws CertPathValidatorException
    {
        try
        {
            X509CertificateHolder issuerCert =
                                    new JcaX509CertificateHolder(caCert);
            X509CertificateHolder certToCheck =
                        new JcaX509CertificateHolder((X509Certificate)cert);

            if (certToCheck.getIssuer().equals(issuerCert.getSubject()))
            {
                String message = OCSPProcessingExample.getStatusMessage(
                    responderPair.getPrivate(),
                    issuerCert,
                    revokedSerialNumber, certToCheck);

                if (message.endsWith("good"))
                {
                    System.out.println(message);
                }
                else
                {
                    throw new CertPathValidatorException(message);
                }
            }
        }
        catch (Exception e)
        {
            throw new CertPathValidatorException(
                           "exception verifying certificate: " + e, e);
        }
    }
}
```

Note that the `isForwardChecking()` method returns `true`, and that the 'getSupportedExtensions()' returns `null`. In the case of the `OCSPPatchChecker` the checker is ambivalent to the order in which certificates are presented, but as it is being used for revocation checking it does not handle any extensions, so `getSupportedExtensions()` returns `null` and no change is made to the `extensions` `Set` passed to the `match()` method.

In the following example we apply the `OCSPPatchChecker` to a path validation. You can see where the path checker is added to the `PKIXParameters` object. Note also that

`PKIXParameters.setRevocationEnabled()` is called with `false` as otherwise the path validator will expect to be able to find CRLs for the trust anchor and the CA certificate.

```java
/**
 * Basic example of certificate path validation using a PKIXCertPathChecker with
 * the checker being used for checking revocation status.
 */
public class JcaCertPathWithCheckerExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");
        JcaX509CertificateConverter certConverter =
                            new JcaX509CertificateConverter().setProvider("BC");

        KeyPair trustKp = kpGen.generateKeyPair();

        X509CertificateHolder trustHldr =
                            createTrustAnchor(trustKp, "SHA256withECDSA");
        X509Certificate trustCert = certConverter.getCertificate(trustHldr);

        KeyPair caKp = kpGen.generateKeyPair();

        X509CertificateHolder caHldr = createIntermediateCertificate(
                                        trustHldr,
                                        trustKp.getPrivate(),
                                        "SHA256withECDSA", caKp.getPublic(), 0);
        X509Certificate caCert = certConverter.getCertificate(caHldr);

        KeyPair eeKp = kpGen.generateKeyPair();

        X509Certificate eeCert = certConverter.getCertificate(
            createEndEntity(
                caHldr, caKp.getPrivate(), "SHA256withECDSA", eeKp.getPublic()));

        List certStoreList = new ArrayList();

        certStoreList.add(caCert);
        certStoreList.add(eeCert);

        CertStoreParameters params =
            new CollectionCertStoreParameters(certStoreList);

        CertStore certStore = CertStore.getInstance("Collection", params, "BC");

        Set<TrustAnchor> trust = new HashSet<TrustAnchor>();
        trust.add(new TrustAnchor(trustCert, null));

        CertPathValidator validator =
                            CertPathValidator.getInstance("PKIX", "BC");

        PKIXParameters param = new PKIXParameters(trust);

        X509CertSelector certSelector = new X509CertSelector();
        certSelector.setCertificate(eeCert);

        param.setTargetCertConstraints(certSelector);
        param.addCertStore(certStore);
        param.setRevocationEnabled(false);
        param.addCertPathChecker(
            new OCSPPathChecker(trustKp, trustCert,
                                caCert.getSerialNumber().add(BigInteger.ONE)));
        param.addCertPathChecker(
            new OCSPPathChecker(caKp, caCert,
                                eeCert.getSerialNumber().add(BigInteger.ONE)));

        CertificateFactory certFact = CertificateFactory.getInstance(
                                               "X.509", "BC");

        List<X509Certificate> chain = new ArrayList<X509Certificate>();

        chain.add(caCert);
        chain.add(eeCert);

        CertPath certPath = certFact.generateCertPath(chain);

        try
        {
            PKIXCertPathValidatorResult result =
                (PKIXCertPathValidatorResult)validator.validate(certPath, param);

            System.out.println("validated: " + result.getPublicKey());
        }
        catch (CertPathValidatorException e)
        {
            System.out.println("validation failed: index ("
                + e.getIndex() + "), reason \"" + e.getMessage() + "\"");
        }
    }
}
```

This example will initially run and validate the path as neither of the certificates being checked (the CA certificate and the end-entity certificate) will be regarded as revoked. As before, if you remove the `.add(BigInteger.ONE)` you will be able to see what the example does when it detects a revocation.

The PKIXRevocationChecker Class

The `PKIXRevocationChecker` is an extension of the `PKIXCertPathChecker` that was added in Java 1.8. If one of these is detected in the cert path checker list in the `PKIXParameters` object passed to a validator, it will also ignore the setting of `PKIXParameters.setRevocationEnabled()` and always be used. `PKIXRevocationChecker` implentations are expected to be able to make use of CRLs and OCSP.

Using the CertPathBuilder

The final class we will look at in the CertPath API is the `CertPathBuilder`. This class is also provider dependent as it makes use of a validator internally, so they are created using `CertPathBuilder.getInstance()` methods rather than simple construction.

The `CertPathBuilder` is very useful. Often protocols will allow for certificates to be sent along with signed data or encrypted data and the certificates are generally included in ASN.1 Sets, which are not ordered. The originators of the sets may also include extra certificates and CRLs that are not specifically relevant to the task at hand as well. You may find yourself with a signature to validate or with an end-entity certificate and be wondering if the certificate really is the one you should be using for encrypting a session key with, and a collection of “random” certificates to determine the answer from. In situations like this, you need to make use of the `CertPathBuilder`.

The basic method behind the `CertPathBuilder` is expressed in the constructor to the `PKIXBuilderParameters` class, which takes trust anchors, and a `CertSelector` that acts as the target constraints for the end-entity. Once other settings are configured in the `PKIXBuilderParameters` object, the job of the `CertPathBuilder` is to find an end-entity that matches the target constraints and construct a valid path back to an accepted trust anchor.

Here is an example of the use of the `CertPathBuilder`. As you can see, other than the use of the `PKIXBuilderParameters` what happens is very similar to what you would see with a `CertPathValidator`.

```
/**
 * Basic example of certificate path validation using a PKIXCertPathChecker with
 * the checker being used for checking revocation status.
 */
public class JcaCertPathWithCheckerExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");
        JcaX509CertificateConverter certConverter =
                            new JcaX509CertificateConverter().setProvider("BC");

        KeyPair trustKp = kpGen.generateKeyPair();

        X509CertificateHolder trustHldr =
                            createTrustAnchor(trustKp, "SHA256withECDSA");
        X509Certificate trustCert = certConverter.getCertificate(trustHldr);

        KeyPair caKp = kpGen.generateKeyPair();

        X509CertificateHolder caHldr = createIntermediateCertificate(
                                        trustHldr,
                                        trustKp.getPrivate(),
                                        "SHA256withECDSA", caKp.getPublic(), 0);
        X509Certificate caCert = certConverter.getCertificate(caHldr);

        KeyPair eeKp = kpGen.generateKeyPair();

        X509Certificate eeCert = certConverter.getCertificate(
            createEndEntity(
                caHldr, caKp.getPrivate(), "SHA256withECDSA", eeKp.getPublic()));

        List certStoreList = new ArrayList();

        certStoreList.add(caCert);
        certStoreList.add(eeCert);

        CertStoreParameters params =
            new CollectionCertStoreParameters(certStoreList);

        CertStore certStore = CertStore.getInstance("Collection", params, "BC");
        Set<TrustAnchor> trust = new HashSet<TrustAnchor>();
        trust.add(new TrustAnchor(trustCert, null));

        CertPathValidator validator =
                            CertPathValidator.getInstance("PKIX", "BC");

        PKIXParameters param = new PKIXParameters(trust);

        X509CertSelector certSelector = new X509CertSelector();
        certSelector.setCertificate(eeCert);

        param.setTargetCertConstraints(certSelector);
        param.addCertStore(certStore);
        param.setRevocationEnabled(false);
        param.addCertPathChecker(
            new OCSPPathChecker(trustKp, trustCert,
                                caCert.getSerialNumber().add(BigInteger.ONE)));
        param.addCertPathChecker(
            new OCSPPathChecker(caKp, caCert,
                                eeCert.getSerialNumber().add(BigInteger.ONE)));

        CertificateFactory certFact = CertificateFactory.getInstance(
                                               "X.509", "BC");

        List<X509Certificate> chain = new ArrayList<X509Certificate>();

        chain.add(caCert);
        chain.add(eeCert);

        CertPath certPath = certFact.generateCertPath(chain);

        try
        {
            PKIXCertPathValidatorResult result =
                (PKIXCertPathValidatorResult)validator.validate(certPath, param);

            System.out.println("validated: " + result.getPublicKey());
        }
        catch (CertPathValidatorException e)
        {
            System.out.println("validation failed: index ("
                + e.getIndex() + "), reason \"" + e.getMessage() + "\"");
```

```
    }
  }
}
```

If you run the example you should find that the path has been built up correctly and it will output the certificates, starting from the end-entity leading back to the trust anchor. If all has worked well, this should look something like the following (truncated for page width):

```
CN=Demo End-Entity Certificate, O=The Legion of the Bouncy Castle, L=Melb...
CN=Demo Intermediate Certificate, O=The Legion of the Bouncy Castle, L=Melb...
CN=Demo Root Certificate, O=The Legion of the Bouncy Castle, L=Melb...
```

It is important to pay attention to the choice of target constraints when configuring a `CertPathBuilder`. You can even configure the builder to look for a specific certificate by using:

```
    endConstraints.setCertificate(eeCert);
```

rather than relying on the issuer and serial number as the example does at the moment. Incorrect constraints could result in a certificate path other than the one you need being returned, and ambiguous ones will result in an exception being thrown.

Summary

In this chapter we have continued our investigation of Certificates from Chapter 7, and in this case, what to do when we wish to invalidate an existing certificate and how we might determine that any certificate we need to validate has a traceable path back to something we implicitly trust.

To invalidate certificates, we looked at Certificate Revocation Lists, their attributes and extensions. This covered the creation of a CRL using X509v2CRLBuilder, and how to update it at any point in time in the future. A secondary approach for dealing with Certificate Invalidation was using the Online Certificate Status Protocol (OCSP) and an example using OCSPReqBuilder to create requests.

Finally this chapter covered how to examine a Certificate Path for validity, by checking that the chain from the trust anchor, down to the end-entity certificate was valid. An example of this approach using the CertPathValidator showed the steps required to fully validate certificates.

Chapter 10: Key and Certificate Storage

The JCA provides a general purpose keystore class which allows for file based storage of keys and, sometimes, acts as a front end for hardware based keys as well. In this chapter we will just look at the file based types. Java supports 3 different keystore types “out of the box” and Bouncy Castle adds a few more. Several of the original keystore types in both Java and Bouncy Castle are now regarded as deprecated as they have not been able to keep up with changes in the protection requirements for security. This has also affected the way the Java keytool is used - with the older keystore types, certificates and the like were generally stored in the clear, so it was possible to do some processing of a keystore without providing a password. This is no longer the case.

Setup for the Examples

Before we start looking at the keystore types it would be useful to have something to store in them. In this case that will require both a private key and a an associated certificate for the public key. The regular Java VM has a class for linking these together already, the `X500PrivateCredential`, which was added in Java 1.5. We have created our own version here as the class is not as widely spread across the different Java profiles as you might expect (for example, at the time of writing, it is missing from Android). Here is a definition for a simple carrier class for a private key and its associated certificate.

```java
/**
 * Carrier class for a private key and its corresponding public key certificate.
 * <p>
 * The regular Java API also has javax.security.auth.x500.X500PrivateCredential,
 * this class is a basic replacement for that. There is also a slightly more
 * general class in the BC PKIX API - org.bouncycastle.pkix.PKIXIdentity.
 * </p>
 */
public class PrivateCredential
{
    private final X509Certificate certificate;
    private final PrivateKey privateKey;

    /**
     * Base constructor.
     *
     * @param certificate the public key certificate matching privateKey.
     * @param privateKey the private key matching the certificate parameter.
     */
    public PrivateCredential(X509Certificate certificate, PrivateKey privateKey)
    {
        this.certificate = certificate;
        this.privateKey = privateKey;
    }

    public PrivateKey getPrivateKey()
    {
        return privateKey;
    }

    public X509Certificate getCertificate()
    {
        return certificate;
    }
}
```

Having a carrier class defined, we will use the following method for generating a private key and an associated certificate which is self-signed.

```
  /**
   * Create a private key with an associated self-signed certificate
   * returning them wrapped in an X500PrivateCredential
   *
   * Note: We use generateECKeyPair() from chapter6.EcDsaUtils and
   * createTrustAnchor() from chapter8.JcaX509Certificate.
   *
   * @return an X500PrivateCredential containing the key and its certificate.
   */
  public static PrivateCredential createSelfSignedCredentials()
      throws GeneralSecurityException, OperatorCreationException
  {
      JcaX509CertificateConverter certConverter =
                         new JcaX509CertificateConverter().setProvider("BC");

      KeyPair selfSignedKp = generateECKeyPair();

      X509CertificateHolder selfSignedHldr =
                         createTrustAnchor(selfSignedKp, "SHA256withECDSA");

      X509Certificate selfSignedCert = certConverter.getCertificate(selfSignedHldr);

```

```
        return new PrivateCredential(selfSignedCert, selfSignedKp.getPrivate());
    }
```

We will make use of the `createSelfSignedCredentials()` method to provide a key and certificate test data to store in the various keystores we look at.

Early KeyStore Types: JKS, JCEKS, BKS, and UBER

The JKS, JCEKS, BKS, and UBER formats are all early formats provided by the JCA, in the case of JKS and JCEKS, or Bouncy Castle in the case of BKS and UBER. They are now starting to fall out of use for storing private/secret keys, partly as there are more secure formats now available. The JKS and BKS keystores are still commonly used for public key certificates.

The JKS format is the original keystore format for the JCA. The format provides some level of tamper protection and private keys have some encryption associated with them using a proprietary stream cipher based on SHA-1. The keystore cannot store secret keys either. A file with a JKS keystore usually has the extension ".jks" if it has one at all. It is possible to examine most of the contents of a JKS keystore using the key tool without knowing the keystore password, although it is not possible to recover a private key without a password.

The JCEKS format was introduced with the JCE. This store format can also store secret keys and in this case Triple-DES is used to protect private and secret keys. A file with a JCEKS keystore usually has the extension ".jceks" when one is provided. It is also possible to examine most of the contents of a JCEKS keystore using the key tool without knowing the keystore password, although it is not possible to recover a protected secret or private key without a password.

The BKS format is a similar style format to JCEKS. It is possible to examine most of the contents of a BKS keystore using the key tool without knowing the keystore password, although it is not possible to recover a private key without a password. BKS can also store secret and private keys and uses Triple-DES to protect them. Passwords are converted into keys in BKS using the PKCS#12 PBKDF.

The UBER format is based on BKS, but unlike BKS stores the keystore as an encrypted blob, protected using the Twofish encryption algorithm. This means that the keystore cannot be accessed using the Java keytool without using a password.

The following example code shows how to generate a basic JKS keystore with a single private key and a self-signed certificate.

```
/**
 * Basic example of using JKS to store a single private key and self-signed
 * certificate.
 */
public class JKSExample
{
    public static void main(String[] args)
        throws Exception
    {
        PrivateCredential cred = createSelfSignedCredentials();

        KeyStore store = KeyStore.getInstance("JKS");

        store.load(null, null);

        store.setKeyEntry("key", cred.getPrivateKey(), "keyPass".toCharArray(),
            new Certificate[] { cred.getCertificate() });

        FileOutputStream fOut = new FileOutputStream("basic.jks");

        store.store(fOut, "storePass".toCharArray());

        fOut.close();
    }
}
```

Note that the `store.load(null, null);` performs the task of initialising the keystore.

If you run the example it will create a keystore file called “basic.jks” in the JKS format. We will use this file in the next section to look at some of the basic commands with the keytool.

The keytool Command

The keytool is a Java utility which provides for command line processing of keystores. By default it works with JKS keystores, so for a basic viewing, or display of an already existing keystore, you just need to make use of the `-list` option, as well as provide the location of the keystore, using `-keystore`, and the store password, using `-storepass`. If the store password is not passed as an option, the keytool command will prompt for it.

Examining a Keystore

Assuming you wanted to examine the JKS keystore we created in the last section using the keytool, you could use the keytool as follows:

```
keytool -list -keystore basic.jks -storepass storePass
```

Running this command should show something like the following output.

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

key, <dd/mm/yyyy>, PrivateKeyEntry,
Certificate fingerprint (SHA1): <a SHA-1 fingerprint>
```

Where `<dd/mm/yyyy>` is the date at which the entry was created in day/month/year format.

The fingerprint will look something like this:

```
ED:44:9F:92:5B:DA:C8:DD:34:9A:7A:A5:B3:8D:04:D7:CE:6B:69:27
```

although you would expect the numbers to be different as the idea of the SHA-1 fingerprint is to provide a 20 byte value that should uniquely identify the certificate. With more recent JVMs you will also find that the fingerprint has been upgraded to SHA-256, in which case it will contain 32 hex values, with the description showing as:

```
Certificate fingerprint (SHA-256): ...
```

To see what happens when you do not enter a password on the command line, or via the `Enter keystore password` prompt try the following, and just press the `<enter>` key without typing in any characters of the password when you are prompted for the password.

```
keytool -list -keystore basic.jks
```

If you simply hit `<enter>` at the password prompt you should see something like the following:

```
*****************  WARNING WARNING WARNING  *****************
* The integrity of the information stored in your keystore  *
* has NOT been verified!  In order to verify its integrity, *
* you must provide your keystore password.                  *
*****************  WARNING WARNING WARNING  *****************

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

key, <dd/mm/yyyy>, PrivateKeyEntry,
Certificate fingerprint (SHA1): <a SHA-1 fingerprint>
```

Generating a Key Pair

It is also possible to create a keystore with an initial key pair from the command line using the keytool. For example try:

```
keytool -genkey -alias key -keyalg RSA -keystore rsa.jks -keysize 2048
```

the command will prompt for a password as none has been passed on the command line, and after generating the key pair will also prompt for elements for the X.500 DN to be associated with the self-signed certificate that will contain the public key that will be stored with the private key.

You can look at the result by running the list command on the generated keystore:

```
keytool -list -keystore rsa.jks
```

Assuming you are looking at a pre-Java 9 JVM you will see something like the following, showing the private key entry that we generated for the alias key:

```
Enter keystore password:
Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

key, <dd/mm/yyyy>, PrivateKeyEntry,
Certificate fingerprint (SHA1): <a SHA-1 fingerprint>
```

With the arrival of Java 9, the JVM maintainers moved to using PKCS12 as the default file format, rather than JKS. We will have a look at PKCS12 further on in this chapter, but if you run the list command using Java 9 or later, you will see something more like this:

```
Enter keystore password:
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

key, <dd mmm yyyy>, PrivateKeyEntry,
Certificate fingerprint (SHA-256): <a SHA-256 fingerprint>
```

indicating the keystore is a PKCS12 type and showing the creation date with month given in short-name form, rather than digits.

Generating a Certification Request

Having generated a key with a self-signed certificate we often want to get a certificate from a Certification Authority to replace our self-signed certificate with.

```
keytool -certreq -alias key -file csr.txt -keystore rsa.jks
```

The `csr.txt` file is human readable (in a sense). If you look at it you will see a blob of base64 encoded data between lines containing `-----BEGIN NEW CERTIFICATE REQUEST-----` and ending with `-----END NEW CERTIFICATE REQUEST-----`. We will look at these in more detail in Chapter 12, but the quick summary here is that what you are looking at in the file is actually a base64 encoding of a DER encoded PKCS#10 certification request. It contains two things of note: the public key that you generated earlier, and the X.500 name you want associated with it. Both of these are integrity protected using a signature generated using the private key associated with public key.

Importing a Certificate Received From a Certificate Authority

Once you have sent your certification request off to a Certification Authority (CA) you will get back another file containing the actual certificate the CA has signed. You need to load this into the keystore to replace your self signed one. Usually this is a two step process, you need to import the CA's trust anchor, and then you can import the CA's response and associate it with your generated key by doing the import to the same alias you used for creating your key.

In the examples below we have considered one way of doing this: the CA has sent its trust anchor in a file called `ca-trustanchor.pem` and the response to our certification request in a file called `ca-response.pem`.

Our first step is to import the trust anchor as a trusted certificate into the keystore. We do this as follows:

```
keytool -import -alias ca-name -keystore rsa.jks -file ca-trustanchor.pem
```

Where `ca-name` is an alias we have chosen for the CA in question and `ca-trustanchor.pem` is the trust anchor certificate. The keytool command will then print some details about the trust anchor certificate and ask for confirmation to trust it. Assuming all things go well, you should answer `yes` to confirm the certificate is to be trusted.

Next we import the response, but this time the import is done using the same alias we used when we generated the original key - `key`.

```
keytool -import -alias key -keystore rsa.jks -file ca-response.pem
```

When this import is executed the self-signed certificate that was originally generated for our key will be discarded and the CA response certificate in `ca-response.pem` will be installed in place of it. At this point, as far as anyone else relying on the CA you are using is concerned, you should be be able to use your certificate to identify yourself “officially”.

The PKCS12 KeyStore Type

The PKCS12 store type takes its name, and format, from “PKCS #12: Personal Information Exchange Syntax”. Originally PKCS #12 was an RSA controlled standard but it is now under the care of the IETF, with RFC 7292 [94] being the latest version.

Creating a PKCS12 file in code using the Bouncy Castle provider follows roughly the same process as for a JKS file, with one difference - the key does not have a password associated with it. You can see the key password is set to `null` in the example below:

```java
/**
 * Basic example of using JKS to store a single private key and self-signed
 * certificate.
 */
public class PKCS12Example
{
    public static void main(String[] args)
        throws Exception
    {
        PrivateCredential cred = createSelfSignedCredentials();

        KeyStore store = KeyStore.getInstance("PKCS12", "BC");

        store.load(null, null);

        store.setKeyEntry("key", cred.getPrivateKey(), null,
            new Certificate[] { cred.getCertificate() });

        FileOutputStream fOut = new FileOutputStream("basic.p12");

        store.store(fOut, "storePass".toCharArray());

        fOut.close();
    }
}
```

The reason for the absence of a key password is that, traditionally, PKCS12 was designed for personal data, so only a password protecting the keystore was required. Bouncy Castle follows this pattern, as do most other implementations.

You can list the created keystore using the keytool command, although on earlier JVMs you need to specify the keystore type using the `-storetype` option as in:

```
keytool -list -keystore basic.p12 -storetype PKCS12
```

The BCFKS/BCSFKS KeyStore Type

The BCFKS keystore type came about as a result of our first FIPS certification. The issue our users were facing related to the need to store keys on disk, but to be sure that the file format was fully FIPS compliant. This, of course, included using PBKDF2 for password conversion and being able to support a different passwords for individual keys as the keystore itself. PKCS #12, the closest alternative, was ruled out as very few implementations support the use of different key and keystore passwords and the regular PBE algorithm is also not FIPS compliant, nor is the MAC used to protect the keystore.

Keeping that in mind, the following example creates a BCFKS key store.

```
/**
 * Basic example of using BCFKS to store a single private key and self-signed
 * certificate.
 */
public class BCFKSExample
{
    public static void main(String[] args)
        throws Exception
    {
        PrivateCredential cred = createSelfSignedCredentials();

        KeyStore store = KeyStore.getInstance("BCFKS", "BC");

        store.load(null, null);

        store.setKeyEntry("key", cred.getPrivateKey(), "keyPass".toCharArray(),
            new Certificate[] { cred.getCertificate() });

        FileOutputStream fOut = new FileOutputStream("basic.fks");

        store.store(fOut, "storePass".toCharArray());

        fOut.close();
    }
}
```

From a creation point of view, the BCFKS keystore functions in a very similar way to the JKS keystore. Note that the BCFKS format, while allowing for individual passwords to be associated with keys does take a similar approach to PKCS#12 in the sense that it assumes once a process has unlocked a key it should be okay to use it again, i.e. the key password is only checked once. If you are running an application where you want to guarantee isolation of each thread, use the IBCFKS variant which is detailed below. IBCFKS is an immutable wrapper around a BCFKS keystore, so the on-disk storage format is the same but the behaviour when loaded is different.

Customizing BCFKS

As the BCFKS format is fully driven by the algorithm identifiers stored in its data file, it is possible to introduce different algorithms into the format.

Using Scrypt

The first customization that was introduced was the ability to introduce the use of the scrypt algorithm. The scrypt algorithm provides a memory hard wrapper around the FIPS scheme PBKDF2. We had a look at scrypt originally in Chapter 5. In the next example we see how an `ScryptConfig` can be used to initialize a BCFKS keystore so that the keystore is protected using PBKDF2 based on scrypt.

```java
/**
 * Example of using BCFKS to store a single private key and self-signed
 * certificate, using scrypt for the key store protection.
 */
public class BCFKSwithScryptExample
{
    public static void main(String[] args)
        throws Exception
    {
        PrivateCredential cred = createSelfSignedCredentials();

        // specify scrypt parameters
        PBKDFConfig config = new ScryptConfig.Builder(1024, 8, 1)
                                                .withSaltLength(20).build();

        KeyStore store = KeyStore.getInstance("BCFKS", "BC");

        // initialize empty store to use scrypt
        store.load(new BCFKSLoadStoreParameter.Builder()
                                .withStorePBKDFConfig(config)
                                .build());

        store.setKeyEntry("key", cred.getPrivateKey(), "keyPass".toCharArray(),
                            new Certificate[] { cred.getCertificate() });

        FileOutputStream fOut = new FileOutputStream("scrypt.fks");

        store.store(fOut, "storePass".toCharArray());

        fOut.close();
    }
}
```

As we saw in Chapter 5 the idea of scrypt is not only to introduce password hashing and salting to protect the password, but it also introduces memory hardness - in the sense that it consumes a lot of memory in order to prevent the hashing process being readily adaptable to acceleration on things like repurposed graphics accelerators or specialized hardware.

Using IBCFKS

The IBCFKS keystore is designed to be shared around between different threads. It is an immutable wrapper around a regular BCFKS keystore which always requires a password to unlock a key. If you already have a BCFKS keystore which you do not wish to change you can use IBCFKS to prevent the application threads from changing the keystore after loading. For example:

```
KeyStore store = KeyStore.getInstance("IBCFKS", "BC");
FileInputStream fIn = new FileInputStream("basic.fks");

store.load(fIn, "storePass".toCharArray());
```

can be used to load the BCFKS key store we created in the earlier example.

Converting KeyStores Using the keytool

Every now and then you will find youself in possession of a keystore in one format where you will need one in a different format altogether. You can use the keytool operation `-importstore` to carry out this translation. The following example shows how to translate a JKS format file into a BCFKS file:

```
keytool -importkeystore -srckeystore rsa.jks \
                -destkeystore rsa.bcfks -deststoretype BCFKS
```

There is no specification for the source key store in this example as we are using a JKS keystore (the default). If the source key store was of some other type you would need to use `-srcstoretype` to specify the type of the source key store.

One word of caution on translation of keystores. For the most part, where a key store only consists of private keys and their associated certificates, translation should always be trouble free. Not all keystore types support the storage of symmetric secret keys, so if symmetric secret keys are in the the source keystore there may be issues with translation. In this case you may need to split the original keystore out so that you have a source key store which only has private keys and certificates in it.

Summary

In this chapter we have examined the historical formats for storing keys and certificates which are now deprecated for general use. We examine the properties of the modern FIPS compatible key stores and how they can be used within our software.

The chapter primarily uses the 'keytool' for most of the examples, as while key and certificate storage can be done via the BouncyCastle APIs, most of the operations on keystores is via the command line.

Finally we cover the translation of keystores into different formats, again using the 'keytool'. Keep in mind that translation may not work easily if there are symmetric keys stored in the keystore being translated.

Chapter 11: Cryptographic Message Syntax, S/MIME, and TSP

In this chapter we will look at higher level protocols for message processing that make use of the cryptographic primitives we have looked at so far. We will start with Cryptographic Message Syntax, a format which provides the starting point for the other message formats, such as Secure MIME (S/MIME) and Time Stamp Protocol (TSP). In terms of releases, APIs for dealing with these message types mainly appear in jars beginning with the prefix bcpkix, the one exception being the JavaMail based S/MIME API which appears in jars starting with bcmail.

Cryptographic Message Syntax

Cryptographic Message Syntax was originally described in PKCS#7, and for this reason it is often referred to as PKCS#7 as well as by the acronym CMS. In its original form it provided mechanisms for signing, encryption and MAC based authentication of messages. Things have moved on a bit since then and ongoing development to the standard has been taken over by the IETF, with the latest version of CMS described in RFC 5652 [74]. The encoding of CMS is based around ASN.1 and with the exception of structures used as input to signature or MAC algorithms, CMS structures are defined as using the BER encoding rules. The use of BER means that CMS structures are often built using indefinite length encodings and are able to deal with message data which is larger than the available memory on the machine that might be processing it.

CMS structures are all encapsulated in a basic structure, the ContentInfo structure. In RFC 5652, the ContentInfo structure and its component fields are defined as follows:

```
ContentInfo ::= SEQUENCE {
    contentType ContentType,
    content [0] EXPLICIT ANY DEFINED BY contentType }

ContentType ::= OBJECT IDENTIFIER
```

In essence the structure provides just what is required to tag a particular structure with an object identifier. There is even an object identifier for the ContentInfo structure itself:

```
id-ct-contentInfo OBJECT IDENTIFIER ::= { iso(1) member-body(2)
         us(840) rsadsi(113549) pkcs(1) pkcs9(9) smime(16) ct(1) 6 }
```

There are identifiers and structures defined for most tasks we need to perform in exchanging compressed, encrypted, signed, or otherwise authenticated messages. The Bouncy Castle APIs contain high level APIs which are equivalent to most of these structures as well as ASN.1 classes which can be used for dealing with the raw data composing the structures. As some CMS structures can be extremely large, the APIs support both an in-memory and streaming model for creating and parsing CMS structures.

The package name for the CMS API is `org.bouncycastle.cms`. We will start by looking at the simplest structure, which is just used to contain general data.

Data Content

The data content type is identified by the object identifier id-data which is defined as:

```
id-data OBJECT IDENTIFIER ::= { iso(1) member-body(2)
        us(840) rsadsi(113549) pkcs(1) pkcs7(7) 1 }
```

Strictly speaking, id-data can be used to indicate any structure so it can be applied to content which is application specific. That said, the contentType for id-data is generally OCTET STRING, and any additional structure that is present in the content is encoded in the octets making up the contentType.

Bouncy Castle has two interfaces for dealing with in-memory data and a class for supporting stream data. The interfaces also have default implementations for dealing with common situations.

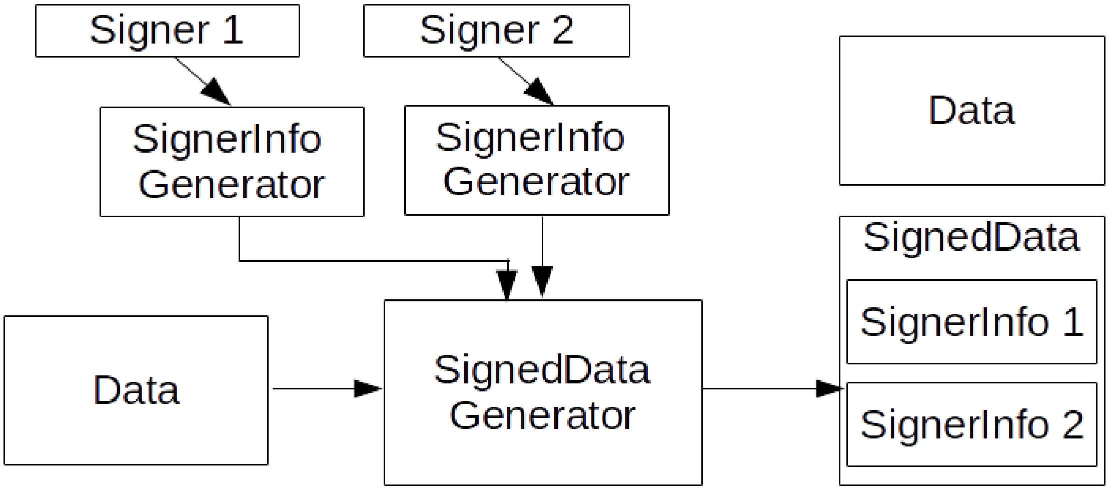
For in-memory use, the basic interface is CMSProcessable, it is data type agnostic and just consists of two methods, one that needs to be implemented to write the data to an output stream, and one to return the content as an `Object`. There is also an extension interface, CMSTypeData, which allows for the data type to be something other than id-data if required and has an additional method, `getContentType()` which returns the `ASN1ObjectIdentifier` representing the content type. The default implementations are provided by `CMSProcessableByteArray` and `CMSProcessableFile` which provide support for handling byte encoded data or files respectively. Both the default implementations have constructors which allow you to specify your own content type where you need `getContentType()` to return something other than the `ASN1ObjectIdentifier` representing id-data.

SignedData Content

SignedData content is specified as being inside a ContentInfo by the presence of the signed-data content type, which is defined as:

```
id-signedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
               us(840) rsadsi(113549) pkcs(1) pkcs7(7) 2 }
```

Although the name SignedData suggests otherwise, SignedData does not necessarily contain the data that has been signed as well, it may only contain information related to the signatures. When it does not contain the data, which is the common case in S/MIME, the SignedData is said to be detached, or external, from the data. Where the SignedData structure is not a detached one, the structure also contains the data that was signed and is described as encapsulated SignedData.



Making detached SignedData with two signers - generally signer details are a private key and certificate

The SignedData structure has the following ASN.1 definition:

```
SignedData ::= SEQUENCE {
    version CMSVersion,
    digestAlgorithms DigestAlgorithmIdentifiers,
    encapContentInfo EncapsulatedContentInfo,
    certificates [0] IMPLICIT CertificateSet OPTIONAL,
    crls [1] IMPLICIT RevocationInfoChoices OPTIONAL,
    signerInfos SignerInfos }

DigestAlgorithmIdentifiers ::= SET OF DigestAlgorithmIdentifier

SignerInfos ::= SET OF SignerInfo
```

The version number can be 1, 3, 4, or 5. In order of testing, the version number is:

- 5 if the SignedData contains "other" certificates and/or "other" CRLs ("other" refers to certificates/CRLs in something other than the X.509 format),

- 4 if version 2 attribute certificates are present,
- 3 if version 1 attribute certificates are present, version 3 SignerInfo are present, and/or encapsulated content which is not of the type id-data is present.
- 1 if none of the above applies.

When the SignedData structure also contains the data which has been signed, the data is stored in the encapContentInfo field. The EncapsulatedContentInfo has a similar structure to ContentInfo. In fact, in the original specification for PKCS#7 the encapContentInfo field was of type ContentInfo. In ASN.1 EncapsulatedContentInfo is defined as follows:

```
EncapsulatedContentInfo ::= SEQUENCE {
    eContentType ContentType,
    eContent [0] EXPLICIT OCTET STRING OPTIONAL }
```

It is the explicit defining of eContent as an octet string which is the difference between PKCS#7 and CMS. It is worth keeping this in mind if you need to deal with any legacy systems or data - the eContent field may not contain a type of OCTET STRING as the original ContentInfo structure has been used instead. Where the SignedData is for a detached, or external, signature the eContent field is absent.

As the names imply, the digestAlgorithms field contains the AlgorithmIdentifiers for the message digests used as the basis of the signatures stored in the SignedData. The certificates and crls fields, if present, contain certificates and certificate revocation lists[17]. Not all of these may be associated with the signatures stored in the signed data, although generally for each SignerInfo there will at least be a certificate that can be used to verify the signature. The generator of the SignedData is under no obligation to do this though and RFC 5652 does allow for the information stored in the certificates and crls fields to be incomplete or even irrelevant.

The actual signatures are in the SignerInfo objects, a set of which appears in the SignedData structure. Note that it is a set, as in an ASN.1 SET, and in ASN.1 the SET type is not ordered. You cannot rely on the ordering of the signers if there is more than one.

We can see how we deal with this by looking at the actual SignerInfo structure.

[17] Both the certificates and crls fields are slightly more sophisticated than I describe here, in general the description fits, but if you really want to see what can be in them have a look at RFC 5652 [74].

```
SignerInfo ::= SEQUENCE {
    version CMSVersion,
    sid SignerIdentifier,
    digestAlgorithm DigestAlgorithmIdentifier,
    signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
    signatureAlgorithm SignatureAlgorithmIdentifier,
    signature SignatureValue,
    unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL }
```

The sid field is the one which provides us with a way of matching a signature that we have against the details of a signer that we might also have. The sid field is really designed with X.509 certificates or public keys in mind and is defined as:

```
SignerIdentifier ::= CHOICE {
     issuerAndSerialNumber IssuerAndSerialNumber,
     subjectKeyIdentifier [0] SubjectKeyIdentifier }
```

with the IssuerAndSerialNumber being a sequence containing the X.500 name of a certificate issuer and the serial number of the certificate the issuer signed that contains the public key needed to verify the signature. If the field contains a subjectKeyIdentifier instead this will be an octet string that can act as an index for the public key. We also saw the SubjectKeyIdentifier type when we were looking at X.509 certificate extensions and often the public key's certificate will also contain the subjectKeyIdentifier as an extension and this can be used to match the certificate needed to verify the signature with the signature in the SignerInfo. The type of sid used affects the version number stored at the start of the structure. If SignerIdentifier is the IssuerAndSerialNumber choice, then the version number is 1, when SignerIdentifier is a SubjectKeyIdentifier the version number is 3.

SignatureValue is simply OCTET STRING which contains the actual signature with the details of the algorithm that was used appearing in the signatureAlgorithm. The signatureAlgorithm is equivalent to the regular AlgorithmIdentifier type we have seen already.

The last two fields worth knowing about are the signedAttrs and unsignedAttrs. These have the same base definition and are defined as follows:

```
SignedAttributes ::= SET SIZE (1..MAX) OF Attribute

UnsignedAttributes ::= SET SIZE (1..MAX) OF Attribute

Attribute ::= SEQUENCE {
    attrType OBJECT IDENTIFIER,
    attrValues SET OF AttributeValue }
```

In the old style ASN.1 used in RFC 5652, AttributeValue is defined as ANY. When the signedAttrs is present, the signature is calculated over the DER encoding of the set making up the field and there will be an attribute in signedAttrs containing the hash of any data, either encapsulated or external, that was associated with the signature. If signedAttrs is not present then the signature is calculated over the data associated with the signature only. The unsignedAttrs field provides a way of introducing data to be stored with the signature after it was originally generated, things like counter signatures and time-stamps appear in the unsigned attributes.

Processing SignedData Content

Before we start looking at the Bouncy Castle APIs for CMS SignedData we need to acknowledge that creation of these structures requires at least a signing key and a public key certificate for verifying the signatures.

In the interests of simplifying the examples, we have assumed that a message to be signed and the private key and signing certificate have been constructed using the following code:

```
byte[] msg = Strings.toByteArray("Hello, world!");

KeyPair selfSignedKp = generateECKeyPair();
X509CertificateHolder selfSignedCert =
    createTrustAnchor(selfSignedKp, "SHA256withECDSA");
```

Note that in this setup the certificate is self-signed. In a real world situation, where the signing party has not been able to personally give you their public key certificate, you would normally expect the certificate to be an end-entity at the end of a certificate path - a path which you would want to validate. It is also worth keeping in mind that SignedData may not contain all, or any, certificates from the path required to verify the signatures that it contains. In our case, for the sake of simplicity, we will generated SignedData structures where the public key certificate of the signing key is available.

As mentioned earlier, the CMS standard is written with the use of BER encoding in mind. For this reason the Bouncy Castle APIs support two models for dealing with CMS SignedData. One model is an in-memory one, which assumes the SignedData being processed can be easily loaded into memory, the other model is a streaming based one. In some ways this is slightly harder to use as you have to be aware of where you are in the structure before you can try some operations, but it does allow for the handling of indefinite length data. We will have a look at the in-memory model first.

Processing Using the In-memory Model

The foundation class for in-memory CMS SignedData is the `CMSSignedData` class. Instances of `CMSSignedData` are produced using `CMSSignedGenerator` objects.

Given we have a private key and associated certificate to use we can create a basic `CMSSignedData` object by going through three, possibly four, steps:

- we use the private key to create a `ContentSigner` for the algorithm we want.

- we create a provider for the digest calculators we need to support in order to hash the signed data.
- we use the `ContentSigner` and the certificate to create a `SignerInfoGenerator` and add that to the `CMSSignedDataGenerator` we are using.
- if we are making the certificate available to anyone reading the SignedData structure we are trying to generate, we also add the certificate to the generator as well.

The following example method shows how this works in practice:

```java
/**
 * Create a SignedData structure.
 *
 * @param signingKey the key to generate the signature with.
 * @param signingCert the cert to verify signingKey's signature with.
 * @param msg the raw message data
 * @param encapsulate true if the data being signed should wrapped.
 * @return a CMSSignedData holding the SignedData created.
 */
public static CMSSignedData createSignedData(
    PrivateKey signingKey, X509CertificateHolder signingCert,
    byte[] msg, boolean encapsulate)
    throws CMSException, OperatorCreationException
{
    ContentSigner contentSigner =
        new JcaContentSignerBuilder("SHA256withECDSA")
                  .setProvider("BC").build(signingKey);

    DigestCalculatorProvider digestCalcProvider =
        new JcaDigestCalculatorProviderBuilder().setProvider("BC").build();

    SignerInfoGenerator signerInfoGenerator =
        new SignerInfoGeneratorBuilder(digestCalcProvider)
                    .build(contentSigner, signingCert);

    CMSSignedDataGenerator gen = new CMSSignedDataGenerator();

    gen.addSignerInfoGenerator(signerInfoGenerator);

    Store<X509CertificateHolder> certs =
        new CollectionStore<X509CertificateHolder>(
                        Collections.singletonList(signingCert));

    gen.addCertificates(certs);

```

```
        CMSTypedData typedMsg = new CMSProcessableByteArray(msg);

        return gen.generate(typedMsg, encapsulate);
    }
```

While this may seem a little long winded, the reason for the approach is that it allows mixing and matching of `ContentSigner` and `DigestCalculatorProvider` objects. `ContentSigner` and `DigestCalculatorProvider` are both interfaces, so the ability to mix and match means that only a conforming implementation of the interface is required to be usable with the SignerInfoGenerator. Being able to do this becomes vital if a hardware cryptography adapter such as a HSM, is involved. HSMs are often only capable of signing and may not even support a comprehensive JCA style API.

In the simplest case, where the `ContentSigner` and the `DigestCalculatorProvider` are both being created from the same provider, it is possible to condense the first 3 steps above into a single step using the following code to create the `SignerInfoGenerator` using `JcaSimpleSignerInfoGeneratorBuilder`:

```
SignerInfoGenerator signerInfoGenerator =
    new JcaSimpleSignerInfoGeneratorBuilder()
        .setProvider("BC")
        .build("SHA256withECDSA", signingKey, signingCert);
```

The `JcaSimpleSignerInfoGeneratorBuilder` still creates a `ContentSigner` and a `DigestCalculatorProvider` but does so internally using the configured provider, or providers appropriate to the JVMs security provider priority configuration if no specific provider is specified via the `setProvider` method.

The method `createSignedData()` we have defined allows us to create a SignedData structure which either encapsulates the data being signed or acts as a detached signature.

To create a SignedData structure with encapsulated data we can invoke the method using our example parameters shown earlier as follows:

```
CMSSignedData encapSigned = createSignedData(selfSignedKp.getPrivate(),
                                        selfSignedCert, msg, true);
```

From a Bouncy Castle point of view, verifying the structure held in a `CMSSignedData` object also involves three steps:

- we provide the original data that was the source for the signatures in the SignedData structure to the object
- we find the `SignerInformation` instance that is related to the signing certificate we have.
- we test the signature on the `SignerInformation` object using signing certificate using a `SignerInformationVerif`

The `SignerInformation` class is the equivalent to the SignerInfo structure defined in CMS and is what is generated internally in a `CMSSignedDataGenerator` by using the `SignerInfoGenerator` objects that are provided to it. We can match certificates with their associated SignerInfo object by using the information contained in the SignerIdentifier in the sid field of the SignerInfo. `SignerInformation.getSID()` provides the SignerIdentifier and it can be used to search the `Store` returned by `CMSSignedData.getCertificates()`.

In the case we are looking at, where the structure also contains the certificate needed to verify it and also contains the data, we can verify the CMSSignedData using the following method:

```
/**
 * Verify the passed in encoding of a CMS SignedData, assumes encapsulated data.
 *
 * @param encodedSignedData the BER encoding of the SignedData
 */
public static void verifySignedEncapsulated(byte[] encodedSignedData)
    throws CMSException, CertificateException, OperatorCreationException
{
    CMSSignedData signedData = new CMSSignedData(encodedSignedData);
    SignerInformationStore signers = signedData.getSignerInfos();
    Store certs = signedData.getCertificates();

    for (SignerInformation signerInfo : signers)
    {
        Collection<X509CertificateHolder> certCollection =
                                   certs.getMatches(signerInfo.getSID());
        X509CertificateHolder cert = certCollection.iterator().next();

        System.out.println("sig verified: " +
            signerInfo.verify(
                    new JcaSimpleSignerInfoVerifierBuilder()
                                .setProvider("BC").build(cert)));
    }
}
```

Apart from the `SignerInformation` class, the other class we see is the `JcaSimpleSignerInfoVerifierBuilder` which provides the same kind of assistance as the `JcaSimpleSignerInfoGenera` class does. Where more flexibility is required, a long hand approach is also available by using the `SignerInformationVerifier` class.

As the SignedData is fully self-contained we can invoke the verify method by just passing the encoding of the SignedData:

```
    verifySignedEncapsulated(encapSigned.getEncoded());
```

A natural question to ask at this point is “why not always do it this way?”. The issue with encapsulated SignedData is that the data is hidden in the ASN.1 encoding of the object. We can still get access to it by doing the following:

```
    CMSTypedData cmsData = encapSigned.getSignedContent();
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();

    cmsData.write(bOut);

    System.out.println(Strings.fromByteArray(bOut.toByteArray()));
```

but there are situations where it can be useful not to wrap the data being signed but maintain the signatures associated with the data that was signed as detached signatures. We will see an example of where this is done in practice when we look at S/MIME further on in the chapter.

Other than not wrapping the signed data in the SignedData structure, generating a detached signature is basically the same as generating an encapsulated one. Our `createSignedData()` method already has the capability to create detached signatures, we just need to pass `false` for the encapsulate argument and perform the new call to `createSignedData()` as follows:

```
    CMSSignedData detachSigned = createSignedData(selfSignedKp.getPrivate(),
                                           selfSignedCert, msg, false);
```

Where things change is what is required to verify the signature - in this case we can no longer assume we have the data and must pass it in separately. This gives us the following slightly different method for doing the verification step:

```
/**
 * Verify the passed in encoding of a CMS SignedData, assumes a detached
 * signature with msg representing the raw data that was signed.
 *
 * @param encodedSignedData the BER encoding of the SignedData
 * @param msg the data that was used to create the SignerInfo in the SignedData
 */
public static void verifySignedDetached(byte[] encodedSignedData, byte[] msg)
    throws CMSException, CertificateException, OperatorCreationException
{
    CMSSignedData signedData = new CMSSignedData(
                         new CMSProcessableByteArray(msg), encodedSignedData);

    SignerInformationStore signers = signedData.getSignerInfos();
    Store<X509CertificateHolder> certs = signedData.getCertificates();

    for (SignerInformation signerInfo : signers)
    {
        Collection<X509CertificateHolder> certCollection
                                 = certs.getMatches(signerInfo.getSID());
        X509CertificateHolder cert = certCollection.iterator().next();

        System.out.println("sig verified: " +
            signerInfo.verify(
                    new JcaSimpleSignerInfoVerifierBuilder()
                                .setProvider("BC").build(cert)));
    }
}
```

where the only difference is the way the CMSSignedData object is constructed.

In terms of invocation, the method just requires the raw message data (msg) as well as the encoding of the SignedData structure. Using the sample parameters we have defined already, we would just invoke the method as follows:

```
    verifySignedDetached(detachSigned.getEncoded(), msg);
```

Processing Using the Streaming Model

One of the ideas behind PKCS#7 originally was that it should be possible to sign and encapsulate arbitrarily large objects in a SignedData structure. The Bouncy Castle APIs also support this by providing additional classes that are stream based: the `CMSSignedDataStreamGenerator` for generating a SignedData structure in a streaming fashion, and the `SignedDataParser` for reading a SignedData structure in a streaming fashion.

We will have a look at generation first. When we are creating a SignedData structure using the streaming model we still need to pass in certificates and SignerInfo generators as we did before. The change in usage comes with writing the data to be encapsulated. When we are ready to write out the data, we call the `open()` method on `CMSSignedDataStreamGenerator` passing it the target stream we want SignedData structure written to. On the call to `open()` the generator will write out the preamble for the structure and then keep writing data written to the stream returned by open into the area allocated to the encapContentInfo field in the SignedData structure. Once `close()` is called on the stream returned by `open()` the post-amble, consisting of the ceritificates/CRLs and SignerInfos, is written out to the target stream and the structure is closed.

The following example creates a SignedData structure using the streaming fashion using the same sample parameters we used in the previous signed data examples.

```
/**
 * Create a SignedData structure using the streaming API.
 *
 * @param signingKey the key to generate the signature with.
 * @param signingCert the cert to verify signingKey's signature with.
 * @param msg the raw message data
 * @return a CMSSignedData holding the SignedData created.
 */
public static byte[] createSignedDataStreaming(
    PrivateKey signingKey, X509CertificateHolder signingCert, byte[] msg)
    throws CMSException, OperatorCreationException, IOException
{
    SignerInfoGenerator signerInfoGenerator =
                new JcaSimpleSignerInfoGeneratorBuilder()
                    .setProvider("BC")
                    .build("SHA256withECDSA", signingKey, signingCert);

    CMSSignedDataStreamGenerator gen = new CMSSignedDataStreamGenerator();

    gen.addSignerInfoGenerator(signerInfoGenerator);

    Store<X509CertificateHolder> certs =
        new CollectionStore<X509CertificateHolder>(
                        Collections.singletonList(signingCert));

    gen.addCertificates(certs);

    ByteArrayOutputStream bOut = new ByteArrayOutputStream();

    OutputStream sOut = gen.open(bOut, true);

    sOut.write(msg);

    sOut.close();

    return bOut.toByteArray();
}
```

As you can see the configuration of the generator is basically the same as for the non-streaming case, the real difference is in the use of the `open()` method to inject the data.

Verification of signed data is slightly complicated in the streaming case as there is only one chance to read the data. Fortunately the SignedData structure contains information about what message digests need to be calculated on the data with the details in the digestAlgorithms field. This requirement does mean the class used for parsing signed data, the `CMSSignedDataParser`, needs to be provided with a `DigestCalculatorProvider` at construction time.

The following code shows an example of how to use a `CMSSignedDataParser` to verify a SignedData structure, in this case without worrying about recovering the encapsulated data:

```java
/**
 * Verify the passed in encoding of a CMS SignedData using the streaming API,
 * assumes encapsulated data.
 *
 * @param encodedSignedData the BER encoding of the SignedData
 */
public static void verifySignedEncapsulatedStreaming(byte[] encodedSignedData)
    throws CMSException, OperatorCreationException,
           IOException, CertificateException
{
    CMSSignedDataParser signedDataParser = new CMSSignedDataParser(
        new JcaDigestCalculatorProviderBuilder().setProvider("BC").build(),
        new ByteArrayInputStream(encodedSignedData));

    signedDataParser.getSignedContent().drain();

    SignerInformationStore signers = signedDataParser.getSignerInfos();
    Store certs = signedDataParser.getCertificates();

    for (SignerInformation signerInfo : signers)
    {
        Collection<X509CertificateHolder> certCollection =
                                certs.getMatches(signerInfo.getSID());
        X509CertificateHolder cert = certCollection.iterator().next();

        System.out.println("sig verified: " +
            signerInfo.verify(new JcaSimpleSignerInfoVerifierBuilder()
                                          .setProvider("BC").build(cert)));
    }
}
```

Note that the `signedDataParser.getSignedContent().drain()` method needs to be called before any attempt to recover the SignerInfo objects. The `drain()` method allows the parser to calculate the message digests necessary to do the validation of the SignerInfo objects by forcing it to read the encapsulated data. The `drain()` method is a method on the `CMSTypedStream` class. The `CMSTypedStream` class also supports a `getContentStream()` method which can be called when you wish to recover the data, as the following sample shows, `getContentStream()` returns an `InputStream`:

```
    CMSTypedStream typedStream = signedDataParser.getSignedContent();

    InputStream data = typedStream.getContentStream();
```

and a full read of the returned stream will recover the encapsulated data that was signed originally.

Troubleshooting and Customization

The APIs around the SignedData classes go further than we have described so far. The signed and unsigned attributes can be customized, counter signatures and time stamps can be added to signed data at a later date, and Bouncy Castle provides some API support for trying to deal with the more common errors that that are made so you can be compatible with deviations from the standard when there is no other choice. We will have a look at some of these sections of the API here.

Adding or Changing Standard Attributes

In the normal case, as in today, the actual signature stored in a SignerInfo is calculated on the DER encoding of the signed attributes. Both the signed attributes and the unsigned attributes are created using implementations of `CMSAttributeTableGenerator`, an interface which allows a custom version of an attribute generator to be defined.

In the example that follows we use a `CMSAttributeTableGenerator` to create a new `AttributeTable` which is missing the CMS algorithm protection attribute, an attribute that was introduced in RFC 6211 [83].

```
/**
 * Create a SignedData structure changing the signed attributes
 * it is created with from the default ones.
 *
 * @param signingKey the key to generate the signature with.
 * @param signingCert the cert to verify signingKey's signature with.
 * @param msg the raw message data
 * @param encapsulate true if the data being signed should wrapped.
 * @return a CMSSignedData holding the SignedData created.
 */
public static CMSSignedData createSignedDataWithAttributesEdit(
    PrivateKey signingKey, X509CertificateHolder signingCert,
    byte[] msg, boolean encapsulate)
    throws CMSException, OperatorCreationException
{
    ContentSigner contentSigner =
        new JcaContentSignerBuilder("SHA256withECDSA")
                  .setProvider("BC").build(signingKey);

    DigestCalculatorProvider digestCalcProvider =
        new JcaDigestCalculatorProviderBuilder().setProvider("BC").build();

    SignerInfoGenerator signerInfoGenerator =
        new SignerInfoGeneratorBuilder(digestCalcProvider)
            .setSignedAttributeGenerator(new CMSAttributeTableGenerator()
            {
                public AttributeTable getAttributes(Map parameters)
                    throws CMSAttributeTableGenerationException
                {
                    AttributeTable table =
                        new DefaultSignedAttributeTableGenerator()
                                            .getAttributes(parameters);

                    return table.remove(CMSAttributes.cmsAlgorithmProtect);
                }
            })
            .build(contentSigner, signingCert);

    CMSSignedDataGenerator gen = new CMSSignedDataGenerator();

    gen.addSignerInfoGenerator(signerInfoGenerator);

    Store<X509CertificateHolder> certs =
        new CollectionStore<X509CertificateHolder>(
                        Collections.singletonList(signingCert));

    gen.addCertificates(certs);

    CMSTypedData typedMsg = new CMSProcessableByteArray(msg);

    return gen.generate(typedMsg, encapsulate);
}
```

We might remove the CMS algorithm protection attribute if we were dealing with a legacy service that could not cope with its presence. Hopefully we would not need to do that, but the other thing the example introduces is the `DefaultSignedAttributeTableGenerator`. If you are customizing signed attributes, this class will generate all the standard attributes for you. In terms of making use of it, you can either extend it, or use it internally as it is in the example code. Usually you will want all the standard attributes, what you then add depends on what will be processing the SignedData structures you are producing.

You can use a similar process to add unsigned attributes. By default the unsigned attributes are empty, so they do not have a default generator. The unsigned attributes are used for things like counter signatures, which we will look at next, and timestamps generated using Time Stamp Protocol (TSP), which we will look at further on in the chapter.

Adding a Counter Signature

In addition to being able to generate signatures on data, CMS also allows for signatures to be generated that can act as counter signatures on an existing signer's signature. The process for doing this in the Bouncy Castle APIs takes a few steps. First we need to add the `SignerInfoGenerator` objects for the counter signers to a `CMSSignedDataGenerator` and then we call `CMSSignedDataGenerator.generateCounterSigners()` passing in the `SignerInformation` object that represents the signer we wish to counter sign. Finally, having generated the counter signers we add them to the `SignerInformation` object we just counter signed for later retrieval.

The following example method shows how a single counter signature can be added, when we assume it is always the first `SignerInformation` associated with a `CMSSignedData` we wish to counter sign.

```java
/**
 * Update a SignerInfo to include a counter signature. The resulting
 * CMSSignedData will also contain the certificate for the counter signer.
 *
 * @param original the SignedData with the SignerInfo to be counter signed.
 * @param counterSignKey the key being used for counter signing.
 * @param counterSignCert the certificate associated with counterSignKey.
 * @return an updated SignedData with the counter signature.
 */
public static CMSSignedData addCounterSignature(
    CMSSignedData original,
    PrivateKey counterSignKey, X509CertificateHolder counterSignCert)
    throws CMSException, OperatorCreationException
{
    SignerInfoGenerator signerInfoGenerator =
        new JcaSimpleSignerInfoGeneratorBuilder()
            .setProvider("BC")
            .build("SHA256withECDSA", counterSignKey, counterSignCert);

    CMSSignedDataGenerator gen = new CMSSignedDataGenerator();

    gen.addSignerInfoGenerator(signerInfoGenerator);

    SignerInformationStore signers = original.getSignerInfos();
    SignerInformation signerInfo = signers.iterator().next();

    signerInfo = SignerInformation.addCounterSigners(
                        signerInfo, gen.generateCounterSigners(signerInfo));

    Collection originalCerts = original.getCertificates().getMatches(null);

    Set totalCerts = new HashSet();

    totalCerts.addAll(originalCerts);
    totalCerts.add(counterSignCert);

    CMSSignedData counterSigned = CMSSignedData.replaceSigners(
                                original, new SignerInformationStore(signerInfo));

    counterSigned = CMSSignedData.replaceCertificatesAndCRLs(
                 counterSigned, new CollectionStore(totalCerts), null, null);

    return counterSigned;
}
```

You can see how different counter signature generation is from regular CMS SignedData generation by looking at the return value of `generateCounterSigners()` - it returns a store of `SignerInformation` objects. Even when counter signing a signer associated with encapsulated data, the counter signatures are generated on the original signer's signature and are only meaningful when looked at in the context of the original signer as well.

A counter signature can be verified in the same way as a regular one, by calling the `verify()` method on the `SignerInformation` object that contains it. Bouncy Castle also has an instance method on the `CMSSignedData` class, `verifySignatures()` which you can use to verify all the SignerInfo structures including the counter signatures. To use it you provide an implementation of the `SignerInformationVerifierProvider` interface. You will see how it is done in the following code:

```
/**
 * Verify all SignerInfos from a SignedData structure, including any
 * counter signatures.
 *
 * @param signedData BER encoding of the SignedData structure.
 */
public static void verifyAllSigners(CMSSignedData signedData)
    throws CMSException
{
    final Store<X509CertificateHolder> certs = signedData.getCertificates();

    System.out.println("signers verified: " + signedData.verifySignatures(
        new SignerInformationVerifierProvider()
        {
            public SignerInformationVerifier get(SignerId signerId)
                throws OperatorCreationException
            {
                try
                {
                    X509CertificateHolder cert = (X509CertificateHolder)certs
                                    .getMatches(signerId).iterator().next();

                    return new JcaSimpleSignerInfoVerifierBuilder()
                                              .setProvider("BC").build(cert);
                }
                catch (CertificateException e)
                {
                    throw new OperatorCreationException(
                             "verifier provider failed: " + e.getMessage(), e);
                }
            }
        }));
}
```

As you can see the `SignerInformationVerifierProvider` interface has a single `get()` method which takes a signer identifier as an argument and expects to return a `SignerInformationVerifier`. In the case of our example we know that all the certificates are available in the `CMSSignedData` object we were given so we can just create the verifiers around the certificates we have. As we warned earlier it may be more complicated than this in practice.

The following code snippet is to fill in the gaps concerning where the counter signer came from and how the two methods would be called together. You can treat it as a continuation of the earlier sample code we used for creating the `encapSigned` object.

```
KeyPair counterKp = generateECKeyPair();
X509CertificateHolder counterCert =
    createTrustAnchor(counterKp, "SHA256withECDSA");

CMSSignedData counterSigned = addCounterSignature(encapSigned,
                                    counterKp.getPrivate(), counterCert);

verifyAllSigners(counterSigned);
```

As you can see there is nothing odd about the credentials used for counter signing. The difference in this case is simply about the manner in which they are applied - to a SignerInfo rather than the data that was originally signed.

Dealing with Improper Encoding of Signed Attributes

Occasionally you will see apparently correct SignerInfo objects fail to verify. Assuming the creators of the SignedData structure involved have otherwise done things correctly the common reason for this failure is they have forgotten to DER encode the signed attributes used to calculate a signature that is verified by the signer's certificate. The DER encoding is important in this case as the attributes are in a SET, which is an unordered type, so in order for DER encoding to always produce the same encoding the SET making up the attributes must be sorted before its encoding is generated.

Ideally anyone who is making this mistake will fix it, sometimes this is not possible so the Bouncy Castle APIs make it possible to create a custom `SignerInformation` class which allows regular definite-length encoding to be used on the signed attributes.

You can do this as follows:

```java
/**
 * Override for SignerInformation to stop DER sorting of the signed attributes
 * in signature evaluation.
 */
public class SignerInfoUsingDefLength
    extends SignerInformation
{
    protected SignerInfoUsingDefLength(SignerInformation baseInfo)
    {
        super(baseInfo);
    }

    /**
     * Override to allow the encoded attributes to be returned as they are
     * in the SignerInfo.
     *
     * @return the signed attributes as a definite length encoding.
     * @throws IOException in case of an encoding error.
     */
    public byte[] getEncodedSignedAttributes()
        throws IOException
    {
        return signedAttributeSet.getEncoded(ASN1Encoding.DL);
    }
}
```

The use of `ASN1Encoding.DL` is not something you should commonly accept though - the idea of DER is to provide only one possible encoding that can be used to verify a signature. Definite-length encoding of a SET says nothing about the order of the elements and the additional flexibility could be exploited by someone looking to change the meaning of the signature in the SignerInfo.

EnvelopedData Content

EnvelopedData is used to carry encrypted information. EnvelopedData is also carried inside ContentInfo and its presence is indicated by the enveloped-data content type:

```
id-envelopedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
                us(840) rsadsi(113549) pkcs(1) pkcs7(7) 3 }
```

EnvelopedData always encapsulates the data that has been encrypted and has the following ASN.1 structure:

```
EnvelopedData ::= SEQUENCE {
    version CMSVersion,
    originatorInfo [0] IMPLICIT OriginatorInfo OPTIONAL,
    recipientInfos RecipientInfos,
    encryptedContentInfo EncryptedContentInfo,
    unprotectedAttrs [1] IMPLICIT UnprotectedAttributes OPTIONAL }
```

As with SignedData the version number varies according to the contents of the EnvelopedData. If the originatorInfo field contains "other" certificates or "other" CRLs (with "other" think not X.509 format), the version number is 4. If the originatorInfo field contains version 2 attribute certificates or password or "other" recipients are present the version number is 3. If originatorInfo is absent and unprotectedAttrs is absent and all the members of recipientInfos are version 0, then the version number is 0. Assuming none of the previous things apply the version number is 2.

The originatorInfo, if present, and the recipientInfos fields carry information about the originator of the EnvelopedData and its intended recipients. The fields have the following ASN.1 type definitions:

```
OriginatorInfo ::= SEQUENCE {
    certs [0] IMPLICIT CertificateSet OPTIONAL,
    crls [1] IMPLICIT RevocationInfoChoices OPTIONAL }

RecipientInfos ::= SET SIZE (1..MAX) OF RecipientInfo
```

OriginatorInfo contains optional fields with the same definitions as we saw earlier with SignedData. Assuming you are one of the intended recipients of the EnvelopedData, ideally if present, the OriginatorInfo structure will contain enough information for you to be able to send an encrypted reply back to the original sender. RFC 5652 does allow for the information stored in the originatorInfo field to be incomplete or irrelevant to the actual originator.

RecipientInfos, as the definition implies, is a set of details for the use of recipients that are able to decrypt the enveloped data. The RecipientInfo type is a CHOICE which has the following definition:

```
RecipientInfo ::= CHOICE {
    ktri KeyTransRecipientInfo,
    kari [1] KeyAgreeRecipientInfo,
    kekri [2] KEKRecipientInfo,
    pwri [3] PasswordRecipientInfo,
    ori [4] OtherRecipientInfo }
```

We will have a look at some of the different RecipientInfo choices a bit further down, but the most useful thing to say right now is that each choice is expected to contain an EncryptedKey type which encodes and wraps the symmetric key used to encrypt the enveloped data using an algorithm that should be appropriate for the recipient the RecipientInfo is targeted at. The EncryptedKey type is simply a byte array and has the following definition:

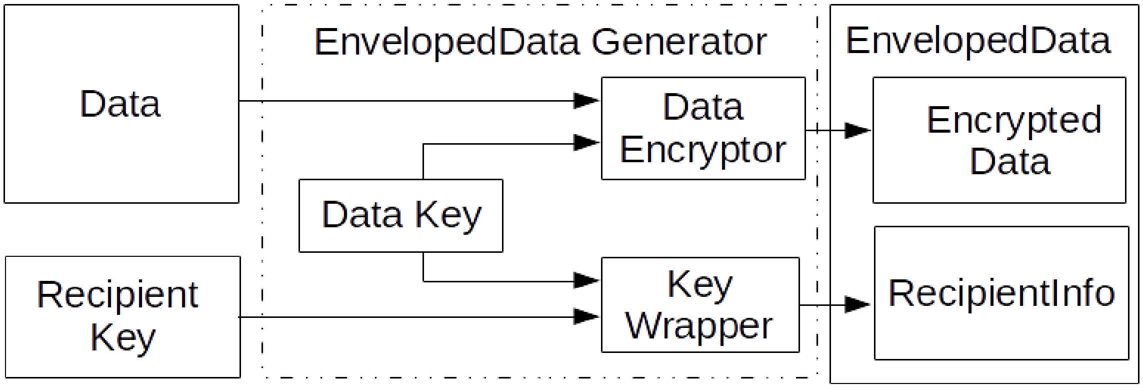
```
EncryptedKey ::= OCTET STRING
```

In addition to the encrypted key there is normally a recipient identifier, or RID, inside each RecipientInfo structure as well. This allows a recipient to identify which RecipientInfo it should use for unwrapping the encrypted key so that it can recover the data that was encrypted.

The encryptedContentInfo field contains the data we are trying to protect. Its type has the following ASN.1 definition:

```
EncryptedContentInfo ::= SEQUENCE {
    contentType ContentType,
    contentEncryptionAlgorithm ContentEncryptionAlgorithmIdentifier,
    encryptedContent [0] IMPLICIT EncryptedContent OPTIONAL }

EncryptedContent ::= OCTET STRING
```

As you can see the type is quite basic. It contains just enough information to allow a recipient to take the encrypted key field in their RecipientInfo, unwrap the encrypted key and then use the contentEncryptionAlgorithm (at the end of the day a regular AlgorithmIdentifier) to create a cipher that can be used to recover the original data that was enveloped.



Making EnvelopedData - the data key is generated internally and wrapped by the recipient key

Finally, EnvelopedData has an unprotectedAttrs field which can be used to associate additional attributes with the protected data. This field uses the same Attribute type that SignedData does and has the following ASN.1 definition:

```
UnprotectedAttributes ::= SET SIZE (1..MAX) OF Attribute
```

Processing EnvelopedData Content

We will now look at creating and processing EnvelopedData using the CMS API. For simplicity the example code in each section assumes the following things have been defined:

```
JcaX509CertificateConverter converter =
    new JcaX509CertificateConverter().setProvider("BC");

byte[] msg = Strings.toByteArray("Hello, world!");

CMSEnvelopedDataGenerator envGen = new CMSEnvelopedDataGenerator();
```

We are using `CMSEnvelopedDataGenerator` as we will look at generating and parsing using the in-memory model first. EnvelopedData support in the Bouncy Castle APIs also includes a streaming model. Recipient generation is the same for both the in-memory and streaming model, so while there are some differences between the way the different models work for generating and parsing the EnvelopedData, recipient generation and use is fundamentally the same. We will have a look at the streaming model after we have reviewed the various recipient types available.

In the case of the in-memory model, assuming some RecipientInfo generators have been added, the `CMSEnvelopedDataGenerator` object returns a `CMSEnvelopedData` object when its `generate()` method is called.

```
    CMSEnvelopedData envData = envGen.generate(
        new CMSProcessableByteArray(msg),
        new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
            .setProvider("BC")
            .build());
```

We can then recover a BER encoding of the enveloped-data structure by calling the `getEncoded()` method on the `CMSEnvelopedData` object, as in:

```
    CMSEnvelopedData envData = envGen.generate(
        new CMSProcessableByteArray(msg),
        new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
            .setProvider("BC")
            .build());
```

We will now have a look at some of the supported recipient types.

ktri - the KeyTransRecipientInfo Choice

The ktri value is chosen when the RecipientInfo is based on a key transport mechanism such as what you would use with an algorithm like RSA.

It has a simple ASN.1 structure, reflecting the fact that it is basically a key wrap using an asymmetric key, rather than just a symmetric one. We can see from the below that it allows for using two methods for identifying the Recipient, in much the same way as we saw with SignerInfo.

```
KeyTransRecipientInfo ::= SEQUENCE {
    version CMSVersion,  -- always set to 0 or 2
    rid RecipientIdentifier,
    keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
    encryptedKey EncryptedKey }

RecipientIdentifier ::= CHOICE {
    issuerAndSerialNumber IssuerAndSerialNumber,
    subjectKeyIdentifier [0] SubjectKeyIdentifier }
```

In the case where the RecipientIdentifier is an IssuerAndSerialNumber the version field is set to 0, otherwise it is set to 2.

The following method shows how to add a recipient based on RSA using OAEP as the encoding mechanism for the symmetric key wrapping.

```java
/**
 * Add a KeyTransRecipientInfo to the passed in enveloped-data generator.
 *
 * @param envGen the generator to add the KeyTransRecipientInfo to.
 * @param encryptionCert the public key certificate for the targeted recipient.
 */
public static void addKeyTransRecipient(
        CMSEnvelopedGenerator envGen,
        X509Certificate encryptionCert)
throws GeneralSecurityException
{
    JcaAlgorithmParametersConverter paramsConverter =
                                            new JcaAlgorithmParametersConverter();

    AlgorithmIdentifier oaepParams = paramsConverter.getAlgorithmIdentifier(
        PKCSObjectIdentifiers.id_RSAES_OAEP,
        new OAEPParameterSpec("SHA-256",
            "MGF1", new MGF1ParameterSpec("SHA-256"),
            PSource.PSpecified.DEFAULT));

    envGen.addRecipientInfoGenerator(
        new JceKeyTransRecipientInfoGenerator(
            encryptionCert, oaepParams).setProvider("BC"));
}
```

The example introduces the `JcaAlgorithmParametersConverter` which is a utility class that will convert an `AlgorithmParameters` object or an `AlgorithmParametersSpec` object into an `AlgorithmIdentifier` which can then be used by the `JceKeyTransRecipientInfoGenerator` to set the parameters for the key wrap for the recipient. In this case we have elected to use OAEP based on the SHA-256 message digest. The passing of `encryptionCert` as the first parameter to the `JceKeyTransRecipientInfoGenerator` serves two purposes, it provides the information required to generate the recipient identifier and it also provides the recipient's public key to use for the OAEP key wrapping.

The following sample code can be used to call the `addKeyTransRecipient()` method:

```java
KeyPair recipientRsaKp = generateRSAKeyPair();
X509Certificate recipientRsaCert = converter.getCertificate(
    createTrustAnchor(recipientRsaKp, "SHA256withRSA"));

addKeyTransRecipient(envGen, recipientRsaCert);
```

and when the `addKeyTransRecipient()` method returns the `CMSEnvelopedGenerator`, `envGen`, will have a `RecipientInfoGenerator` for `recipientRsaCert` added to it.

It is also possible to use a recipient identifier based on a key identifier, such as a public key's subject key identifier. In this case we need to create a `JceKeyTransRecipientInfoGenerator` passing both the key identifier and the public key to use for key wrapping to the constructor. To set up a RecipientInfo generator using a key identifier and public key instead of a certificate and using the same OAEP parameters as before we would use:

```
envGen.addRecipientInfoGenerator(
    new JceKeyTransRecipientInfoGenerator(
        keyIdentifier,
        oaepParams,
        wrappingKey).setProvider("BC"));
```

where `keyIdentifier` is a `byte[]` representing the public key's identifier and `wrappingKey` is the `PublicKey` object for the recipient.

In order to extract and decrypt the data in the envelope we first need to identify which recipient to use and then apply our private key to unwrap our encrypted copy of the secret key that was used to protect the data. Have a look at the following method that takes a BER encoding of an EnvelopedData structure and then looks up the correct RecipientInfo and returns the decrypted data.

```
/**
 * Extract the original data that was encrypted from the EnvelopedData
 * structure by using the recipient matching the passed in parameters.
 *
 * @param encEnvelopedData the BER encoded EnvelopedData structure.
 * @param privateKey the recipient private key (for key transport).
 * @param encryptionCert the recipient's corresponding public key certificate.
 * @return the original data that was enveloped as a byte[].
 */
public static byte[] extractUsingKeyTransRecipient(
    byte[] encEnvelopedData,
    PrivateKey privateKey, X509Certificate encryptionCert)
    throws CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientInformation recipient = recipients.get(
                                new JceKeyTransRecipientId(encryptionCert));

    if (recipient != null)
    {
        return recipient.getContent(
                new JceKeyTransEnvelopedRecipient(privateKey)
```

```
                        .setProvider("BC"));
        }

        throw new IllegalArgumentException("recipient for certificate not found");
    }
```

Just as we used our certificate to provide an identifier for the recipient when the RecipientInfo was created we can also use our certificate to create a `JceKeyTransRecipientId` which can be used to look up a KeyTransRecipientInfo associated with our certificate. In the Bouncy Castle APIs these objects are of the type `RecipientInformation` and, assuming we find one for our certificate, we can call the `RecipientInformation.getContent()` method passing a `Recipient` that has been created using our private key. The `RecipientInformation` object then uses our `Recipient` object internally to recover the wrapped secret key contained in its KeyTransRecipientInfo structure and decrypts the encrypted data encapsulated in the CMSEnvelopedData object.

The following lines of code show a simple example of applying the `extractUsingKeyTransRecipient()` method to recover and print the test message from the BER encoding of an EnvelopedData structure we have generated.

```
byte[] keyTransRecovered = extractUsingKeyTransRecipient(
    envEnc, recipientRsaKp.getPrivate(), recipientRsaCert);

System.err.println(Strings.fromByteArray(keyTransRecovered));
```

Assuming you are using the test message given earlier and the KeyTransRecipientInfo was properly added, if you run the above you should see the string:

```
Hello, world!
```

output to the console.

kari - the KeyAgreeRecipientInfo Choice

The kari value is chosen when the RecipientInfo is based on a key agreement mechanism such as what you would use with an algorithm like Diffie-Hellman, MQV, or Unified Diffie-Hellman.

The following method shows how to add a recipient based on EC co-factor Diffie-Hellman with the agreed key being a 256 bit AES wrapping key:

```
/**
 * Add a KeyAgreeRecipientInfo to the passed in enveloped-data generator.
 *
 * @param envGen the generator to add the KeyAgreeRecipientInfo to.
 * @param initiatorKey the private key for the enveloped-data originator.
 * @param initiatorCert the public key certificate for the originator.
 * @param recipientCert the public key certificate for the targeted recipient.
 */
public static void addKeyAgreeRecipient(
    CMSEnvelopedGenerator envGen,
    PrivateKey initiatorKey, X509Certificate initiatorCert,
    X509Certificate recipientCert)
throws GeneralSecurityException
{
    envGen.addRecipientInfoGenerator(
        new JceKeyAgreeRecipientInfoGenerator(
            CMSAlgorithm.ECCDH_SHA384KDF,
            initiatorKey,
            initiatorCert.getPublicKey(),
            CMSAlgorithm.AES256_WRAP)
            .addRecipient(recipientCert).setProvider("BC"));
}
```

There are two big differences between the key agreement case and the key transport case we looked at earlier. The first is that the public key of the creator of the RecipientInfo needs to be provided and the agreement process produces an agreed value rather than directly encrypting a block. The second is that as it is an agreed value we produce, a symmetric wrapping algorithm also needs to be specified in order to use the agreed value to wrap the key used to encrypted the data that was enveloped. This second level use of the agreed value is worth appreciating as it means that the key agreement recipient can also appear in a message carrying any other recipient. As far as encrypting the data is concerned everyone needs to recover the same key, it is just that the mechanism employed to wrap the key is different.

We can see how these differences manifest in the contents of the ASN.1 used to define a KeyAgreeRecipientInfo structure:

```
KeyAgreeRecipientInfo ::= SEQUENCE {
    version CMSVersion,  -- always set to 3
    originator [0] EXPLICIT OriginatorIdentifierOrKey,
    ukm [1] EXPLICIT UserKeyingMaterial OPTIONAL,
    keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
    recipientEncryptedKeys RecipientEncryptedKeys }

OriginatorIdentifierOrKey ::= CHOICE {
    issuerAndSerialNumber IssuerAndSerialNumber,
    subjectKeyIdentifier [0] SubjectKeyIdentifier,
    originatorKey [1] OriginatorPublicKey }

OriginatorPublicKey ::= SEQUENCE {
    algorithm AlgorithmIdentifier,
    publicKey BIT STRING }

RecipientEncryptedKeys ::= SEQUENCE OF RecipientEncryptedKey

RecipientEncryptedKey ::= SEQUENCE {
    rid KeyAgreeRecipientIdentifier,
    encryptedKey EncryptedKey }

KeyAgreeRecipientIdentifier ::= CHOICE {
    issuerAndSerialNumber IssuerAndSerialNumber,
    rKeyId [0] IMPLICIT RecipientKeyIdentifier }

RecipientKeyIdentifier ::= SEQUENCE {
    subjectKeyIdentifier SubjectKeyIdentifier,
    date GeneralizedTime OPTIONAL,
    other OtherKeyAttribute OPTIONAL }
```

The originator field being a CHOICE allows a few ways for the recipient to recover the originator public key it needs and also by using an OriginatorPublicKey choice allows the actual originator of the message to stay anonymous. The ukm field provides data that can be used to ensure that, where static keys are in use, no two key agreements between the same two parties result in the same wrapping key being generated. The keyEncryptionAlgorithm field describes the wrapping algorithm used to create encrypted keys. Finally, you will probably notice that more than one recipient key can be included in the KeyAgreeRecipientInfo. The Bouncy Castle API supports this and it is the reason behind the use of the `addRecipient()` method in the `JceKeyAgreeRecipientInfoGenerator` to pass the recipient details, rather than having that done by using a constructor.

In our current example we are only looking at one recipient, so the following code fragment is enough to make use of the `addKeyAgreeRecipient()` method:

```
    KeyPair originatorEcKp = generateECKeyPair();
    X509Certificate originatorEcCert = converter.getCertificate(
        createTrustAnchor(originatorEcKp, "SHA256withECDSA"));

    KeyPair recipientEcKp = generateECKeyPair();
    X509Certificate recipientEcCert = converter.getCertificate(
        createTrustAnchor(recipientEcKp, "SHA256withECDSA"));

    addKeyAgreeRecipient(envGen,
             originatorEcKp.getPrivate(), originatorEcCert, recipientEcCert);
```

As you can see the example calling code, unlike the example code for key transport, has to generate two key pairs and two certificates as the details of the key pair for the message originator are important. This is from the perspective of both the initiator doing the calculation of the initial shared secret and thus the wrapping key used to protect the encryption key used for the data, and also for the recipient themselves as they need the public key of the originator to combine with their private key to do there own shared value calculation.

On the other hand, the parameters passed by the recipient to recover the original data are the same as you would see in the case of a key transport based solution:

```
/**
 * Extract the original data that was encrypted from the EnvelopedData
 * structure by using the recipient matching the passed in parameters.
 *
 * @param encEnvelopedData the BER encoded EnvelopedData structure.
 * @param recipientKey the recipient private key (for agreement).
 * @param recipientCert the recipient's corresponding public key certificate.
 * @return the original data that was enveloped as a byte[].
 */
public static byte[] extractUsingKeyAgreeRecipient(
    byte[] encEnvelopedData,
    PrivateKey recipientKey, X509Certificate recipientCert)
throws CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientId rid = new JceKeyAgreeRecipientId(recipientCert);
    RecipientInformation recipient = recipients.get(rid);

    return recipient.getContent(
        new JceKeyAgreeEnvelopedRecipient(recipientKey).setProvider("BC"));
}
```

and for the most part the code is identical, just making use of a different RecipientId, the JceKeyAgreeRecipientId, and a different recipient type: the JceKeyAgreeEnvelopedRecipient.

```
    byte[] keyAgreeRecovered = extractUsingKeyAgreeRecipient(
        envEnc, recipientEcKp.getPrivate(), recipientEcCert);

    System.err.println(Strings.fromByteArray(keyAgreeRecovered));
```

Assuming you are using the test message given earlier and the KeyAgreeRecipientInfo was properly added, if you run the above you should see the string:

```
Hello, world!
```

You can also try combining this example with the previous one for key transport and you will see that both mechanisms can be used quite comfortably in the same EnvelopedData structure.

kekri - the KEKRecipientInfo Choice

The kekri value is chosen when the RecipientInfo is based on a symmetric key-encryption-key (KEK) mechanism such as AES KWP.

The KEK recipient makes use of the following ASN.1 definitions:

```
KEKRecipientInfo ::= SEQUENCE {
    version CMSVersion,  -- always set to 4
    kekid KEKIdentifier,
    keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
    encryptedKey EncryptedKey }

KEKIdentifier ::= SEQUENCE {
    keyIdentifier OCTET STRING,
    date GeneralizedTime OPTIONAL,
    other OtherKeyAttribute OPTIONAL }
```

This is a much simpler RecipientInfo than the key agreement version and has more in common with the way the KeyTransRecipientInfo is defined - not surprisingly as KEK is just a symmetric wrapping operation as opposed to an asymmetric one. Normally the KEKIdentifier is just composed of an OCTET STRING and that is the way it is used in our method for adding a KEK recipient that appears next. RFC 5652 recommends against making use of the OtherKeyAttribute for interoperability reasons.

Here is a method for adding a KEK recipient generator:

```
/**
 * Add a KEKRecipientInfo to the passed in enveloped-data generator.
 *
 * @param envGen the generator to add the KEKRecipientInfo to.
 * @param keyID the keyID to store for the recipient to match.
 * @param wrappingKey the wrapping key corresponding to the keyID.
 */
public static void addKEKRecipient(
    CMSEnvelopedGenerator envGen,
    byte[] keyID, SecretKey wrappingKey)
{
    envGen.addRecipientInfoGenerator(
        new JceKEKRecipientInfoGenerator(keyID, wrappingKey)
            .setProvider("BC"));
}
```

In order to make use of the method we need code which will provide a key identifier as well as the symmetric key we wish to use for wrapping. The RecipientInfo generator will select a standard key-wrapping algorithm, appropriate for the symmetric key passsed in. For example in the case following:

```
byte[] keyID = Strings.toByteArray("KeyID");
SecretKey wrappingKey = new SecretKeySpec(
                Hex.decode("000102030405060708090a0b0c0d0e0f"), "AES");
addKEKRecipient(envGen, keyID, wrappingKey);
```

the key wrapping algorithm will be AES KW.

As the RecipientInfo in the EnvelopedData is tagged with a key identifier, extracting the original data is just a matter of making the right key identifier available in order to locate the correct RecipientInfo and providing the correct symmetric key to do the key unwrap required to retrieve the original data.

The following provides an example of how to do the data extraction using a KEK key and its key identifier. Have a look at the code and you will see that while it is not using public/private keys, the approach still follows the pattern we have established with the other recipient types.

```
/**
 * Extract the original data that was encrypted from the EnvelopedData
 * structure by using the recipient matching the passed in parameters.
 *
 * @param encEnvelopedData the BER encoded EnvelopedData structure.
 * @param keyID the keyID matching the recipient.
 * @param wrappingKey the wrapping key corresponding to the keyID.
 * @return the original data that was enveloped as a byte[].
 */
public static byte[] extractUsingKEKRecipient(
    byte[] encEnvelopedData,
    byte[] keyID, SecretKey wrappingKey)
    throws CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientId rid = new KEKRecipientId(keyID);
    RecipientInformation recipient = recipients.get(rid);

    return recipient.getContent(
        new JceKEKEnvelopedRecipient(wrappingKey)
            .setProvider("BC"));
}
```

This final fragment just shows how to call the extract method.

```
    byte[] kekRecovered = extractUsingKEKRecipient(envEnc, keyID, wrappingKey);

    System.err.println(Strings.fromByteArray(kekRecovered));
```

Which when run should display the “Hello, world!” message again.

The example code from this section can also be combined with the previous examples and you will see that all the recipient types we have discussed so far can be combined in a single message.

pwri - the PasswordRecipientInfo Choice

The pwri value is chosen when the RecipientInfo is based on a PBE based key wrapping mechanism.

The difference between this RecipientInfo and the one used for KEK is that it carries a key derivation algorithm rather than a key identifier. The ASN.1 for PasswordRecipientInfo follows.

```
PasswordRecipientInfo ::= SEQUENCE {
    version CMSVersion,   -- Always set to 0
    keyDerivationAlgorithm [0] KeyDerivationAlgorithmIdentifier OPTIONAL,
    keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
    encryptedKey EncryptedKey }
```

The following method shows how to add a password based RecipientInfo to a generator:

```java
/**
 * Add a PasswordRecipientInfo to the passed in enveloped-data generator.
 *
 * @param envGen the generator to add the PasswordRecipientInfo to.
 * @param passwd the password to use as the basis of the PBE key.
 * @param salt the salt to use to generate the PBE key.
 * @param iterationCount the iterationCount to use to generate the PBE key.
 */
public static void addPasswordRecipient(
    CMSEnvelopedGenerator envGen,
    char[] passwd,
    byte[] salt,
    int iterationCount)
{
    envGen.addRecipientInfoGenerator(
        new JcePasswordRecipientInfoGenerator(CMSAlgorithm.AES256_CBC, passwd)
            .setProvider("BC")
            .setPasswordConversionScheme(PasswordRecipient.PKCS5_SCHEME2_UTF8)
            .setPRF(PasswordRecipient.PRF.HMacSHA384)
            .setSaltAndIterationCount(salt, iterationCount));
}
```

The PBE scheme used is the one which was originally recommended in RFC 3211 [52], PKCS#5 Scheme 2, also known as PBKDF2. As you can see, both from the ASN.1 and the sample code, there is no identifier a recipient could make use of to recognize the RecipientInfo to choose for processing. The only thing a recipient can base its initial check of the RecipientInfo on is the use of the pwri choice. This issue does not mean you cannot have more than one PBE based RecipientInfo in a single EnvelopedData message, but it does mean the only way to check the recipient is to try and decrypt the encrypted key stored in the PasswordRecipientInfo structure.

You can invoke the `addPasswordRecipient()` method using the following code:

```
    char[] passwd = "password".toCharArray();

    addPasswordRecipient(envGen, passwd, Strings.toByteArray("Salt"), 2048);
```

It is worth noting that in real life you would use a bigger and (hopefully) random value for the salt and you would want to be using a password now with an entropy value of around 112 bits, which for an 8 bit ASCII character set means at least 14 characters.

In the case of extraction, the salt and iteration count are already stored as parameters in the RecipientInfo we are using. So, as you can see from the following method, the only extra data we need to provide with the encoding of the EnvelopedData structure is the password.

```
/**
 * Extract the original data that was encrypted from the EnvelopedData
 * structure by using the recipient matching the passed in parameters.
 *
 * @param encEnvelopedData the BER encoded EnvelopedData structure.
 * @param passwd the password to use as the source of the PBE key.
 * @return the original data that was enveloped as a byte[].
 */
public static byte[] extractUsingPasswordRecipient(
    byte[] encEnvelopedData,
    char[] passwd)
    throws CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientId rid = new PasswordRecipientId();
    RecipientInformation recipient = recipients.get(rid);

    return recipient.getContent(
        new JcePasswordEnvelopedRecipient(passwd)
            .setProvider("BC")
            .setPasswordConversionScheme(PasswordRecipient.PKCS5_SCHEME2_UTF8));
}
```

Finally, the following code fragment can be used to invoke the `extractUsingPasswordRecipient()` method:

```
    byte[] passwordRecovered = extractUsingPasswordRecipient(envEnc, passwd);

    System.err.println(Strings.fromByteArray(passwordRecovered));
```

And as before, assuming the same sample data, you will be greeted with “Hello, world!” if all has gone well.

Using the Streaming Model

In case of very large data blocks EnvelopedData, like SignedData, can also be created and processed using a streaming model. If you are planning to make use of the streaming model instead you need to use the `CMSEnvelopedStreamGenerator` class which can be created as follows:

```
CMSEnvelopedDataStreamGenerator envStreamGen =
                                      new CMSEnvelopedDataStreamGenerator();
```

Adding recipient generators to `CMSEnvelopedStreamGenerator` is the same as for `CMSEnvelopedDataGenerator`, both classes inherit from `CMSEnvelopedGenerator` and you may have already noticed that `CMSEnvelopedGenerator` was the class used in all the recipient add methods. As this is the case while we will only look at the use of a KeyTransRecipientInfo below, the same rules can be used with all the other RecipientInfo types we have looked at.

As you can see in the code fragment below, the `addKeyTransRecipient()` method is used the same way as before, however the way the data to be encrypted is added to the EnvelopedData structure is different.

```
addKeyTransRecipient(envStreamGen, recipientRsaCert);

ByteArrayOutputStream bOut = new ByteArrayOutputStream();
OutputStream out = envStreamGen.open(
    bOut,
    new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
        .setProvider("BC")
        .build());

out.write(msg);

out.close();

envEnc = bOut.toByteArray();
```

In terms of adding the data, the significant code starts with the `envStream.open()` call. The OutputStream passed to the `open()` method is, ultimately, where the encoding of the EnvelopedData structure ends up. In our case, for the sake of example, we are using a `ByteArrayOutputStream`, so we are able to recover the EnvelopedData's encoding by calling `bOut.toByteArray()` as we do at the end of the fragment.

Note: we only call `bOut.toByteArray()` after calling `out.close()` - closing the `OutputStream` returned by `open()` ensures that the EnvelopedData encoding is properly terminated.

The method for doing the extraction looks very similar to what we saw earlier in the equivalent method for extracting the data that was encrypted when using a KeyTransInfo recipient. We use a

`CMSEnvelopedDataParser` rather than an in-memory `CMSEnvelopedData` object to read the EnvelopedData as follows:

```
/**
 * Extract the original data that was encrypted from the EnvelopedData
 * structure by using the recipient matching the passed in parameters
 * and the CMS streaming model.
 *
 * @param encEnvelopedData the BER encoded EnvelopedData structure.
 * @param privateKey the recipient private key (for key transport).
 * @param encryptionCert the recipient's corresponding public key certificate.
 * @return the original data that was enveloped as a byte[].
 */
public static CMSTypedStream streamExtractUsingKeyTransRecipient(
    byte[] encEnvelopedData,
    PrivateKey privateKey, X509Certificate encryptionCert)
    throws CMSException, IOException
{
    CMSEnvelopedDataParser envelopedData
                             = new CMSEnvelopedDataParser(encEnvelopedData);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientInformation recipient = recipients.get(
                                new JceKeyTransRecipientId(encryptionCert));

    if (recipient != null)
    {
        return recipient.getContentStream(
                new JceKeyTransEnvelopedRecipient(privateKey)
                    .setProvider("BC"));
    }

    throw new IllegalArgumentException("recipient for certificate not found");
}
```

While we used the `getContentStream()` method in the sample method, the reality is we could have also used `getContent()` as we did before. We have not done so in this case in order to preserve what is required to be able to process structures which contain more data than the JVM's memory can hold. Using `getContentStream()` does change the return type of the method that we are used to seeing and requires it to be processed a bit differently if we are to successfully produce our "Hello, world!" string. With the use of `getContentStream()` the following code is required:

```
    CMSTypedStream cmsContent = streamExtractUsingKeyTransRecipient(
        envEnc, recipientRsaKp.getPrivate(), recipientRsaCert);

    System.err.println(Strings.fromByteArray(
                           Streams.readAll(cmsContent.getContentStream())));
```

Note that if we had kept the use of `getContent()` the RecipientInformation object will perform the same operations as we did in the above code fragment. Note also that with the streaming model, as the unprotectedAttrs field follows the encrypted content, if you are using the streaming model you need to either fully read or fully drain the content stream first before attempting to load the contents of the unprotectedAttrs field.

S/MIME

Secure MIME, or S/MIME, provides methods for sending MIME data securely and also takes into account transmission issues that may occur with different MIME types. The current APIs in Bouncy Castle for S/MIME are based on RFC 5751 [75] which describes S/MIME version 3.2 and was published in 2010. Most commonly S/MIME is used for creating signed and/or encrypted electronic mail and there is a good chance you have already seen an S/MIME created message in your mail box already. The cryptographic side of things in S/MIME is based on Cryptographic Message Syntax, the protocol we have just been looking at.

Bouncy Castle now has two APIs for handling S/MIME: the original one, which makes use of JavaMail, and a more recent one which will parse the MIME files directly itself and does not require JavaMail. Both APIs are useful in their own way so we will look at examples from each. If you are going to use the JavaMail version, found in the bcmail-* jar files in the Bouncy Castle distribution it also helps to be familiar with JavaMail. While you can get the examples to work without a lot of familiarity with JavaMail, knowledge of the JavaMail API will mean you can apply and modify the examples given here more effectively.

S/MIME Signed Messages

There are two formats for signed messages in S/MIME. The multipart/signed message is the most common format for a signed message in S/MIME. The messages are constructed in two parts the first part being the signed content, and then a detached signature in the second part, which provides SignerInfo structures that can be used to verify the integrity of the data in the first part.

S/MIME also supports encapsulated signed data, which appears in a single body part instead. This is rarely used, mainly because it makes it impossible to read the signed data [18]: if you cannot verify the signature. The MIME type for encapsulated messsages is application/pkcs7-signature (or in some cases application/x-pkcs7-signature), named after the original standard for CMS, PKCS#7. If you run into

[18] These days there is a bit of "it seemed like a good idea at the time" about this policy. If you cannot verify the signature, you cannot tell if the signed data has been tampered with. In a lot of circumstances this really does mean you probably do not want to act on the data in any case.

one of these you will see that the message is just an ASN.1 blob usually encoded using Base64 to avoid issues over the transmission of binary data using a mail gateway that might not be quite up to the task.

Although the Bouncy Castle APIs support both these signed message types, here we will just look at the most common case multipart/signed. It is an important one to look at as it usually requires canonicalization of text data, both for creating signatures and also for verifying them. We say usually, as while regular S/MIME requires canonicalization as its default transfer encoding is 7 bit, standards like AS2, have a default transfer encoding of binary, so do not always require canonicalization.

The canonicalization process varies according to the media type, in most cases there is only one form, but if you are dealing with odd media types it is worth checking to see if a canonicalization process does apply. The most important one to be aware of for our purposes is entities with the major media type of text. In the case of data which is of a type like text/plain, the line endings in the file all need to be of the form "\r\n" and the character set used should be one of the standard IANA ones listed in the IANA "Character Sets" document [121].

Outside of canonicalization, some mail gateways do still do other strange and unexpected things to MIME messages. If you wind up with signatures constantly failing when it appears that they should work, for example you have already taken canonicalization into account, other "fun" things to look for are the removal of whitespace between the last non-blank character in a line and the line end characters, and replacement of special characters with different encodings of the same special character - this can happen where the character set supports multiple encodings for the same character. In situations like this you will find you may need to come up with some canonicalization rules of your own.

Before we start with the examples, we will need to set up some keys and a sample message.

Here is the definition of our sample message:

```
byte[] bodyPart = Strings.toByteArray(
       "Content-Type: text/plain; name=null; charset=us-ascii\r\n" +
           "Content-Transfer-Encoding: 7bit\r\n" +
           "Content-Disposition: inline; filename=test.txt\r\n" +
           "\r\n" +
           "Hello, world!\r\n");
```

The message is already canonical format, having "\r\n" as the line endings throughout.

Not surprisingly, as we are fundamentally just doing CMS, key and certificate setup is exactly the same as before:

```
KeyPair selfSignedKp = generateECKeyPair();
X509CertificateHolder selfSignedCert =
    createTrustAnchor(selfSignedKp, "SHA256withECDSA");
```

Finally, one other piece of code we will add relates to another feature of S/MIME - S/MIME capabilities. This is an extra attribute that can be included in the signed attributes contained in the SignerInfo in the detached signature associated with the signed data.

The ASN.1 definitions associated with the S/MIME capabilities attribute are as follows:

```
smimeCapabilities OBJECT IDENTIFIER ::= {iso(1) member-body(2)
        us(840) rsadsi(113549) pkcs(1) pkcs-9(9) 15}

SMIMECapability ::= SEQUENCE {
   capabilityID OBJECT IDENTIFIER,
   parameters ANY DEFINED BY capabilityID OPTIONAL }

SMIMECapabilities ::= SEQUENCE OF SMIMECapability
```

As you can see, the attribute value is just a sequence of SMIMECapability. The Bouncy Castle API provides the classes `SMIMECapabilityVector` and `SMIMECapability` to simplify the production and processing of capabilities. In this case we can produce a simple method for adding a few capabilities as follows:

```
/**
 * Basic method to provide some S/MIME capability attributes for anyone
 * responding to our message.
 *
 * @return an AttributeTable with the additional attributes.
 */
public static AttributeTable generateSMIMECapabilities()
{
    ASN1EncodableVector signedAttrs = new ASN1EncodableVector();

    SMIMECapabilityVector caps = new SMIMECapabilityVector();
    caps.addCapability(SMIMECapability.aES128_CBC);
    caps.addCapability(SMIMECapability.aES192_CBC);
    caps.addCapability(SMIMECapability.aES256_CBC);
    caps.addCapability(SMIMECapability.preferSignedData);

    signedAttrs.add(new SMIMECapabilitiesAttribute(caps));

    return new AttributeTable(signedAttrs);
}
```

The capabilities are used to tell the recipient of our message what our preferences are in terms of how we want them to respond to us. In the example code above we are signalling that we process messages encrypted using AES in CBC mode for a variety of key sizes and that we also prefer signed messages in a response. You will see in the examples that the `AttributeTable` returned by this method is added to our `SignerInfoGeneratorBuilder` so that the additional attribute for S/MIME capabilities is included in the SignerInfo structure that is generated as part of the signing process.

Now that we have some test data and a key with a certificate to use, we can now try generating some multipart/signed documents. We will look at the new, non-JavaMail API first.

Multipart/signed Using the PKIX S/MIME API

The PKIX S/MIME API is a streaming API built on top of the CMS streaming API.

Creating a multipart/signed using the PKIX S/MIME API is a two part process, the first part is to configure a `SMIMESignedWriter.Builder` that is suitable for writing the the content to be signed to, the second is then using the `SMIMESignedWriter` that was constructed to generate the multipart/signed containing the signed data and the detached signature.

Have a look at the following example code for creating a multipart/signed.

```java
/**
 * Create a multipart/signed for the body part in message.
 *
 * @param signingKey private key to generate the signature with.
 * @param signingCert public key certificate associated with the signing key.
 * @param message byte[] representing the body part to be signed.
 * @return a byte[] containing a multipart/signed MIME object.
 */
public static byte[] createSignedMultipart(
    PrivateKey signingKey,
    X509CertificateHolder signingCert,
    byte[] message)
    throws OperatorCreationException, CMSException, IOException
{
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    SMIMESignedWriter.Builder sigBldr = new SMIMESignedWriter.Builder();

    sigBldr.addCertificate(signingCert);

    sigBldr.addSignerInfoGenerator(
        new JcaSimpleSignerInfoGeneratorBuilder()
                 .setProvider("BC")
                 .setSignedAttributeGenerator(generateSMIMECapabilities())
                 .build("SHA256withECDSA", signingKey, signingCert));

    SMIMESignedWriter sigWrt = sigBldr.build(bOut);

    OutputStream out = sigWrt.getContentStream();

    out.write(message);
31
32         out.close();
33
34         return bOut.toByteArray();
35     }
```

Other than the use of the `SMIMESignedWriter` rather than a `CMSSignedDataGenerator` the data processing side of the code is almost identical to the use of the streaming CMS API we saw earlier.

On line 23 you can see where the S/MIME capabilities are added. The `setSignedAttributeGenerator()` method, in this case, does not interfere with the regular attributes for the SignerInfo, but it does prime the generator with additional attributes for inclusion when the standard signed attributes are created and hashed.

Verifying the signature and the data, on the other hand, looks a bit different. The S/MIME parsing makes use of a `MimeParser` which uses an "event" driven model. Parsing executes listener methods based on what the content headers say for the body part it is parsing.

Here is a sample verifier. It will parse a multipart/signed message, writing the content of the signed part to a passed in stream when the content containing part is encountered, and then checking the status of the signature for the data previously found when the detached signature part is encountered.

```
/**
 * Verify the signerInfo associated with signerCert, writing the content to
 * contentStream as we go.
 *
 * @param multiPart input stream containing the signed multiPart to examine.
 * @param signerCert a certificate associated with a private key that signed
 *                   the content.
 * @param contentStream output stream to receive the signed content.
 * @return true if the signed multipart verified for signerCert, false otherwise.
 */
public static boolean verifySignedMultipart(
    InputStream multiPart,
    X509CertificateHolder signerCert,
    OutputStream contentStream)
    throws IOException
{
    AtomicBoolean isVerified = new AtomicBoolean(false);

    MimeParserProvider provider = new SMimeParserProvider("7bit",
                                          new BcDigestCalculatorProvider());

    MimeParser parser = provider.createParser(multiPart);
```

```

        parser.parse(new SMimeParserListener()
        {
            public void content(
                MimeParserContext parserContext,
                Headers headers,
                InputStream inputStream)
                throws IOException
            {
                byte[] content = Streams.readAll(inputStream);

                contentStream.write(content);
            }

            public void signedData(
                MimeParserContext parserContext,
                Headers headers,
                Store certificates, Store CRLs, Store attributeCertificates,
                SignerInformationStore signers)
                throws CMSException
            {
                SignerInformation signerInfo = signers.get(
                    new SignerId(
                            signerCert.getIssuer(), signerCert.getSerialNumber()));

                try
                {
                    isVerified.set(signerInfo.verify(
                        new JcaSimpleSignerInfoVerifierBuilder()
                               .setProvider("BC").build(signerCert)));
                }
                catch (Exception e)
                {
                    throw new CMSException(
                        "unable to process signerInfo: " + e.getMessage(), e);
                }
            }
        });

        return isVerified.get();
    }
```

Notice also in this case that the SignedData structure has been deconstructed before been passed to the `signedData()` method. Unlike before, where we started with the full CMS SignedData structure and had to work through its components ourselves. Another difference is that we have our signing certificate already so it is just a matter of locating the SignerInfo that matches the `SignerId` created using the certificate.

This final piece of sample code just shows how to run both methods using our previously defined sample body part and using the private key and certificate we generated earlier.

```
byte[] data = createSignedMultipart(
                    selfSignedKp.getPrivate(), selfSignedCert, bodyPart);

ByteArrayOutputStream contentStream = new ByteArrayOutputStream();

System.out.println("verified: "+ verifySignedMultipart(
               new ByteArrayInputStream(data), selfSignedCert, contentStream));
System.out.println(Strings.fromByteArray(contentStream.toByteArray()));
```

Assuming all goes well, the code should produce the following:

```
verified: true
Hello, world!
```

Showing that the signature was successfully verified and that the content of the text/plain body part was successfully recovered as well.

Multipart/signed Using the Mail S/MIME API

The Mail S/MIME API also follows a streaming model, but in this case the parsing is handled using the JavaMail API and the "unit" message type is a MimeBodyPart class.

Here is an example method for creating a multipart/signed based the Mail S/MIME API.

```
/**
 * Create a multipart/signed for the body part in message.
 *
 * @param signingKey private key to generate the signature with.
 * @param signingCert public key certificate associated with the signing key.
 * @param message the body part to be signed.
 * @return a MimeMultipart of the type multipart/signed MIME object.
 */
public static MimeMultipart createSignedMultipart(
    PrivateKey signingKey,
    X509CertificateHolder signingCert,
    MimeBodyPart message)
    throws OperatorCreationException, SMIMEException
{
    List<X509CertificateHolder> certList = new ArrayList<>();
    certList.add(signingCert);
    Store<X509CertificateHolder> certs = new CollectionStore<>(certList);

    SMIMESignedGenerator gen = new SMIMESignedGenerator();
    gen.addSignerInfoGenerator(
        new JcaSimpleSignerInfoGeneratorBuilder()
            .setProvider("BC")
            .setSignedAttributeGenerator(generateSMIMECapabilities())
            .build("SHA256withECDSA", signingKey, signingCert));
    gen.addCertificates(certs);

    return gen.generate(message);
}
```

As you can see it looks, more or less, just like the previous one. Also like the previous one it is making use of the `generateSMIMECapabilities()` method to add the additional capabilities attribute the signed attributes in the SignerInfo which will be carried in the detached signature.

For the case of verification, the solution looks very similar to what we saw with the verification of signatures under the streaming version of the CMS API, largely because of the way the JavaMail API handles the MIME body parts internally. The following method will verify the detached signature in the `MimeMultipart` we generated in the previous `createSignedMultipart()` method.

```
/**
 * Verify a MimeMultipart containing a multipart/signed object.
 *
 * @param signedMessage the multipart/signed.
 * @param signerCert the certificate of one of the signers of signedMessage.
 * @return true if the multipart/signed verified for signerCert, false otherwise.
 */
public static boolean verifySignedMultipart(
    MimeMultipart signedMessage,
    X509CertificateHolder signerCert)
    throws GeneralSecurityException, OperatorCreationException,
           CMSException, MessagingException
{
    SMIMESigned signedData = new SMIMESigned(signedMessage);

    SignerInformationStore signers = signedData.getSignerInfos();

    SignerInformation signer = signers.get(
        new SignerId(signerCert.getIssuer(),
                         signerCert.getSerialNumber()));

    return signer.verify(new JcaSimpleSignerInfoVerifierBuilder()
                               .setProvider("BC").build(signerCert));
}
```

Here is some sample code which will allow you to put the two methods together:

```
MimeMultipart signedMM = createSignedMultipart(selfSignedKp.getPrivate(),
    selfSignedCert, new MimeBodyPart(new ByteArrayInputStream(bodyPart)));

System.out.println("verified: "+ verifySignedMultipart(
               signedMM, selfSignedCert));
System.out.println(signedMM.getBodyPart(0).getContent());
```

Note the only real difference between this and our previous S/MIME example is the use of the JavaMail class types as return values and also as input.

The sample code should produce exactly the same output as our previous example case, as in:

```
verified: true
Hello, world!
```

also showing that the SignerInfo for `signerCert` verified correctly and the content of the signed body part was recovered.

S/MIME Enveloped Messages

S/MIME Enveloped messages always carry the message as encapsulated data and have the MIME type application/pkcs7-mime, or the older application/x-pkcs7-mime. The content of the S/MIME message itself is just an encoding of a ContentInfo holding a CMS EnvelopedData structure.

S/MIME Enveloped messages can use any of the CMS Enveloped recipient types. We will just look at one example for the sake of brevity here, but if you look at the instructions for different recipients that were laid out in the section on CMS you will find you can do the same for S/MIME Enveloped recipients as well.

Enveloped Messages Using the PKIX S/MIME API

Creating a S/MIME Enveloped message using the PKIX S/MIME API is similar to creating a signed message. The first step is to configure an `SMIMEEnvelopedWriter.Builder` with the recipients that are meant to be able to decrypt the message and the second step is to use the builder to create an `SMIMEEnvelopedWriter` which is then used to output the S/MIME enveloped message by having the content fed through it.

The following example method shows how to create an S/MIME enveloped message using RSA OAEP for key transport AES-256 CBC mode for the message encryption.

```
/**
 * Create an application/pkcs7-mime for the body part in message.
 *
 * @param encryptionCert public key certificate associated with the
 *                       intended recipient.
 * @param message byte[] representing the body part to be encrypted.
 * @return a byte[] containing a application/pkcs7-mime MIME object.
 */
public static byte[] createEnveloped(
    X509Certificate encryptionCert,
    byte[] message)
    throws GeneralSecurityException, CMSException, IOException
{
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    SMIMEEnvelopedWriter.Builder envBldr = new SMIMEEnvelopedWriter.Builder();

    JcaAlgorithmParametersConverter paramsConverter =
                                        new JcaAlgorithmParametersConverter();

    AlgorithmIdentifier oaepParams = paramsConverter.getAlgorithmIdentifier(
        PKCSObjectIdentifiers.id_RSAES_OAEP,
        new OAEPParameterSpec("SHA-256",
            "MGF1", new MGF1ParameterSpec("SHA-256"),
            PSource.PSpecified.DEFAULT));

    envBldr.addRecipientInfoGenerator(
        new JceKeyTransRecipientInfoGenerator(
            encryptionCert, oaepParams).setProvider("BC"));

    SMIMEEnvelopedWriter sigWrt = envBldr.build(bOut,
                new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                    .setProvider("BC")
                    .build());

    OutputStream out = sigWrt.getContentStream();

    out.write(message);

    out.close();

    return bOut.toByteArray();
}
```

If you look back at the example in CMS, which is in the section on “ktri - the KeyTransRecipientInfo Choice”, you will see that the setup of the recipient generator is identical.

Decryption of a message follows the same listener style model that we saw with signature verification. We set up a parser for the message and provide a listener which is invoked when the MIME type we are looking for is encountered. In the case of enveloped data the call back also gives us the originator information and the recipients for the message that was found. We use those to extract the data.

The following example method shows how to recover the plain text from an S/MIME Enveloped message, assuming that the appropriate recipient is in place.

```
/**
 * Decrypt the encrypted content in an application/pkcs7-mime message, writing
 * the content to contentStream as we go.
 *
 * @param encryptedPart input stream containing the enveloped Part to examine.
 * @param recipientCert a certificate associated with a private key that the
 *                      content was encrypted for.
 * @param recipientKey the private key that the content was encrypted for.
 * @param contentStream output stream to receive the decrypted content.
 */
public static void decryptEnveloped(
    InputStream encryptedPart,
    X509Certificate recipientCert,
    PrivateKey recipientKey,
    OutputStream contentStream)
    throws IOException
{
    MimeParserProvider provider = new SMimeParserProvider("7bit",
                                          new BcDigestCalculatorProvider());

    MimeParser parser = provider.createParser(encryptedPart);

    parser.parse(new SMimeParserListener()
    {
        public void envelopedData(
            MimeParserContext parserContext,
            Headers headers,
            OriginatorInformation originator,
            RecipientInformationStore recipients)
            throws IOException, CMSException
        {
            RecipientInformation recipInfo = recipients.get(
                                new JceKeyTransRecipientId(recipientCert));

            byte[] content = recipInfo.getContent(
                            new JceKeyTransEnvelopedRecipient(recipientKey));

            contentStream.write(content);
        }
    });

    contentStream.close();
}
```

Note that, while the example makes use of `RecipientInformation.getContent()`, it is still possible to make use of `RecipientInformation.getContentStream()` to recover large messages instead.

To make use of the methods, some setup is required. We need to have certificates in place as well as an associated private key for managing the decryption. The code fragment that follows provides the minimal setup required to make use of the methods.

```
byte[] bodyPart = Strings.toByteArray(
       "Content-Type: text/plain; name=null; charset=us-ascii\r\n" +
           "Content-Transfer-Encoding: 7bit\r\n" +
           "Content-Disposition: inline; filename=test.txt\r\n" +
           "\r\n" +
           "Hello, world!\r\n");

KeyPair selfSignedKp = generateRSAKeyPair();
X509CertificateHolder selfSignedCertHldr =
    createTrustAnchor(selfSignedKp, "SHA256withRSA");

X509Certificate selfSignedCert = new JcaX509CertificateConverter()
                    .setProvider("BC").getCertificate(selfSignedCertHldr);
```

With the setup code above in place, the following example code will create an S/MIME Enveloped message using the `createEnveloped()` method and then decrypt the result using the `decryptEnveloped()`

method echoing the recovered message to standard out.

```
byte[] data = createEnveloped(
                 selfSignedCert, bodyPart);

ByteArrayOutputStream contentStream = new ByteArrayOutputStream();

decryptEnveloped(
               new ByteArrayInputStream(data),
                selfSignedCert, selfSignedKp.getPrivate(),
                contentStream);

System.out.println(Strings.fromByteArray(contentStream.toByteArray()));
```

Assuming everything is working, the sample code will produce the following output.

```
Content-Type: text/plain; name=null; charset=us-ascii
Content-Transfer-Encoding: 7bit
Content-Disposition: inline; filename=test.txt

Hello, world!
```

showing that the original body part was fully recovered.

Enveloped Messages Using the Mail S/MIME API

While the SMIMEEnvelopedGenerator is also stream based, the streaming is largely managed internally to the API as it makes use of the JavaMail.

Here is the equivalent createEnveloped() method to the one we looked at earlier from the PKIX API, but written using the BC Mail API instead:

```
/**
 * Create an application/pkcs7-mime from the body part in message.
 *
 * @param encryptionCert public key certificate associated with the
 *                       intended recipient.
 * @param message byte[] representing the body part to be encrypted.
 * @return a MimeBodyPart containing a application/pkcs7-mime MIME object.
 */
public static MimeBodyPart createEnveloped(
    X509Certificate encryptionCert,
    MimeBodyPart message)
    throws GeneralSecurityException, CMSException, SMIMEException
{
    SMIMEEnvelopedGenerator envGen = new SMIMEEnvelopedGenerator();

    JcaAlgorithmParametersConverter paramsConverter =
                                     new JcaAlgorithmParametersConverter();

    AlgorithmIdentifier oaepParams = paramsConverter.getAlgorithmIdentifier(
        PKCSObjectIdentifiers.id_RSAES_OAEP,
        new OAEPParameterSpec("SHA-256",
            "MGF1", new MGF1ParameterSpec("SHA-256"),
            PSource.PSpecified.DEFAULT));

    envGen.addRecipientInfoGenerator(
        new JceKeyTransRecipientInfoGenerator(
            encryptionCert, oaepParams).setProvider("BC"));

    return envGen.generate(message,
                new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                    .setProvider("BC")
                    .build());
}
```

The only meaningful difference is the `generate()` method which takes `message` (the source object for the plain text) and then internally feeds the encrypted content into a `MimeBodyPart` object so that the Enveloped Data is ready for use with the rest of the JavaMail API.

A similar mode of working applies to the decryption of S/MIME Enveloped messages using the BC Mail API as well, as the following example method shows.

```
/**
 * Extract a MIME body part from an enveloped message.
 *
 * @param encryptedMessage the enveloped message.
 * @param recipientCert the certificate associated with the recipient key.
 * @param recipientKey the private key to recover the body part with.
 */
public static MimeBodyPart decryptEnveloped(
    MimeBodyPart encryptedMessage,
    X509Certificate recipientCert,
    PrivateKey recipientKey)
    throws
    CMSException, MessagingException, SMIMEException
{
    SMIMEEnveloped envData = new SMIMEEnveloped(encryptedMessage);

    RecipientInformationStore recipients = envData.getRecipientInfos();

    RecipientInformation recipient = recipients.get(
        new JceKeyTransRecipientId(recipientCert));

    return SMIMEUtil.toMimeBodyPart(recipient.getContent(
        new JceKeyTransEnvelopedRecipient(recipientKey).setProvider("BC")));
}
```

Note that the utility method 'SMIMEUtil.toMimeBodyPart()' is required as the `RecipientInformation` class does not have direct support for JavaMail. If you were simply interested in getting back the content without repacking the data for JavaMail you could avoid the call.

Setup in this case is the same as it is for the PKIX S/MIME API. The difference, of course, comes in the invocation of the methods, as rather than just dealing with streams directly the JavaMail API requires you to present the MIME objects using JavaMail's pre-defined types.

You can see this in the following code for calling the methods - we need to wrap `bodyPart` in a `MimeBodyPart` object for processing.

```
MimeBodyPart signedMM = createEnveloped(
    selfSignedCert, new MimeBodyPart(new ByteArrayInputStream(bodyPart)));

BodyPart decPart = decryptEnveloped(
                      signedMM, selfSignedCert, selfSignedKp.getPrivate());

System.out.println(decPart.getContent());
```

In the case of the example, the other change we have made is to grab content of the encrypted body part for output as well by using the `getContent()` method. As this is the case, when the code, you will see the following output:

```
Hello, world!
```

Showing that the encrypted body part and its content was successfully recovered.

Things to Watch With JavaMail

We have found the following operation has caused enough issues for people that it deserves special mention.

Handling "multipart/mixed" Content

One common source of confusion is how to handle multipart/mixed data. In this case it is necessary to wrap the multipart content in a single MimeBodyPart before passing it to the S/MIME API.

Time-Stamp Protocol

Time-Stamp Protocol (TSP) was originally defined in RFC 3161 [51] with a subsequent update to allow for the use of message digests other than SHA-1 coming out in RFC 5816 [78]. The basic concept behind it is that a Time Stamping Authority (TSA) issues cryptographically secure time-stamps in response to time-stamp requests which consist of hashes, usually representing some document.

Time-Stamp Protocol Requests

A time-stamp request is defined by the TimeStampReq structure defined in RFC 3161. The ASN.1 definition for TimeStampReq and its dependent structures is described as follows:

```
TimeStampReq ::= SEQUENCE  {
    version              INTEGER  { v1(1) },
    messageImprint       MessageImprint,
    reqPolicy            TSAPolicyId              OPTIONAL,
    nonce                INTEGER                  OPTIONAL,
    certReq              BOOLEAN                  DEFAULT FALSE,
    extensions           [0] IMPLICIT Extensions  OPTIONAL  }

MessageImprint ::= SEQUENCE  {
    hashAlgorithm        AlgorithmIdentifier,
    hashedMessage        OCTET STRING  }

TSAPolicyId ::= OBJECT IDENTIFIER
```

As you can see the simplest version of a TimeStampReq just contains the version number and a MessageImprint field. The following example method creates a basic time-stamp request using SHA-256 (so based on RFC 5816) with just the mandatory fields set:

```java
/**
 * Create a simple TSP request for the passed in SHA-256 hash.
 *
 * @param data the data that we want timestamped.
 * @return a DER encoding of the resulting TSP request.
 */
public static byte[] createTspRequest(byte[] data)
    throws IOException, NoSuchAlgorithmException
{
    MessageDigest digest = MessageDigest.getInstance("SHA-256");

    TimeStampRequestGenerator reqGen = new TimeStampRequestGenerator();

    return reqGen.generate(TSPAlgorithms.SHA256,
                                    digest.digest(data)).getEncoded();
}
```

The optional fields are all used to provide extra information in terms of what the requester of the time-stamp would like. The reqPolicy field is to request that the TSA issues the time-stamp response configured with a specific policy, the nonce allows for the requester to match the response to a specific request without fear of a repeat[19], the certReq field is used to request the server to include its signing certificate (and possibly others), and, finally, the extensions field can be used to add extensions (if defined). Extensions would normally be defined according to what is accepted by the server. The other thing to be aware of is that when an extension is marked as critical and the server cannot interpret it, the server will fail the request.

The following sample method allows for the setting of the nonce and the certReq field.

```java
/**
 * Create a TSP request for the passed in SHA-256 hash which also
 * includes a nonce and, possibly, a request for certificates.
 *
 * @param data the data we want time-stamped.
 * @param nonce a nonce associated with this request.
 * @param requestTsaCert if true, authority should send back a
 *                       copy of its certificate.
 * @return a DER encoding of the resulting TSP request.
 */
public static byte[] createNoncedTspRequest(
    byte[] data, BigInteger nonce, boolean requestTsaCert)
    throws IOException, NoSuchAlgorithmException
{
```

[19] Note with some servers setting the nonce field is mandatory.

```
        TimeStampRequestGenerator reqGen = new TimeStampRequestGenerator();

        reqGen.setCertReq(requestTsaCert);

        MessageDigest digest = MessageDigest.getInstance("SHA-256");

        return reqGen.generate(
                 TSPAlgorithms.SHA256, digest.digest(data), nonce).getEncoded();
    }
```

As you can see the nonce is an extra parameter on the generate request as the nonce value should only be used once.

Time-Stamp Protocol Responses

The minimal requirements for generating a TSP response are the TSA's private key, a serial number, and the the policy identifier to show what policy the TSA created the TSP token under. This is usually server specific and issued under an OID branch controlled by the server's owners. The policy oid will normally be associated with an actual document that details how the TSP server will deal with a particular time-stamp request. RFC 3628 [55] provides requirements for a baseline time-stamp policy for a TSP server. As we saw earlier, the time-stamp request might also include information which requests the policy that the time-stamp should be provided under in the reqPolicy field.

The time-stamp response has the following ASN.1 definition:

```
TimeStampResp ::= SEQUENCE  {
    status                  PKIStatusInfo,
    timeStampToken          TimeStampToken     OPTIONAL  }
```

The first part of the response, the status field, indicates the success or failure of the request. The second part, the timeStampToken, is only set where the request has been successful.

PKIStatusInfo is also a sequence and has some optional fields that can be set if the request has been rejected by the server.

```
   PKIStatusInfo ::= SEQUENCE {
      status        PKIStatus,
      statusString  PKIFreeText     OPTIONAL,
      failInfo      PKIFailureInfo  OPTIONAL  }
```

PKIFreeText is simply a SEQUENCE of UTF8String.

PKIStatus is defined as an INTEGER with 6 possible values. The possible values, and their associated meanings are:

- 1 - granted
- 2 - grantedWithMods
- 3 - rejection
- 4 - waiting
- 5 - revocationWarning
- 6 - revocation

Where the timeStampToken field is not set (in the case where status is not 1 or 2), the failInfo field will hold the definitive information as to why a request was rejected. Unlike PKIStatus, PKIFailureInfo is actually a BIT STRING not an INTEGER and has the following ASN.1 definition:

```
PKIFailureInfo ::= BIT STRING {
    badAlg               (0),
    badRequest           (2),
    badDataFormat        (5),
    timeNotAvailable    (14),
    unacceptedPolicy    (15),
    unacceptedExtension (16),
    addInfoNotAvailable (17),
    systemFailure       (25) }
```

The meanings of the different BIT STRING values are given in RFC 3161 as:

- badAlg: messageImprint contains an unrecognized, or unsupported, AlgorithmIdentifier;
- badRequest: the request is not permitted or supported;
- badDataFormat: the request has been submitted in the wrong format;
- timeNotAvailable: the TSA's time source is not available;
- unacceptedPolicy: the requested TSA policy is not supported by the TSA;
- unacceptedExtension: an extension in the request (usually a critical one) is not supported by the TSA;
- addInfoNotAvailable: any requests for additional information with time-stamp could not be understood, or the information is not available;
- systemFailure: the request cannot be handled due to a failure in the TSA's system.

Where the request message has the CertReq field set to true a conforming TSA must always at least provide a certificate that can be used to verify the time-stamp signature. Other certificates may also be present. When the CertReq field is not set a conforming TSA will not include any certificates in the response.

Where a request is successful the response has the timeStampToken field set. The TimeStampToken structure is actually a CMS ContentInfo field with the type id-signedData. The actual time-stamp information in the response appears as encapsulated data in the eContent of the SignedData in the ContentInfo. The structure that contains the time-stamp information is a TSTInfo structure and its presence in the eContent is indicated by setting eContentType to id-ct-TSTInfo, which is defined as:

```
id-ct-TSTInfo  OBJECT IDENTIFIER ::= {
 iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) ct(1) 4
}
```

The TSTInfo structure, with an additional structure for time-stamp accuracy that TSTInfo uses, is described by the following ASN.1 definitions:

```
TSTInfo ::= SEQUENCE  {
    version                      INTEGER  { v1(1) },
    policy                       TSAPolicyId,
    messageImprint               MessageImprint,
    serialNumber                 INTEGER,
    genTime                      GeneralizedTime,
    accuracy                     Accuracy                 OPTIONAL,
    ordering                     BOOLEAN             DEFAULT FALSE,
    nonce                        INTEGER                  OPTIONAL,
    tsa                          [0] GeneralName          OPTIONAL,
    extensions                   [1] IMPLICIT Extensions  OPTIONAL   }

Accuracy ::= SEQUENCE {
                seconds        INTEGER           OPTIONAL,
                millis     [0] INTEGER  (1..999) OPTIONAL,
                micros     [1] INTEGER  (1..999) OPTIONAL  }
```

As you would expect the messageImprint field, as well as the nonce field if the same was present in the request, in the TSTInfo structure in the response must echo the contents of the equivalent field in the request. The serialNumber represents an INTEGER issued by the server which is unique to the response.

Time-stamp responses must include at least seconds, although they may go further, and will always be expressed as UTC times. Where the accuracy field is set it can be used to calculate the upper (by adding accuracy to genTime) and the lower (by subtracting accuracy from genTime) limits on what time range the time-stamp represents. Any missing field in the Accuracy structure should be treated as the value zero.

In Bouncy Castle most of the work around creating TSTInfo structures is done within the API, so the generation of a response is mainly concerned with providing a signer and the request parameters. The following sample method provides an example of a time-stamp response generator which can handle requests structured using both the request generation methods we have looked at.

```
/**
 * Create a TSP response for the passed in byte encoded TSP request.
 *
 * @param tsaSigningKey  TSA signing key.
 * @param tsaSigningCert copy of TSA verification certificate.
 * @param serialNumber   response serial number
 * @param tsaPolicy      time stamp
 * @param encRequest     byte encoding of the TSP request.
 */
public static byte[] createTspResponse(
    PrivateKey tsaSigningKey, X509Certificate tsaSigningCert,
    BigInteger serialNumber, ASN1ObjectIdentifier tsaPolicy, byte[] encRequest)
    throws TSPException, OperatorCreationException,
    GeneralSecurityException, IOException
{
    AlgorithmIdentifier digestAlgorithm =
        new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256);
    DigestCalculatorProvider digProvider =
        new JcaDigestCalculatorProviderBuilder().setProvider("BC").build();

    TimeStampRequest tspRequest = new TimeStampRequest(encRequest);

    SignerInfoGenerator tsaSigner = new JcaSimpleSignerInfoGeneratorBuilder()
        .build("SHA256withECDSA", tsaSigningKey, tsaSigningCert);
    TimeStampTokenGenerator tsTokenGen = new TimeStampTokenGenerator(
        tsaSigner, digProvider.get(digestAlgorithm), tsaPolicy);

    // client has requested a copy of the signing certificate
    if (tspRequest.getCertReq())
    {
        tsTokenGen.addCertificates(
            new JcaCertStore(Collections.singleton(tsaSigningCert)));
    }

    TimeStampResponseGenerator tsRespGen = new TimeStampResponseGenerator(
        tsTokenGen, TSPAlgorithms.ALLOWED);

    return tsRespGen.generate(tspRequest, serialNumber, new Date()).getEncoded();
}
```

You will notice in the sample code that, while we explicitly check for and handle the CertReq field in the request by calling the `getCertReq()` method, there is no mention of the nonce, even though our createNoncedTspRequest() method will provide one in the request it generates. The reason for this is that, as there are defined rules about the nonce, the nonce is handled during the final building of the `TimeStampToken` that takes place inside the `generate()` method of the `TimeStampResponseGenerator` class.

The following method verifies a time-stamp response, returning true on success, throwing an exception otherwise.

```
/**
 * Verify that the passed in response can be verified by the TSA certificate and is
 * appropriate to the time-stamp request encoded in encRequest.
 *
 * @param encResponse an ASN.1 binary encoding of a time-stamp response.
 * @param tsaCertificate the certificate for the TSA that generated the response.
 * @param encRequest an ASN.1 binary encoding of the originating request.
 * @return true if the response can be verified, an exception is thrown if
 * validation fails.
 */
public static boolean verifyTspResponse(
    byte[] encResponse,
    X509Certificate tsaCertificate,
    byte[] encRequest)
    throws IOException, TSPException, OperatorCreationException
{
    TimeStampResponse tspResp = new TimeStampResponse(encResponse);

    // response/request validate will throw an exception if there is an issue
    tspResp.validate(new TimeStampRequest(encRequest));

    TimeStampToken tsToken = tspResp.getTimeStampToken();

    // token signature validate will throw an exception if there is an issue
    tsToken.validate(new JcaSimpleSignerInfoVerifierBuilder()
        .setProvider("BC")
        .build(tsaCertificate));

    return true;
}
```

The validate method is quite short as the TSP API checks the criteria given in RFC 3161 to ensure that the time-stamp response is properly created. From a developer's perspective it is just a matter of putting together the `SignerInformationVerifier` object that is required to verify the signature. Note that part of the criteria given in RFC 3161 is that the certificate used to verify the TSP request must include an ExtendedKeyUsage extension which is marked as critical and asserts id-kp-timeStamping as the sole key purpose.

All that remains now is to look at a full working example of use. We will do that in the next section.

An Application of Time-Stamps: CAdES-T

Initially RFC 3126 "Electronic Signature Formats for long term electronic signatures" [50], provided a number of ways to incorporate the use of time-stamps into document signing. RFC 5126 "CMS Advanced Electronic Signatures (CAdES)" [68] built on this further to provide a means of specifying long-life digital signatures in conjunction with the use of CMS [74].

The two attributes introduced in RFC 5652 for using time-stamps in CMS signatures are in section 5.11.4, the content-time-stamp attribute, and in section 6.1.1 which covers the signature-time-stamp. A content-time-stamp is an optional CMS signed attribute which contains a time-stamp of the data that was the subject of CMS signing. We will have a closer look at the signature-time-stamp attribute here as it provides a means of incorporating a time-stamp into a CMS signature after creation. In this case a time-stamp is added to one, or more, of the SignerInfo structures included in the CMS SignedData structure as an unsigned attribute.

The first step in this process is create the appropriate TSP request. The following method shows how to create a TSP request using the methods we have already looked at for the first SignerInfo structure in a CMS SignedData structure.

```
/**
 * Create a TSP request for the first SignerInfo in a SignedData structure.
 *
 * @param nonce nonce to use with the request.
 * @param encSignedData the encoding of the SignedData structure.
 * @return a TSP request.
 */
public static byte[] createSignerInfoTspRequest(
    BigInteger nonce,
    byte[] encSignedData)
    throws GeneralSecurityException,
           CMSException, IOException
{
    CMSSignedData signedData = new CMSSignedData(encSignedData);
    SignerInformation signer = signedData.getSignerInfos().iterator().next();

    return createNoncedTspRequest(signer.getSignature(), nonce, true);
}
```

Note that the TSP request is created using the contents of the signature stored in the SignerInfo structure represented by the `SignerInformation` object that was retrieved from the `CMSSignedData` object representing the SignedData structure we are adding the time-stamp too.

Once the TSP request has been processed and the response returned, the next step in the process is to create a CMS Attribute structure which is then added to the unsigned attributes for the CMS SignerInfo structure whose signature was the source of the original request.

The following method shows how to create the correct CMS attribute for inclusion in the SignerInfo structure:

```java
/**
 * Create a CMS attribute for carrying a TSP response.
 *
 * @param tspResponse response from the time-stamp authority we want to reference.
 * @return a suitable attribute for a TSP response.
 */
public static Attribute createTspAttribute(byte[] tspResponse)
    throws TSPException, IOException
{
    TimeStampResponse response = new TimeStampResponse(tspResponse);
    TimeStampToken timestampToken = response.getTimeStampToken();

    return new Attribute(PKCSObjectIdentifiers.id_aa_signatureTimeStampToken,
                   new DERSet(timestampToken.toCMSSignedData().toASN1Structure()));
}
```

And this example method shows how to put everything together, producing a CMS SignedData structure which has had its first SignerInfo structure edited to include a time-stamp over it:

```java
/**
 * Replace the SignerInformation in the CMS SignedData represented by
 * encSignedData with a time-stamped version.
 *
 * @param encSignedData encoding of the SignedData to be time-stamped.
 * @param nonce nonce for the TSP request.
 * @param tsaSigningKey time-stamp authority signing key.
 * @param tsaSigningCert time-stamp authority certificate.
 * @param serialNumber serial number for time-stamp response
 * @param tsaPolicyOID policy under which time-stamp authority is working.
 * @return encoding of a new version of encSignedData with a time-stamp.
 */
public static byte[] createTimeStampedSignedData(
    byte[] encSignedData,
    BigInteger nonce,
    PrivateKey tsaSigningKey, X509Certificate tsaSigningCert,
    BigInteger serialNumber, ASN1ObjectIdentifier tsaPolicyOID)
```

```
        throws OperatorCreationException,
               GeneralSecurityException,
               CMSException, TSPException, IOException
    {
        CMSSignedData signedData = new CMSSignedData(encSignedData);
        SignerInformation signer = signedData.getSignerInfos().iterator().next();

        byte[] tspReq = createSignerInfoTspRequest(nonce, encSignedData);

        byte[] tspResp = createTspResponse(tsaSigningKey, tsaSigningCert,
                                            serialNumber, tsaPolicyOID, tspReq);

        // check the response
        verifyTspResponse(tspResp, tsaSigningCert, tspReq);

        // set up attribute table with TSP attribute in it.
        ASN1EncodableVector timestampVector = new ASN1EncodableVector();

        timestampVector.add(createTspAttribute(tspResp));

        AttributeTable at = new AttributeTable(timestampVector);

        // create replacement signer
        signer = SignerInformation.replaceUnsignedAttributes(signer, at);

        // create replacement SignerStore
        SignerInformationStore newSignerStore = new SignerInformationStore(signer);

        // replace the signers in the signed data object
        return CMSSignedData.replaceSigners(signedData, newSignerStore).getEncoded();
    }
```

Having created a time-stamped SignerInfo structure we will now look at the process required to verify the time-stamp at a later date. Doing this requires rebuilding the `TimeStampToken` from the appropriate attribute in the unsigned attributes for the SignerInfo structure.

```
/**
 * Verify the time-stamp token attribute on the passed in CMS SignedData.
 * Note: in this case we are using the certificate stored in the TimeStampToken
 * associated with the SignedData.
 *
 * @param cmsSignedData encoding of the CMS SignedData we want to check.
 * @return true if verifies, exception thrown otherwise.
 */
public static boolean verifyTimeStampedSigner(byte[] cmsSignedData)
    throws CMSException,
           IOException, TSPException,
           CertificateException, OperatorCreationException
{
    CMSSignedData signedData = new CMSSignedData(cmsSignedData);
    SignerInformation signer = signedData.getSignerInfos().iterator().next();

    TimeStampToken tspToken = new TimeStampToken(
        ContentInfo.getInstance(
            signer.getUnsignedAttributes()
                .get(PKCSObjectIdentifiers.id_aa_signatureTimeStampToken)
                    .getAttributeValues()[0]));

    Collection certCollection = tspToken.getCertificates()
                                    .getMatches(tspToken.getSID());
    Iterator certIt = certCollection.iterator();
    X509CertificateHolder cert = (X509CertificateHolder)certIt.next();

    // this method throws an exception if validation fails.
    tspToken.validate(new JcaSimpleSignerInfoVerifierBuilder()
                                .setProvider("BC").build(cert));

    return true;
}
```

Note you will not always be able to rely on the `TimeStampToken` including the verification certificate. Naturally, you would want to make sure the verification certificate is valid as well. It is not a good idea to accept these things automatically at face value.

This final piece of example code shows how to put all the above together, and produce a time-stamp SignedData structure from one that was created using the CMS methods we looked at earlier as well.

```
        byte[] msg = Strings.toByteArray("Hello, world!");

        KeyPair selfSignedKp = generateECKeyPair();
        X509CertificateHolder selfSignedCert =
            createTrustAnchor(selfSignedKp, "SHA256withECDSA");

        CMSSignedData encapSigned = createSignedData(selfSignedKp.getPrivate(),
            selfSignedCert, msg, true);

        verifySignedEncapsulated(encapSigned.getEncoded());

        CMSTypedData cmsData = encapSigned.getSignedContent();
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();

        cmsData.write(bOut);

        System.out.println(Strings.fromByteArray(bOut.toByteArray()));

        JcaX509CertificateConverter certConverter =
            new JcaX509CertificateConverter().setProvider("BC");

        KeyPair tspKp = generateECKeyPair();

        X509Certificate tspCert =
            certConverter.getCertificate(
                createSpecialPurposeEndEntity(
                    selfSignedCert, selfSignedKp.getPrivate(), "SHA256withECDSA",
                    tspKp.getPublic(), KeyPurposeId.id_kp_timeStamping));

        ASN1ObjectIdentifier madeUpTsaPolicyOID = new ASN1ObjectIdentifier("1.1.1");

        byte[] encStampedSignedData = createTimeStampedSignedData(
            encapSigned.getEncoded(),
            BigInteger.valueOf(System.currentTimeMillis()),
            tspKp.getPrivate(), tspCert,
            BigInteger.ONE, madeUpTsaPolicyOID);

        System.out.println("TSP verified: "
            + verifyTimeStampedSigner(encStampedSignedData));
```

You can see that we were unable to come up with a general use OID for the TSA policy and instead had to use `madeUpTsaPolicyOID` - the OID's you will see in time-stamp responses will vary, but they will refer to the security policy of the TSA that issued the time-stamp. You want to check the OID value in normal use to make sure the security policy you expected was the one under which the TSA issued you the time-stamp.

Keep in mind also that the certificate used for verifying the TSP response needs to include an ExtendedKeyUsage extension which is marked as critical and assert the key purpose OID value id-kp-timeStamping. Verification will fail otherwise. You can see where this has been dealt with on line 26 with the use of the `createSpecialPurposeEndEntity()` method - although in real life you would not be using your CMS signing key to be issuing a time-stamp verification certificate.

Assuming all else goes well, running the code will produce:

```
sig verified: true
Hello, world!
TSP verified: true
```

Indicating that the data was signed correctly and that the signature-time-stamp attribute was added correctly and able to be verified as well.

Summary

In this chapter we have examined a set of cryptographic protocols that are the work horses of many standards on the Internet: CMS, S/MIME, and TSP.

We have looked at ways of signing and encrypting data under CMS and S/MIME, as well as the different Bouncy Castle APIs for supporting the procotols, including the JavaMail based BC API for dealing with S/MIME. We also looked at TSP as a method for creating long-life signatures by itself, as well as a way to use TSP to augment CMS signing.

Finally in the CAdES-T example, we covered the process of adding a time-stamp to an already generated CMS SignedData structure as a means of supporting the longevity of the signatures included in the SignedData structure.

Chapter 12: Certification Requests and Certificate Management

As we have already seen CAs do not just generate certificates from nothing - certificate generation is often carried out in response to a certification request. Certification requests are often remote, and as a result also protect the users private key from any abuse by a CA that turns out to be rogue or, in some way, careless.

That said, while the intention of certification requests is clear, what should go in one is not so clear and consequently a number of protocols have been proposed for sending messages to a CA for the purpose of generating certificates. In this chapter, we will look at how certification request messages are put together and what protocols they are used with. This will help us understand the request/response cycle that a client and a CA need to go through to process a certification request and get a certificate from the CA back to the client.

PKCS #10 Certification Requests

PKCS #10 takes its name from the original RSA standard. The current definition for a PKCS #10 certificate requests is now described in RFC 2986 [49] with a description of its associated MIME type in RFC 5967 [81].

In Bouncy Castle you can create and process PKCS #10 certificate requests using either the `PKCS10CertificationReque` and the `PKCS10CertificationRequest` classes which can be found in the `org.bouncycastle.pkcs` package, or classes extended from them such as the `JcaPKCS10CertificationRequestBuilder` and the `JcaPKCS10CertificationRequest`. As with other cases in the PKIX APIs the classes that start with 'Jca' are aware of the JCA types, such as `PublicKey`, and can be configured with JCA providers as appropriate.

At its most basic level a PKCS #10 request is just a signing of a structure containing the subject name you wish to associate with the public key in the certification request. RFC 2986 describes the structure in ASN.1 as follows:

```
CertificationRequestInfo ::= SEQUENCE {
     version       INTEGER { v1(0) } (v1,...),
     subject       Name,
     subjectPKInfo SubjectPublicKeyInfo{{ PKInfoAlgorithms }},
     attributes    [0] Attributes{{ CRIAttributes }}
}

CertificationRequest ::= SEQUENCE {
     certificationRequestInfo CertificationRequestInfo,
     signatureAlgorithm AlgorithmIdentifier{{ SignatureAlgorithms }},
     signature          BIT STRING
}
```

Definitions are provided for SubjectPublicKeyInfo amd Attributes as well, but these structures are the same as the ones we have already encountered with certificates and attribute certificates in Chapter 8. You can see from the ASN.1 that the way the CertificationRequest structure is put together is essentially the same as the method used to construct the ASN.1 Certificate structure.

The signature on the PKCS #10 request is accepted by a CA as it also provides the proof-of-possession (POP) to show that the certification request is from someone authorised to request the association with the subject name. The POP is provided by generating the signature using the private key associated with the public key in the request - establishing that the request was created by someone who had to know the private key. This does mean that PKCS #10 requests, unlike those defined in RFC 4211 [59], can only be used with keys that can also be used to generate signatures. This also means that if the public key is supposed to be used for encryption, in a FIPS context, configuration of the provider is required to allow the dual use for the purpose of signing the certification request.

You can see an example of how you would build a basic certification request under the JCA in the code fragment below:

```java
/**
 * Create a basic PKCS#10 request.
 *
 * @param keyPair the key pair the certification request is for.
 * @param sigAlg the signature algorithm to sign the PKCS#10 request with.
 * @return an object carrying the PKCS#10 request.
 * @throws OperatorCreationException in case the private key is
 * inappropriate for signature algorithm selected.
 */
public static PKCS10CertificationRequest createPKCS10(
        KeyPair keyPair, String sigAlg)
        throws OperatorCreationException
{
    X500NameBuilder x500NameBld = new X500NameBuilder(BCStyle.INSTANCE)
```

```
            .addRDN(BCStyle.C, "AU")
            .addRDN(BCStyle.ST, "Victoria")
            .addRDN(BCStyle.L, "Melbourne")
            .addRDN(BCStyle.O, "The Legion of the Bouncy Castle");

    X500Name subject = x500NameBld.build();

    PKCS10CertificationRequestBuilder requestBuilder
            = new JcaPKCS10CertificationRequestBuilder(
                                        subject, keyPair.getPublic());

    ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
                            .setProvider("BC").build(keyPair.getPrivate());

    return requestBuilder.build(signer);
}
```

On the CA side you need to be able to validate the certificate on the request. The following method does this under the JCA:

```
/**
 * Simple method to check the signature on a PKCS#10 certification test with
 * a public key.
 *
 * @param request the encoding of the PKCS#10 request of interest.
 * @return true if the public key verifies the signature, false otherwise.
 * @throws OperatorCreationException in case the public key is unsuitable
 * @throws PKCSException if the PKCS#10 request cannot be processed.
 */
public static boolean isValidPKCS10Request(
        byte[] request)
    throws OperatorCreationException, PKCSException,
           GeneralSecurityException, IOException
{
    JcaPKCS10CertificationRequest jcaRequest =
                new JcaPKCS10CertificationRequest(request).setProvider("BC");
    PublicKey key = jcaRequest.getPublicKey();

    ContentVerifierProvider verifierProvider =
                                   new JcaContentVerifierProviderBuilder()
                                       .setProvider("BC").build(key);

    return jcaRequest.isSignatureValid(verifierProvider);
}
```

As you can see it is a three step process, first the key is extracted from the request, in this case by using the encoding to create and configure a `JcaPKCS10CertificationRequest` and calling its `getPublicKey()` method. Second, the public key is then used to create a general verifier provider using the Bouncy Castle provider with the `JcaContentVerifierProviderBuilder` class. Third and finally, the content verifier provider is passed to the `isSignatureValid()` method where (hopefully) it will produce a verifier inside the method call which can be used to validate signatures made using the signature algorithm.

Incorporating Extension Requests

PKCS #10 also allows a certification request to include requests around specific extensions that the holder of the private key would like to have included in the certificate. The most common one of these is the subjectAlternativeName defined in RFC 5280 [72] and represented in Bouncy Castle by the field `Extension.subjectAltName`. Extension requests are added using the PKCS #9 extension request attribute OID represented by `PKCSObjectIdentifiers.pkcs_9_at_extensionRequest` with the value “1.2.840.113549.1.9.14”.

The following example shows how to use the `pkcs_9_at_extensionRequest` OID to add a subjectAltName extension providing an email address to be associated with the CSR’s subject.

```
    /**
     * Create a PKCS#10 request including an extension request detailing the
     * email address the CA should include in the subjectAltName extension.
     *
     * @param keyPair the key pair the certification request is for.
     * @param sigAlg the signature algorithm to sign the PKCS#10
     *                           request with.
     * @return an object carrying the PKCS#10 request.
     * @throws OperatorCreationException in case the private key is
     * inappropriate for signature algorithm selected.
     * @throws IOException on an ASN.1 encoding error.
     */
    public static PKCS10CertificationRequest createPKCS10WithExtensions(
            KeyPair keyPair, String sigAlg)
            throws OperatorCreationException, IOException
    {
        X500NameBuilder x500NameBld = new X500NameBuilder(BCStyle.INSTANCE)
                .addRDN(BCStyle.C, "AU")
                .addRDN(BCStyle.ST, "Victoria")
                .addRDN(BCStyle.L, "Melbourne")
                .addRDN(BCStyle.O, "The Legion of the Bouncy Castle");
        X500Name subject = x500NameBld.build();

        PKCS10CertificationRequestBuilder requestBuilder
                = new JcaPKCS10CertificationRequestBuilder(
                                            subject, keyPair.getPublic());

        ExtensionsGenerator extGen = new ExtensionsGenerator();

        extGen.addExtension(Extension.subjectAlternativeName, false,
                new GeneralNames(
                        new GeneralName(
                                GeneralName.rfc822Name,
                                "feedback-crypto@bouncycastle.org")));

        Extensions extensions = extGen.generate();

        requestBuilder.addAttribute(
                PKCSObjectIdentifiers.pkcs_9_at_extensionRequest, extensions);

        ContentSigner signer = new JcaContentSignerBuilder(sigAlg)
                                .setProvider("BC").build(keyPair.getPrivate());

        return requestBuilder.build(signer);
    }
```

As you can see from the use of the `ExtensionsGenerator` class multiple types of extensions can be easily added. It is worth keeping in mind that while it is the case an extension can be added to a CSR using `pkcs_9_at_extensionRequest`, the CA is under no obligation to add them. Contrariwise if you are writing a CA and you are willing to process the extension request attribute you should check what extensions are being requested to make sure nothing unwanted creeps into one of the certificates you issue.

BCFIPS Provider Configuration for PKCS #10

As it is for the most part a bad idea, FIPS 140-2 does not allow the use of dual use keys. The mantra is basically encrypt or sign as you wish but please do not do both with the same key (mostly). The "mostly" is that there is one caveat on this, allowed for both in SP 800-56B Rev 1 [10], Section 6.1 and SP 800-57 Pt1 Rev 4 [12], Section 8.1.5.1.1.2. The caveat is that is okay to use an encryption key for signing once, for the purpose of generating a certification request so that a certificate can be issued for the encryption key.

The Bouncy Castle FIPS provider normally blocks this out of hand with RSA keys, however a system property is provided to enable dual use of RSA keys for the purpose described in SP 800-56B Rev 1. The system property is called `org.bouncycastle.rsa.allow_multi_use` and if you set it to `true` PKCS #10 for RSA encryption keys will work as required.

Certificate Request Message Format

One of the short-comings of PKCS #10 is that it is designed with algorithms that can be used for signing only. The only proof-of-possession step is the validation of the signature on the PKCS #10 request. Normally the validation step is taken to show that the owner of the public key signed the request so must have had the private key. RFC 4211 [59] defines Certificate Request Message Format (CRMF), a protocol that deals with this limitation by providing alternate mechanisms for handling keys that can be used only for encryption and key agreement as well as supporting keys that can be used for signature generation and verification. CRMF looks at the issuing of a certificate as the co-ordination between two entities, a registration authority (RA) and a certification authority (CA), where the RA validates a request as being authorised and the CA issues the actual certificate. Obviously both these entities might be the same actual "thing", but it is worth keeping in mind that this is not required.

The core structure of CRMF is the CertReqMsg which is a three part structure and has the following ASN.1 definition:

```
CertReqMsg ::= SEQUENCE {
   certReq   CertRequest,
   popo       ProofOfPossession  OPTIONAL,
   -- content depends upon key type
   regInfo   SEQUENCE SIZE(1..MAX) of AttributeTypeAndValue OPTIONAL
}
```

The certReq field is defined by a structure which contains just about every possibility you would consider and has the following definition:

```
CertRequest ::= SEQUENCE {
   certReqId     INTEGER,        -- ID for matching request and reply
   certTemplate  CertTemplate, --Selected fields of cert to be issued
   controls      Controls OPTIONAL } -- Attributes affecting issuance

CertTemplate ::= SEQUENCE {
   version      [0] Version               OPTIONAL,
   serialNumber [1] INTEGER               OPTIONAL,
   signingAlg   [2] AlgorithmIdentifier   OPTIONAL,
   issuer       [3] Name                  OPTIONAL,
   validity     [4] OptionalValidity      OPTIONAL,
   subject      [5] Name                  OPTIONAL,
   publicKey    [6] SubjectPublicKeyInfo  OPTIONAL,
   issuerUID    [7] UniqueIdentifier      OPTIONAL,
   subjectUID   [8] UniqueIdentifier      OPTIONAL,
   extensions   [9] Extensions            OPTIONAL }

OptionalValidity ::= SEQUENCE {
   notBefore  [0] Time OPTIONAL,
   notAfter   [1] Time OPTIONAL } --at least one must be present

Time ::= CHOICE {
   utcTime        UTCTime,
   generalTime    GeneralizedTime }
```

The certReqId is simply an integer the requestor includes so that it can identify the response coming back from the CA when its request is processed. The fields in CertTemplate basically mean what you would imagine from when we looked at the structure of an X.509 certificate in Chapter 8. It should be pointed out here that there are restrictions on some of them which are covered in RFC 4211, so if you are planning to do a lot of in depth work with CRMF it is worth downloading a copy of the RFC as well.

While we are also familiar with AttributeTypeAndValue from Chapter 8 as well, the controls field is one which deserves more explanation as we will see examples of its use further on. CRMF allows for the requestor to send the RA registration tokens, authentication information, publication requests, archive information for the requestor private key, the details of a certificate being updated by the new request, and what protocol the CA should use if it needs to send an encrypted response. All these items can be specified in the controls field.

The popo field is where the additional forms of proof-of-possession are recognised. It is an ASN.1 CHOICE item with the following options.

```
   ProofOfPossession ::= CHOICE {
       raVerified        [0] NULL,
       signature         [1] POPOSigningKey,
       keyEncipherment   [2] POPOPrivKey,
       keyAgreement      [3] POPOPrivKey }
```

The raVerified choice is for use where the RA has already confirmed the ownership of the public key for the CA. The signature choice provides a sequence including a signature which is computed over parts of the request to fulfill the proof-of-possession requirement with a similar approach like that of PKCS #10. The keyEncipherment choice provides a way of specifying how dealing with a certificate issued for an encryption only key should be dealt with. Finally, the keyAgreement choice provides a way of specifying how a certificate issued for a key that can only be used with key agreement should be dealt with.

Signature Based Proof-of-Possession

The first example we will look at is generating a simple request using the closest approach to PKCS #10, that is a certification request for a signing key.

```
/**
 * Basic example for generating a CRMF certificate request with POP for
 * an signing algorithm like DSA or a key pair for signature generation
 * from an algorithm like RSA.
 *
 * @param kp key pair whose public key we are making the request for.
 * @param subject subject principal to be associated with the certificate.
 * @param algorithm name of signature algorithm to use.
 * @param certReqID identity (for the client) of this certificate request.
 */
public static byte[] generateRequestWithPOPSig(
    KeyPair kp, X500Principal subject, String algorithm, BigInteger certReqID)
    throws CRMFException, IOException, OperatorCreationException
{
    JcaCertificateRequestMessageBuilder certReqBuild
        = new JcaCertificateRequestMessageBuilder(certReqID);

    certReqBuild
        .setPublicKey(kp.getPublic())
        .setSubject(subject)
        .setProofOfPossessionSigningKeySigner(
            new JcaContentSignerBuilder(algorithm)
                .setProvider("BC")
                .build(kp.getPrivate()));

    return certReqBuild.build().getEncoded();
}
```

As you can see in the code, the private key associated with the public key the certificate is being requested for is used to generate a signed component included in the request to verify proof-of-possession.

For signature based proof-of-possession, CRMF also allows us to share a secret between an RA/CA and a requestor and then use that as the basis for requesting the certificate. If we are making use of this we leave the subject out of the request and our method becomes the following:

```
/**
 * Authenticating example for generating a CRMF certificate request with POP
 * for a signing algorithm. In this case the CA will verify the subject from
 * the MAC validation.
 *
 * @param kp key pair whose public key we are making the request for.
 * @param certReqID identity (for the client) of this certificate request.
 * @param algorithm name of signature algorithm to use.
 * @param reqPassword authorising password for this request.
 */
public static byte[] generateRequestWithPOPSig(
    KeyPair kp, String algorithm, BigInteger certReqID, char[] reqPassword)
    throws CRMFException, IOException, OperatorCreationException
{
    JcaCertificateRequestMessageBuilder certReqBuild
        = new JcaCertificateRequestMessageBuilder(certReqID);

    certReqBuild
        .setPublicKey(kp.getPublic())
        .setAuthInfoPKMAC(
            new PKMACBuilder(
                new JcePKMACValuesCalculator()),
            reqPassword)
        .setProofOfPossessionSigningKeySigner(
            new JcaContentSignerBuilder(algorithm)
                .setProvider("BC")
                .build(kp.getPrivate()));

    return certReqBuild.build().getEncoded();
}
```

with the MAC employed providing the means for the CA/RA to authenticate our request and match up the appropriate subject ID, in line with whatever other information might be available.

Encryption Based Proof-of-Possession

For an encryption only key, generating proof-of-possession based on a private key signature is not always an option. CRMF takes this into account and a basic request under those circumstances looks like the following:

```
/**
 * Basic example for generating a CRMF certificate request with POP for
 * an encryption only algorithm like ElGamal.
 *
 * @param kp key pair whose public key we are making the request for.
 * @param subject subject principal to be associated with the certificate.
 * @param certReqID identity (for the client) of this certificate request.
 */
public static byte[] generateRequestWithPOPEnc(
    KeyPair kp, X500Principal subject, BigInteger certReqID)
    throws CRMFException, IOException
{
    JcaCertificateRequestMessageBuilder certReqBuild
        = new JcaCertificateRequestMessageBuilder(certReqID);

    certReqBuild
        .setPublicKey(kp.getPublic())
        .setSubject(subject)
        .setProofOfPossessionSubsequentMessage(SubsequentMessage.encrCert);

    return certReqBuild.build().getEncoded();
}
```

In this case, the key is the call to the `certReqBuild.setProofOfPossessionSubsequentMessage()`. The `SubsequentMessage.encrCert` value specifies that the CA/RA should return the certificate in an encrypted envelope (usually EnvelopedData from CMS), and that the requester will demonstrate proof-of-possession by decrypting the envelope and recovering the certificate correctly. The default version of this call, which just takes the SubsequentMessage setting, passes back the request as being proof-of-possession for encryption keys.

Key Agreement Based Proof-of-Possession

For key agreement the methods for encryption based proof-of-possession can also be used if the CA supports the use of EnvelopedData with key agreement. The following example specifies the certificate should be sent back encrypted for key agreement.

```
/**
 * Basic example for generating a CRMF certificate request with POP for
 * a key agreement public key.
 *
 * @param kp key pair whose public key we are making the request for.
 * @param subject subject principal to be associated with the certificate.
 * @param certReqID identity (for the client) of this certificate request.
 */
public static byte[] generateRequestWithPOPAgree(
    KeyPair kp, X500Principal subject, BigInteger certReqID)
    throws CRMFException, IOException
{
    JcaCertificateRequestMessageBuilder certReqBuild
        = new JcaCertificateRequestMessageBuilder(certReqID);

    certReqBuild
        .setPublicKey(kp.getPublic())
        .setSubject(subject)
        .setProofOfPossessionSubsequentMessage(
            ProofOfPossession.TYPE_KEY_AGREEMENT, SubsequentMessage.encrCert);

    return certReqBuild.build().getEncoded();
}
```

The difference between this call and the previous one is the setting of the proof-of-possession type to `TYPE_KEY_AGREEMENT`. An additional form of proof-of-possession is also available for key agreement keys and is described in RFC 4211, Section 4.3, which refers to the technique described in RFC 2875, Section 3 [48]. At the moment this technique relies heavily on SHA-1 so while it is listed as mandatory for server support it is better to use an EnvelopedData based approach where available.

Certificate Management over CMS

The full details of Certificate Management over CMS (CMC) were initially described in RFC 5272 [69], RFC 5273 [70], and 5274 [71] in 2008 and were updated by RFC 6402 in 2011. The protocol is interesting as it is used to support RFC 7030 - Enrolment over Secure Transport (EST) [91] which we will look at next.

Enrolment over Secure Transport

As the name might imply, Enrolment over SecureTransport, as described in RFC 7030 [91] is a registration process designed to be carried out over TLS.

Example Set Up

As we are leveraging TLS the setup really just requires us to obtain a copy of the trust anchors applicable to the EST server so that we can authenticate the connection. The method below is an example of a minimal approach to the task of fetching a remote trust anchor:

```java
/**
 * Grab a set of trust anchors from a URL and return them. Note, we've skipped
 * doing any sort of verification here for the sake of the example. More caution
 * is recommended in real life.
 *
 * @param url the URL pointing at the trust anchor set.
 * @return a set of TrustAnchors based on what was retrieved from the URL.
 */
public static Set<TrustAnchor> fetchTrustAnchor(String url)
    throws Exception
{
    CertificateFactory fac = CertificateFactory.getInstance("X509");
    HashSet<TrustAnchor> out = new HashSet<TrustAnchor>();

    InputStream taStream = new URL(url).openStream();
    for (Certificate cert : fac.generateCertificates(taStream))
    {
        out.add(new TrustAnchor((java.security.cert.X509Certificate)cert, null));
    }
    taStream.close();

    return out;
}
```

In real life you would obviously want to do more to make sure the trust anchor was really the one you should be using, even the EST demo site http://www.testrfc7030.com/[20] recommends checking the current SHA-256 hash value (details of the current one are on the site). The fetch method does provide us with enough to run a client example requesting a certificate.

Getting Information From the EST Server

Once you have your trust anchor, you can then query the EST server to find out what CAs the EST server issues under and also what Certificate Signing Request (CSR) attributes are supported.

The CSR Attributes are used to inform the client about additional information that the server needs to request as part of the certificate issuing process and also what restrictions there are on the client in

[20] www.testrfc7030.com

terms of what can be requested. Examples of restrictions might include things like algorithms, or even that elliptic curves need to belong to a particular set.

Building a basic EST client in Bouncy Castle is quite straight forward, as the following method shows. Have a look at the code and then we will look at it more closely.

```
/**
 * Connect to an EST server and print out it's CA details and CSR Attributes.
 *
 * @param trustAnchors trustAnchors to validate (or not) the connection
 * @param hostName     hostName of the EST server
 * @param portNo       port number to connect on
 */
public static void listCAsAndCSRAttributes(
    Set<TrustAnchor> trustAnchors, String hostName, int portNo)
    throws ESTException
{
    String serverRootUrl = hostName + ":" + portNo;

    X509TrustManager[] trustManagers = JcaJceUtils.getCertPathTrustManager(
                                                         trustAnchors, null);

    // Create the service builder
    JsseESTServiceBuilder estBldr = new JsseESTServiceBuilder(
                                               serverRootUrl, trustManagers);

    // Refuse to match on a "*.com". In the most general case you
    // could use the known suffix list.
    estBldr.withHostNameAuthorizer(
        new JsseDefaultHostnameAuthorizer(Collections.singleton("com")));

    // Build our service object
    ESTService estService = estBldr.build();

    // Fetch and list the CA details
    CACertsResponse csrCerts = estService.getCACerts();

    for (X509CertificateHolder id : csrCerts.getCertificateStore()
                                                    .getMatches(null))
    {
        System.out.println("CA: " + id.getIssuer()
                                  + "[" + id.getSerialNumber() + "]");
    }
```

```

        // Fetch and list the CSR attributes
        CSRRequestResponse csrAttributes = estService.getCSRAttributes();

        for (ASN1ObjectIdentifier id : csrAttributes.getAttributesResponse()
                                                            .getRequirements())
        {
            System.out.println("CSR attribute OID: " + id);
        }
    }
```

The method divides into roughly three parts, establishing an EST service, which is done using the `JsseESTServiceBuilder` and uses the passed in trust anchors to establish its authenticity. The EST package includes a wild-card matcher for identifying servers based on wild-card certificates, the `JsseDefaultHostnameAuthorizer` class. In our case we're just looking to match a certificate coming from a ".com" domain, but we are expecting something more definitive than a certificate which represents "*.com". In the general case you could download a list of common suffixes to block multiple top-level-domains such "org", "net", and "info". The `JsseESTServiceBuilder` also has methods on it for additional configuration, such as specifying the TLS protocol, the JSSE provider, and cipher suites to use as well.

Once the EST service is built and returned as an `ESTService` object, the service has methods on it that allow us to query the services public interfaces. Following on from the call to the `build()` method we ask the service to provide it with the CA certificates it issues under - `estService.getCACerts()` - and then the CSR attributes that apply to server.

Invoking our setup code and our query code on the EST demo site can be done in a couple of lines as follows:

```
        Set<TrustAnchor> trustAnchors = fetchTrustAnchor(
                                          "http://testrfc7030.com/dstcax3.pem");

        listCAsAndCSRAttributes(trustAnchors, "testrfc7030.com", 8443);
```

Running the code on the test server will produce something similar to the following:

```
CA: CN=estExampleCA[1274829662040349O932]
CSR attribute OID: 1.2.840.113549.1.9.1
CSR attribute OID: 1.3.132.0.34
CSR attribute OID: 2.16.840.1.101.3.4.2.2
CSR attribute OID: 1.3.6.1.1.1.1.22
```

How the attribute OIDs should be interpreted and how the information should be sent back (where required) is defined by the server.

Getting a Certificate

In the response we saw earlier we can see the server has included the OID for SHA-384 (2.16.840.1.101.3.4.2.2) and the EC curve P-384 (1.3.132.0.34), usually we would interpret this as request for any certification requests to be using a SHA-384 certificate with a key on the P-384 curve. While the demo system we are using does not actually enforce this, the example method below is written to respect this requirement:

```
/**
 * Enroll with an EST CA by sending a PKCS#10 certification request with a
 * server that expects HTTPAuth.
 *
 * @param trustAnchors trustAnchors to validate (or not) the connection
 * @param hostName     hostName of the EST server
 * @param portNo       port number to connect on
 * @param userName     user ID to use for connecting
 * @param password     password associated with user ID
 * @param keyPair      the key pair to generate the CSR for
 * @return a Store of the response certificates as X509CertificateHolders
 */
public static Store<X509CertificateHolder> estEnroll(
    Set<TrustAnchor> trustAnchors,
    String hostName, int portNo,
    String userName, char[] password,
    KeyPair keyPair)
    throws Exception
{
    // Build our service object
    JsseESTServiceBuilder estBldr = new JsseESTServiceBuilder(
        hostName, portNo, JcaJceUtils.getCertPathTrustManager(
                                            trustAnchors, null));

    estBldr.withHostNameAuthorizer(
        new JsseDefaultHostnameAuthorizer(Collections.singleton("com")));

    ESTService estService = estBldr.build();

    // Create PKCS#10 certification request
    PKCS10CertificationRequestBuilder pkcs10Builder =
        new JcaPKCS10CertificationRequestBuilder(
                    new X500Name("CN=EST Test Client"),
                    keyPair.getPublic());
    ContentSigner contentSigner = new JcaContentSignerBuilder("SHA384WITHECDSA")
                                  .setProvider("BC").build(keyPair.getPrivate());

    PKCS10CertificationRequest certReq = pkcs10Builder.build(contentSigner);

    // Specify how we will authenticate ourselves
    ESTAuth auth = new JcaHttpAuthBuilder(null, userName, password)
        .setNonceGenerator(new SecureRandom()).setProvider("BC").build();

    // Request the certificate
    boolean isReenroll = false;   // show this is a new enrollment
    EnrollmentResponse enrollmentResponse = estService.simpleEnroll(
                                                        isReenroll, certReq, auth);

    // The enrollment action can be deferred by the server.
    while (!enrollmentResponse.isCompleted())
    {
        Thread.sleep(Math.max(
            1000,
            enrollmentResponse.getNotBefore() - System.currentTimeMillis()));
        enrollmentResponse = estService.simpleEnroll(isReenroll, certReq, auth);
    }

    return enrollmentResponse.getStore();
}
```

Building the server is similar to before, we then build a PKCS#10 certification request to provide the server with our public key so it can create a certificate for us. As we are trying to get a certificate issued, the server is expecting us to provide some level of authentication, so we construct an `ESTAuth` object to use HTTP-Auth (over TLS) to identify ourselves to the server.

The next step is to perform the enrollment and get the response. EST also provides support for re-issue of certificates so in this case, as it's our first, we call the `simpleEnroll()` method with re-enrollment set to false and await our response. An EST server has the right to tell a client to come back later, so before attempting to extract the certificate from the server response we call `isCompleted()` to make sure we really received a certificate. In the event `isCompleted()` returns false the response will be populated with a value giving us the a time we can try again, the best advice is too sleep till then and then retry. Assuming all goes well and we have a completed response, the `EnrollmentResponse` then includes one or more certificates for us to use with the end-entity being the one we associated with our certification request.

It's also fairly simple to exercise this code with the EST demo server. Assuming you continue with the previous code fragment for fetching the trust anchor and listing the attributes, the following sample code will execute the enrollment request and print out some details about the returned certificate:

```
    ECGenParameterSpec ecGenSpec = new ECGenParameterSpec("P-384");
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("ECDSA", "BC");
    kpg.initialize(ecGenSpec, new SecureRandom());

    KeyPair kp = kpg.generateKeyPair();

    Store<X509CertificateHolder> certs = estEnroll(
                                        trustAnchors,
                                        "testrfc7030.com", 8443,
                                        "estuser", "estpwd".toCharArray(),
                                        kp);
    for (X509CertificateHolder cert : certs.getMatches(null))
    {
        System.out.println("Issuer : " + cert.getIssuer()
                                     + "[" + cert.getSerialNumber() + "]");
        System.out.println("Subject: " + cert.getSubject());
    }
```

Getting a Certificate with POP

Getting a Certificate with POP EST also provides support for using proof-of-possession or POP. There's a slight twist to this though, the protocol requires access to the TLS tls-unique channel binding type. This channel binding is defined in RFC 5929 [80] but is not currently exposed by the JSSE API. The EST package provides a `ChannelBindingProvider` interface that can be added using the service builder. At the time of writing only the BCJSSE supports this so the following code provides an implementation for the `ChannelBindingProvider` that only works with the BCJSSE.

```
/**
 * BouncyCastle specific channel binding provider to allow access to
 * things like tls-unique
 */
public class BCJsseChannelBindingProvider
    implements ChannelBindingProvider
{
    public boolean canAccessChannelBinding(Socket sock)
    {
        return sock instanceof BCSSLSocket;
    }

    public byte[] getChannelBinding(Socket sock, String binding)
    {
        BCSSLConnection bcon = ((BCSSLSocket)sock).getConnection();
        if (bcon != null)
        {
            return bcon.getChannelBinding(binding);
        }
        return null;
    }
}
```

A sample usage for the `BCJsseChannelBindingProvider` and the POP using the `simpleEnrollPoP()` method on the `ESTService` class is given below.

```
        estBldr.withChannelBindingProvider(new BCJsseChannelBindingProvider());

        ESTService estService = estBldr.build();

        enrollmentResponse = estService.simpleEnrollPoP(
                                reEnroll, pkcs10Builder, contentSigner, auth);
```

Note that in this case we need to pass the builder for the certification request to the enrollment method, not a pre-built certification request. This is so the attributes required for POP can be added to the request once the connection to the server is established as it is the only time at which the information needed becomes available.

Certificate Management Protocol

Certificate Management Protocol (CMP) is currently based on RFC 4210 [58]. It provides an ASN.1 based protocol for the management and distribution of X.509 certificates and certification requests. Like CMS there are additional RFCs providing updates for newer algorithms that can be used with the protocol.

```
PKIMessage ::= SEQUENCE {
     header           PKIHeader,
     body             PKIBody,
     protection   [0] PKIProtection OPTIONAL,
     extraCerts   [1] SEQUENCE SIZE (1..MAX) OF CMPCertificate
                      OPTIONAL
}
```

where `CMPCertificate ::= Certificate`, the same structure used by X.509.

The `header` structure is used to contain transaction, address, and other information, either for a request, or for a response. As the name suggests `body` contains the actual information being carried by the message, with the `PKIBody` structure being assigned a particular type code. The last field `extraCerts` is for additional certificates that may be useful to the receiver of a message, for example a CA may send additional certificates to allow formation of a certificate validation path to the end-entity certificate it has just issued in the body of the message.

The `PKIProtection` type referred to by `protection` field is actually a `BIT STRING`. which is a calculated value, be it a MAC or signature, calculated from the DER encoding of the following structure:

```
ProtectedPart ::= SEQUENCE {
    header        PKIHeader,
    body          PKIBody
}
```

The details of the protection algorithm used appear in the `PKIHeader` structure that forms the start of the message.

The following method shows how to use the Bouncy Castle API to create a simple CMP message carrying a certification request and using MAC protection and authentication.

```java
    /**
     * Create a MAC protected CMP ProtectedPKIMessage containing a CRMF certification
     * request.
     *
     * @param sender sender of the certification request.
     * @param recipient the target CA address.
     * @param certReqMsg the actual CRMF certification request.
     * @param password the password to use.
     * @return a ProtectedPKIMessage
     */
    public static ProtectedPKIMessage generateMacProtectedMessage(
        X500Name sender, X500Name recipient,
        byte[] senderNonce,
        CertificateRequestMessage certReqMsg,
        char[] password)
        throws CMPException, OperatorCreationException
    {
        MacCalculator pbCalculator = new JcePBMac1CalculatorBuilder(
                      "HmacSHA256", 256).setProvider("BC").build(password);

        return new ProtectedPKIMessageBuilder(
                    new GeneralName(sender), new GeneralName(recipient))
                            .setMessageTime(new Date())
                            .setSenderNonce(senderNonce)
                            .setBody(new PKIBody(
                                PKIBody.TYPE_INIT_REQ,
                                new CertReqMessages(
                                        certReqMsg.toASN1Structure())))
                            .build(pbCalculator);
    }
```

You will note that the algorithm used for the MAC calculator is different from the scheme we used with CRMF earlier. While this shows that different MAC algorithms can be accommodated, this case also shows an algorithm which is more FIPS friendly, being based on RFC 8018 [100] which describes what was originally known as PKCS #5 PBKDF2. There is usually no need to apply a MAC to the CRMF message in this case as the CMP MAC structure covers both the CRMF message and the header information of the CMP message.

While the body of the `generateMacProtectedMessage()` message is fairly succinct, the number of arguments passed to the method show that a bit of preparation is required.

The following code shows the preparation required to build the necessary parts:

```
ECGenParameterSpec ecGenSpec = new ECGenParameterSpec("P-384");
KeyPairGenerator kpg = KeyPairGenerator.getInstance("ECDSA", "BC");
kpg.initialize(ecGenSpec, random);
KeyPair kp = kpg.generateKeyPair();

BigInteger certReqId = BigInteger.valueOf(System.currentTimeMillis());

byte[] senderNonce = new byte[16];

random.nextBytes(senderNonce);

CertificateRequestMessage certReqMsg = new CertificateRequestMessage(
        CRMFUtils.generateRequestWithPOPSig(
                kp, new X500Principal("CN=CMP Test Client"),
                "SHA256withECDSA", certReqId));

// create the CMP message wrapping the CRMF certification request.
X500Name sender = new X500Name("CN=Certificate Requester");
X500Name recipient = new X500Name("CN=Certificate Issuer");

ProtectedPKIMessage pkiMessage = generateMacProtectedMessage(
    sender, recipient, senderNonce, certReqMsg, "secret".toCharArray());
```

We generate a key pair and choose a transaction ID and a nonce for the receiver to include in their reply. Next we produce a CertificationRequestMessage using the CRMF API for our public key. We then use these plus our shared secret to produce a protected PKI message using the CMP method. One small note: for brevity the CertificateRequestMessage is built off the encoding returned by the CRMF method we looked at earlier. It is obvious that "in real life" you would not double handle the message in this way. Other than that, the code fragment represents the minimum required.

Naturally, having included the MAC in the request, the receiver will need to validate the MAC to confirm our message is valid. The following simple code fragment shows how the MAC would be verified by the recipient:

```
PBEMacCalculatorProvider macProvider =
    new JcePBMac1CalculatorProviderBuilder().setProvider("BC").build();

System.out.println("message verified: "
    + pkiMessage.verify(macProvider, "secret".toCharArray()));
```

Although only a couple of lines, the code fragment shows the difference between creation and verification of the MAC. For verification we need to build a `MacCalculatorProvider` which will return a `MacCalculator` suitable for the message. The provider is then passed to the `verify()` message together with the secret and the class constructs a `MacCalculator` internally which is then used to calculate a MAC on the message contents. Assuming the previous code fragments have all been implemented, the last fragment above should produce

```
message verified: true
```

Indicating that the MAC reconstruction from the message data was successful.

Summary

In this chapter we have looked at the standard protocols for creating certification requests - PKCS #10 and CRMF - as well as looking at some protocols, such as CMC, EST, and CMP which are typically used for communicating certification requests to certification authorities (CAs).

CAs typically expect some form of Proof-of-Possession (POP), showing that the party requesting the certificate really has the private key. We have also seen that users are able to authenticate themselves with CAs, typically through shared secrets, so that a CA can be confident it is communicating with someone it knows.

While the CA communication protocols offer similar levels of utility, there is a big difference between PKCS #10 and CRMF. CRMF allows for a wider range of mechanisms for Proof-of-Possession and is able to accommodate algorithms which are restricted to just key agreement and encryption, in addition to signature algorithms which can be also be supported by PKCS #10.

Chapter 13: OpenPGP

OpenPGP is a widely used message format for providing security services in message communications and storage. Like CMS it is primarily public key based, but it has its own packet format and its own idea of certificates. The packets, or record structures, used in OpenPGP are defined in RFC 4880 [64]. OpenPGP offers signing, encryption, compression, authentication, and password based encryption. Bouncy Castle offers the full range of these, including the more recent KeyBox storage format from GPG and the use of EC and Edwards Curves.

The Basics

PGP messages are represented as a stream of records. Unlike what we see in CMS these records usually appear as seperate entities in the the stream, so for example, encapsulated signed data is represented by a stream of records making up signature headers, the data, and the final signature record separately. This is quite different from what you see with CMS where all this information is wrapped in one single structure.

The OpenPGP classes can be found in the Bouncy Castle distribution jars beginning with “bcpg”.

The Literal Data Record

The core record type for OpenPGP messaging is the Literal Data record. Compression and encryption in OpenPGP all apply to OpenPGP records and it is the Literal Data record that is used to contain the actual data that you want compressed or encrypted. Likewise, with encapsulated signatures, we will see a Literal Data record is used to mark where the signed message data appears in the OpenPGP stream.

The Literal Data record is described in Section 5.9 of RFC 4880. In the Bouncy Castle APIs they are created using a `PGPLiteralDataGenerator` and you begin one by by calling one of the `open()` methods on the `PGPLiteralData` class. In OpenPGP literal data comes in three types: ascii text (`PGPLiteralData.TEXT`), binary (`PGPLiteralData.BINARY`), and UTF-8 (`PGPLiteralData.UTF8`). A Literal Data record also includes a file name and if none is required the special string “_CONSOLE” (`PGPLiteralData.CONSOLE`) can be used instead.

⚠ Always remember to wrap data that is encrypted or signed in a Literal Data record. All the tools expect this!

Generating Armored Output

OpenPGP has an ASCII armored format which is very similar to PEM. It is base 64 encoded, but includes a checksum at the end of each record. The class for producing armored output is the `ArmoredOutputStream` class in the `org.bouncycastle.bcpg` package. The `ArmoredInputStream` class in the same package provides a parsing mechanism.

There are a few message types: “PGP PUBLIC KEY BLOCK” and “PGP PRIVATE KEY BLOCK” for keys, “PGP SIGNATURE” for detached signatures and a generic type - “PGP MESSAGE” which is a bit of a catch all. There is also a special case “PGP SIGNED MESSAGE” which we will look at when we discuss cleartext signatures further on.

The following code fragment produces armored output from a single Literal Data record.

```java
        byte[] msg = Strings.toByteArray("Hello, world!");

        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        ArmoredOutputStream aOut = new ArmoredOutputStream(bOut);

        PGPLiteralDataGenerator lGen = new PGPLiteralDataGenerator();

        OutputStream lOut = lGen.open(aOut, PGPLiteralData.TEXT,
                            PGPLiteralData.CONSOLE, msg.length, new Date());

        lOut.write(msg);
        lOut.close();

        aOut.close();

        System.out.println(Strings.fromByteArray(bOut.toByteArray()));
```

If you run the code you should see the following:

```
-----BEGIN PGP MESSAGE-----
Version: BCPG v1.63

yxt0CF9DT05TT0xFXX9ONEhlbGxvLCB3b3JsZCE=
=sJgV
-----END PGP MESSAGE-----
```

You can see the checksum on the last line of the message block. It is a 24 bit CRC, so useful for picking up basic transmission errors, but useless from a cryptographic integrity point of view. The presence of the checksum created an interesting problem for the implementation of `ArmoredOutputStream` as the alert box says, it meant the `close()` method on the stream class could not close the underlying stream as the step of calling `close()` had to be used as the trigger to generate the checksum.

> The `close()` method on ArmoredOutputStream is used to signal that the final checksum can be written. As multiple PGP blobs are often written to a single stream, `ArmouredOutputStream.close()` does not close the underlying stream.

Generating Key Rings

PGP stores keys and certificates in key rings. In the simplest case, which we will look at soon, a key ring is simply a master key (always a signing key) and a sub-key which is used for encryption.

The following code generates a basic key ring with a DSA master key and an RSA subkey, returning the encodings for the secret and public key ring components. Read through the code now, it can seem a little unwieldy at first, but the actual “working part” is from lines 15 to 25.

```
 1  /**
 2   * Generate secret/public keys rings with a DSA signing key and an RSA encryption
 3   * key.
 4   *
 5   * @param identity the identity to associate with the keys.
 6   * @param passphrase the passphrase to encrypt the secret keys with.
 7   * @return a byte[][] containing the encoding of secret and public key rings
 8   *              respectively.
 9   */
10  public static byte[][] generateKeyRings(String identity, char[] passphrase)
11      throws GeneralSecurityException, PGPException, IOException
12  {
13      KeyPair dsaKp = DsaUtils.generateDSAKeyPair();
14      KeyPair rsaKp = RsaUtils.generateRSAKeyPair();
15      PGPKeyPair dsaKeyPair = new JcaPGPKeyPair(
16                                      PGPPublicKey.DSA, dsaKp, new Date());
17      PGPKeyPair rsaKeyPair = new JcaPGPKeyPair(
18                                      PGPPublicKey.RSA_ENCRYPT, rsaKp, new Date());
19
20      PGPDigestCalculator sha1Calc = new JcaPGPDigestCalculatorProviderBuilder()
21                                           .build()
22                                           .get(HashAlgorithmTags.SHA1);
23
24      PGPKeyRingGenerator keyRingGen =
25          new PGPKeyRingGenerator(
26                  PGPSignature.POSITIVE_CERTIFICATION, dsaKeyPair, identity, sha1Calc,
27                  null, null,
28                  new JcaPGPContentSignerBuilder(
29                         dsaKeyPair.getPublicKey().getAlgorithm(),
30                         HashAlgorithmTags.SHA384),
31                  new JcePBESecretKeyEncryptorBuilder(
32                             PGPEncryptedData.AES_256, sha1Calc)
33                         .setProvider("BC").build(passphrase));
34         keyRingGen.addSubKey(rsaKeyPair);
35 
36         // create an encoding of the secret key ring
37         ByteArrayOutputStream secretOut = new ByteArrayOutputStream();
38         keyRingGen.generateSecretKeyRing().encode(secretOut);
39         secretOut.close();
40 
41         // create an encoding of the public key ring
42         ByteArrayOutputStream publicOut = new ByteArrayOutputStream();
43         keyRingGen.generatePublicKeyRing().encode(publicOut);
44         publicOut.close();
45 
46         return new byte [][]{secretOut.toByteArray(), publicOut.toByteArray()};
47     }
```

One important item is on line 17, inside the PGPKeyRingGenerator construction a PGP signature type is specified. You'll note this is PGPSignature.POSITIVE_CERTIFICATION. A glance at RFC 4880, Section 5.2.1, that positive certification means the secret key holder has done substantial verification of the claim of identity. The positive certification is used to bind the ID to the master key. The public key is also certified internally, the signature type used there is PGPSignature.SUBKEY_BINDING. Together these two PGP signatures provide roughly what X.509 gives us for both the DSA master key and the RSA encryption sub-key.

You can find out more information about the actual structure of the public and secret components of the PGP keyring in RFC 4880, Section 5.5.2 and 5.5.3 respectively.

Note from version 2.1, GPG defaults to a new key file format, the keybox file format (.kbx). We will take a look at this format as well a bit later in the chapter.

Adding a Sub-Key to an Existing Key Ring

Occasionally you will want to update an encryption key on an existing key ring, usually by adding a new sub-key and leaving the old one in place. The PGPKeyRingGenerator class also has a constructor which allows it to take an existing PGPSecretKeyRing and to use that to generate new PGPSecretKeyRing and PGPPublicKeyRing objects.

An example method for adding a new sub-key follows:

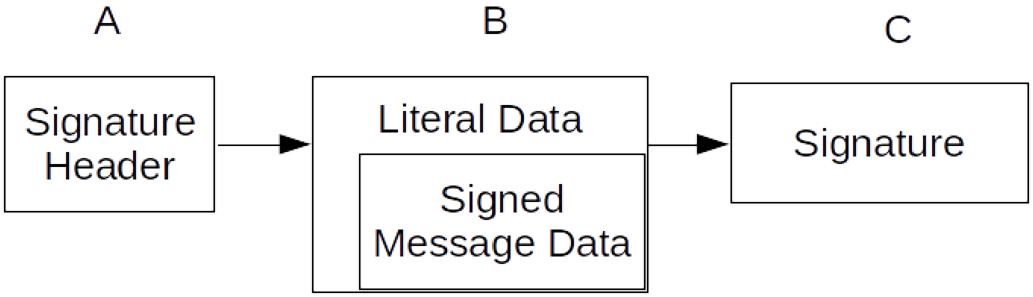
```
  /**
   * Generate secret/public keys rings from an original secret key ring and a new
   * subkey.
   *
   * @param original the original secret key ring to base the new ones on.
   * @param passphrase the passphrase to encrypt/decrypt the secret keys with.
   * @param subKey the public/private subkey pair to add.
   * @return a byte[][] containing the encoding of secret and public key rings
   *                  respectively.
   */
  public static byte[][] addSubKey(PGPSecretKeyRing original, char[] passphrase, PGP\
KeyPair subKey)
      throws PGPException, IOException
  {
      PGPDigestCalculator sha1Calc = new JcaPGPDigestCalculatorProviderBuilder()
                                          .build()
                                          .get(HashAlgorithmTags.SHA1);

      PBESecretKeyDecryptor keyDecryptor = new JcePBESecretKeyDecryptorBuilder().bui\
ld(passphrase);
      PBESecretKeyEncryptor keyEncryptor = new JcePBESecretKeyEncryptorBuilder(
                                  PGPEncryptedData.AES_256, sha1Calc)
                              .setProvider("BC").build(passphrase);

      PGPKeyRingGenerator keyRingGen =
          new PGPKeyRingGenerator(
               original, keyDecryptor, sha1Calc,
               new JcaPGPContentSignerBuilder(
                      original.getPublicKey().getAlgorithm(),
                      HashAlgorithmTags.SHA384),
               keyEncryptor);

      keyRingGen.addSubKey(subKey);

      // create an encoding of the secret key ring
      ByteArrayOutputStream secretOut = new ByteArrayOutputStream();
      keyRingGen.generateSecretKeyRing().encode(secretOut);
      secretOut.close();

      // create an encoding of the public key ring
      ByteArrayOutputStream publicOut = new ByteArrayOutputStream();
```

```
        keyRingGen.generatePublicKeyRing().encode(publicOut);
        publicOut.close();

        return new byte [][]{secretOut.toByteArray(), publicOut.toByteArray()};
    }
```

Note in the method above, the sub-key is added without specifying what hashed/unhashed signature sub-packets to use. In this case the signature sub-packets will be inherited from the primary signature on the master key. There is also an override for `addSubKey()` which allows the hashed and unhashed signature sub-packets to be specified explicitly if required.

Encapsulated Signed Messages

As with CMS, PGP supports both encapsulated signed messages and detached signatures. We will look at encapsulated signatures first. As we mentioned earlier, you can see from the way the code is built that a PGP encapsulated signed message consists of a series of packets, rather than a single structure as with CMS.



Record format for a basic encapsulated signature.

As you can see from the diagram, a basic encapsulated signature consists of three records: a signature header (A) which is also referred to as a One-pass Signature Packet, a Literal Data record containing the signed message (B), and then the actual signature (C). When a signature is nested, the (A, C) pairs need to correspond, so if there were two signatures C_1 and C_2 the records should be arranged as: A_2, A_1, B, C_1, C_2.

Here is a sample method that creates an encapsulated signed message and returns an encoding of the resulting records. Have a look at it first and then we will discuss it further.

```
/**
 * Create a PGP signed message based on the passed in signing key
 * and data.
 *
 * @param signingAlg the signature algorithm to use.
 * @param signingKey the PGP private key to sign with.
 * @param data the message data to be signed.
 * @return an encoding of the signed message
 */
public static byte[] createSignedMessage(
    int signingAlg, PGPPrivateKey signingKey, byte[] data)
    throws PGPException, IOException
{
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();

    // Create a stream for writing a signature to.
    BCPGOutputStream bcOut = new BCPGOutputStream(bOut);

    // set up the signature generator
    PGPSignatureGenerator sGen =
        new PGPSignatureGenerator(
            new JcaPGPContentSignerBuilder(signingAlg, PGPUtil.SHA384)
                .setProvider("BC"));

    sGen.init(PGPSignature.BINARY_DOCUMENT, signingKey);

    // Output the signature header
    // the false means we are not generating a nested signature.
    sGen.generateOnePassVersion(false).encode(bcOut);

    // Create the Literal Data record
    PGPLiteralDataGenerator lGen = new PGPLiteralDataGenerator();

    OutputStream lOut = lGen.open(bcOut,
        PGPLiteralData.BINARY, PGPLiteralData.CONSOLE, data.length, new Date());
    for (int i = 0; i != data.length; i++)
    {
        lOut.write(data[i]);
        sGen.update(data[i]);
    }

    // Finish Literal Data construction
    lOut.close();
```

```java

        // Output the actual signature
        sGen.generate().encode(bcOut);

        // close off the stream.
        bcOut.close();

        return bOut.toByteArray();
    }
```

Looking at the code for setting up the signature generation you can see it follows a similar pattern to the streaming API in CMS. The operators are different as OpenPGP does not have a structure like the ASN.1 AlgorithmIdentifier, but you should find if you apply the same basic philosophy to the use of operators in PGP as you did for CMS and S/MIME things will work.

It is also worth noting that OpenPGP associates a particular type with a signature. In the case for the example, this is `PGPSignature.BINARY_DOCUMENT` as we are signing a binary Literal Data record. You will find definitions for the different signature and certification types used by OpenPGP in Section 5.2.1. Signature Types of RFC 4880. Certifications are different from document signatures and are only used in connection with keys.

Verifying the message involves a couple of steps. The most important thing to notice about verification is that as the Literal Data record exists independently of the signature header and footer, it is necessary to feed the content of the Literal Data record into the signature verifier by hand.

The example given here also makes use of the need to recover the message to verify the signature to write the data that was signed to a stream in case the caller also wishes to see the actual message data. Have a look at the following code.

```java
    /**
     * Recover the original message in bOut, returning true if the signature verifies.
     *
     * @param verifyingKey the PGP public key to verify the message with.
     * @param pgpSignedData the PGP signed message data.
     * @param msgStream a stream to write the recovered message to.
     * @return true if the signed message verifies, false otherwise.
     */
    public static boolean verifySignedMessage(
        PGPPublicKey verifyingKey, byte[] pgpSignedData, OutputStream msgStream)
        throws PGPException, IOException
    {
        // Create a parser for the PGP protocol stream
        JcaPGPObjectFactory pgpFact = new JcaPGPObjectFactory(pgpSignedData);

        // Read the signature header and set up the verification
        PGPOnePassSignatureList onePassList
                                    = (PGPOnePassSignatureList)pgpFact.nextObject();
        PGPOnePassSignature ops = onePassList.get(0);

        ops.init(
            new JcaPGPContentVerifierBuilderProvider().setProvider("BC"),
            verifyingKey);

        // Open up the Literal Data record containing the message
        PGPLiteralData literalData = (PGPLiteralData)pgpFact.nextObject();
        InputStream dIn = literalData.getInputStream();

        // Read the message data
        int ch;
        while ((ch = dIn.read()) >= 0)
        {
            ops.update((byte)ch);
            msgStream.write(ch);
        }

        dIn.close();

        // Read and verify the signature
        PGPSignatureList sigList = (PGPSignatureList)pgpFact.nextObject();
        PGPSignature sig = sigList.get(0);

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

You will notice the method used above, of just creating a `PGPObjectFactory` and then parsing your way through the data is used everywhere for processing PGP messages. Note that "in real life" you do not want to return the recovered message data to a user if the signature fails, although you may wish to put the recovered message somewhere safe for later analysis. The other thing to note is that we have chosen to use the first signature object found as the match for the verification key. While in the case for this example this is correct, it is not correct in general. We will look at a more general purpose way of dealing with signatures when we look at verifying detached signatures below.

Finally the use of the `PGPObjectFactory` is often required for parsing streams as sometimes you cannot be entirely sure what will come next so it is good to be able to specify alternatives. Data may sometimes have a PGP marker packet at the start, which then needs to be skipped, or the literal data might also be wrapped in a compressed data object. As a result of these considerations, if you are processing data from multiple sources, it is usually a good idea to write the processing code with a level of adaptability around the packet types that might arrive in the PGP stream as you parse it.

We can put the two methods together using the following code fragment which creates a message and the necessary keys to allow the signed message to be created and verified, printing the message data when verification is successful.

```
        byte[] message = Strings.toByteArray("Hello, world!");
        KeyPair dsaKp = DsaUtils.generateDSAKeyPair();
        PGPKeyPair dsaKeyPair = new JcaPGPKeyPair(PGPPublicKey.DSA, dsaKp, new Date());

        byte[] signedMessage = createSignedMessage(
            PublicKeyAlgorithmTags.DSA, dsaKeyPair.getPrivateKey(), message);

        ByteArrayOutputStream recStream = new ByteArrayOutputStream();

        if (verifySignedMessage(dsaKeyPair.getPublicKey(), signedMessage, recStream))
        {
            System.out.println(Strings.fromByteArray(recStream.toByteArray()));
        }
```

Running the fragment with the provided methods should produce:

```
Hello, world!
```

indicating that the message stream parsed correctly, the signature verified, and the encapsulated message was recovered as well.

Detached Signatures

Detached signatures in PGP represent roughly the same thing as the detached signatures we see with multipart signed data in S/MIME.

The following method generates a detached signature for binary data using in a passed in byte array as the data to be signed.

```java
/**
 * Generate a detached signature for binary data.
 *
 * @param signingAlg signing algorithm to use.
 * @param signingKey signing key to use.
 * @param data the data to be signed.
 * @return an encoded PGP detached signature.
 */
public static byte[] createDetachedSignature(
    int signingAlg, PGPPrivateKey signingKey, byte[] data)
    throws PGPException, IOException
{
    PGPSignatureGenerator sGen = new PGPSignatureGenerator(
        new JcaPGPContentSignerBuilder(signingAlg, PGPUtil.SHA384)
                                        .setProvider("BC"));

    sGen.init(PGPSignature.BINARY_DOCUMENT, signingKey);

    sGen.update(data);

    PGPSignature pgpSig = sGen.generate();

    return pgpSig.getEncoded();
}
```

As the signature does not contain the data you can see there is no need for things like the `generateOnePassVersion()` as no header information is required.

Also, as the signature is separate from the data that was signed, verification also requires presentation of the original data as part of the verification process. Here is the verification method:

```java
/**
 * Verify a detached signature for binary data.
 *
 * @param verifyingKey the public key to verify the signature with.
 * @param pgpSignature the signature generated for the data.
 * @param data the data that was signed.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyDetachedSignature(
    PGPPublicKey verifyingKey, byte[] pgpSignature, byte[] data)
    throws PGPException, IOException
{
```

```
13          JcaPGPObjectFactory pgpFact = new JcaPGPObjectFactory(pgpSignature);
14  
15          PGPSignatureList sigList = (PGPSignatureList)pgpFact.nextObject();
16  
17          PGPSignature sig = null;
18          for (PGPSignature s: sigList)
19          {
20              if (s.getKeyID() == verifyingKey.getKeyID())
21              {
22                  sig = s;
23                  break;
24              }
25          }
26  
27          if (sig == null)
28          {
29              throw new IllegalStateException("signature for key not found");
30          }
31  
32          sig.init(
33              new JcaPGPContentVerifierBuilderProvider().setProvider("BC"),
34              verifyingKey);
35  
36          sig.update(data);
37  
38          return sig.verify();
39      }
```

In this case the step for picking up the signature that matches the verification key, which spans lines 17 to 25, is more complicated than before, but is correct in general, as a signature in the signature list is only matched if it matches the key id of the verification key that has been provided.

The following code fragment can be used to try out the two sample methods.

```
        byte[] message = Strings.toByteArray("Hello, world!");
        KeyPair dsaKp = DsaUtils.generateDSAKeyPair();
        PGPKeyPair dsaKeyPair = new JcaPGPKeyPair(PGPPublicKey.DSA, dsaKp, new Date());

        byte[] signature = createDetachedSignature(
            PublicKeyAlgorithmTags.DSA, dsaKeyPair.getPrivateKey(), message);

        if (verifyDetachedSignature(dsaKeyPair.getPublicKey(), signature, message))
        {
            System.out.println("signature verified");
        }
        else
        {
            System.out.println("signature failed");
        }
```

Assuming nothing has gone wrong, executing the code should produce the following output:

```
signature verified
```

Indicating a correct signature was created and that the verification key was successfully matched to the signature in the signature list that was created in the signature creation method.

Signing Text Data

PGP also supports the signing of text data. In this case, whether encapsulated or not, the requirement is that text data is canonicalized - in this case meaning that line endings in the signature are always calculated as “\r\n”, rather than what the platform might use natively. We will look at the generation of encapsulated signatures here (generating detached versions can be done by following the changes made in the previous section to the encapsulation code, but for text signatures instead). It is worth noting here that this is not the same definition of canonicalization that is used for cleartext signatures. We will look at cleartext signing in the next section.

The following method creates a text type signature.

```
    /**
     * Create a PGP signed text message based on the passed in signing key
     * and data.
     *
     * @param signingAlg the signature algorithm to use.
     * @param signingKey the PGP private key to sign with.
     * @param data the message data to be signed.
     * @return an encoding of the signed message
     */
    public static byte[] createSignedMessage(
        int signingAlg, PGPPrivateKey signingKey, byte[] data)
        throws PGPException, IOException
    {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();

        // Create a stream for writing a signature to.
```

```
17         BCPGOutputStream bcOut = new BCPGOutputStream(bOut);
18 
19         // set up the signature generator
20         PGPSignatureGenerator sGen =
21             new PGPSignatureGenerator(
22                 new JcaPGPContentSignerBuilder(signingAlg, PGPUtil.SHA384)
23                     .setProvider("BC"));
24 
25         sGen.init(PGPSignature.CANONICAL_TEXT_DOCUMENT, signingKey);
26 
27         // Output the signature header
28         // the false means we are not generating a nested signature.
29         sGen.generateOnePassVersion(false).encode(bcOut);
30 
31         // Create the Literal Data record
32         PGPLiteralDataGenerator lGen = new PGPLiteralDataGenerator();
33 
34         OutputStream lOut = lGen.open(bcOut,
35             PGPLiteralData.TEXT, PGPLiteralData.CONSOLE, data.length, new Date());
36 
37         for (int i = 0; i != data.length; i++)
38         {
39             lOut.write(data[i]);
40             sGen.update(data[i]);
41         }
42 
43         // Finish Literal Data construction
44         lOut.close();
45 
46         // Output the actual signature
47         sGen.generate().encode(bcOut);
48 
49         // close off the stream.
50         bcOut.close();
51 
52         return bOut.toByteArray();
53     }
```

You can see it only differs from the binary signature version in two places, one which sets the signature type on line 25 and where the literal data type is set on line 35. The constant `PGPLiteralData.TEXT` translates into a type code with the value ‘t’ which commonly just means ASCII text. PGP also provides for a type code for Literal Data of ‘u’ which is represented by `PGPLiteralData.UTF8`.

You will probably be wondering where the canonicalization of the data is handled as it does not appear to be dealt with explicitly. For CANONICAL_TEXT_DOCUMENT signatures both the `PGPSignatureGenerator` and the `PGPSignature` classes handle canonicalization internally.

Verifying the message is the same as before. As the canonicalization will be dealt with internally we can make use of the `verifySignedMessage()` method we defined earlier, when we were looking at verifying encapsulated binary signatures.

As a sample of usage, the following code fragment creates a message and the necessary keys to allow the signed message to be created and verified using `verifySignedMessage()`, printing the message data when verification is successful.

```
// platform independent text message generator
ByteArrayOutputStream bOut = new ByteArrayOutputStream();
BufferedWriter bWrt = new BufferedWriter(new OutputStreamWriter(bOut));
bWrt.write("Hello, world!");
bWrt.newLine();
bWrt.close();

byte[] message = bOut.toByteArray();

KeyPair dsaKp = DsaUtils.generateDSAKeyPair();
PGPKeyPair dsaKeyPair = new JcaPGPKeyPair(PGPPublicKey.DSA, dsaKp, new Date());

byte[] signedMessage = createSignedMessage(
    PublicKeyAlgorithmTags.DSA, dsaKeyPair.getPrivateKey(), message);

ByteArrayOutputStream recStream = new ByteArrayOutputStream();

if (verifySignedMessage(dsaKeyPair.getPublicKey(), signedMessage, recStream))
{
    System.out.print(Strings.fromByteArray(recStream.toByteArray()));
}
```

Running the fragment with the provided methods should also produce:

```
Hello, world!
```

indicating that the message stream parsed correctly, the signature verified, and the encapsulated message was recovered as well.

Cleartext Signatures

Cleartext signatures are documented in section 7 of RFC 4880 [64]. The idea behind them is to provide a similar mechanism to the signed multipart we saw with S/MIME - that is a signed message where the signed data can be read without requiring any specialized software to parse it[21]. Reservations about the concept aside, there are a lot of PGP based applications which either produce, or expect cleartext signatures to be used.

Creating Cleartext Signatures

As with signed text data, canonicalization is required to produce a cleartext signature that someone can verify. In this case the canonicalization process is more complicated than before as the message includes armoring so dashes at the start of line must be escaped, the last end of line appearing before the armoring for the signature needs to be ignored in the signature calculation, and trailing spaces on a line are supposed to be ignored.

Fortunately some of these things can be handled transparently and are supported by the `ArmoredOutputStream` class which is used to produce cleartext signatures. Recognising what to put into a signature object and when does require some user involvement however and so a bit of code is required to provide the necessary infrastructure support around these signatures, particularly where the message to be passed through cannot have its line endings changed during processing.

The following utility code handles allows line endings to be preserved in the output, while at the same time dealing with both canonicalization and ignoring trailing whitespace on each line of the data to be signed.

```
// return true if b is a white space character.
static boolean isWhiteSpace(
    byte b)
{
    return b == '\r' || b == '\n' || b == '\t' || b == ' ';
}

// note: trailing white space needs to be removed from the end of
// each line for signature calculation RFC 4880 Section 7.1
static int getLengthWithoutWhiteSpace(
    byte[] line)
{
    int end = line.length - 1;

    while (end >= 0 && isWhiteSpace(line[end]))
    {
        end--;
    }

    return end + 1;
```

[21] These days there is a bit of "it seemed like a good idea at the time" about this idea. If you cannot verify the signature, you cannot tell if the signed data has been tampered with. In a lot of circumstances this really does mean you probably do not want to act on the data in any case.

```
    }

    // add the leading bytes of line to the signature generator and
    // the full line to the output stream.
    private static void processLine(
        PGPSignatureGenerator sGen, OutputStream aOut, byte[] line)
        throws IOException
    {
        // remove trailing white space.
        int length = getLengthWithoutWhiteSpace(line);
        if (length > 0)
        {
            sGen.update(line, 0, length);
        }

        aOut.write(line, 0, line.length);
    }

    // read a line of input, dealing with a variety of line endings.
    private static byte[] readInputLine(
        InputStream fIn)
        throws IOException
    {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();

        int ch;

        while ((ch = fIn.read()) >= 0)
        {
            bOut.write(ch);
            if (ch == '\r')
            {
                fIn.mark(1);
                if ((ch = fIn.read()) == '\n')
                {
                    bOut.write(ch);
                }
                else
                {
                    fIn.reset();
                }
                return bOut.toByteArray();
            }
            else if (ch == '\n')
            {
                return bOut.toByteArray();
            }
        }

        return null;
    }
```

As you can see, preserving line endings does add to the complexity a bit. It may be worth considering whether your application of PGP cleartext signatures requires you to be as general. If your application does not it will simplify the code a bit.

With the support of the `readInputLine()` and `processLine()` methods defined above, an example of a method for producing a cleartext signature from a given private key, digest algorithm and some input data could look as follows:

```
    /**
     * Create a cleartext signed document from the passed in input stream.
     *
     * @param pgpPrivKey the private key to sign with.
     * @param digest the digest algorithm to create the signature with.
     * @param msgIn the input to be included and signed in the message.
     * @param clrOut the output stream to write the cleartext signed document to.
     */
    public static void createCleartextSignature(
        PGPPrivateKey pgpPrivKey, int digest, InputStream msgIn, OutputStream clrOut)
        throws PGPException, IOException
    {
        BufferedInputStream mIn = new BufferedInputStream(msgIn);

        // set up the signature generator.
        int keyAlgorithm = pgpPrivKey.getPublicKeyPacket().getAlgorithm();
        PGPSignatureGenerator sGen = new PGPSignatureGenerator(
            new JcaPGPContentSignerBuilder(keyAlgorithm, digest).setProvider("BC"));

        sGen.init(PGPSignature.CANONICAL_TEXT_DOCUMENT, pgpPrivKey);

        // create the cleartext document.
        ArmoredOutputStream aOut = new ArmoredOutputStream(clrOut);

        aOut.beginClearText(digest);

```

```
27          //
28          // note the last \n/\r/\r\n in the file is ignored
29          //
30          byte[] line = readInputLine(mIn);
31
32          processLine(sGen, aOut, line);
33
34          while ((line = readInputLine(mIn)) != null)
35          {
36              sGen.update((byte)'\r');
37              sGen.update((byte)'\n');
38
39              processLine(sGen, aOut, line);
40          }
41
42          // end the cleartext data
43          aOut.endClearText();
44
45          // output the signature block
46          sGen.generate().encode(aOut);
47
48          aOut.close();
49      }
```

After the signature setup at the beginning of the `createCleartextSignature()` method, the `aOut.beginClearText(digest)` statement produces the start header for the armoring. The `ArmoredOutputStream` aOut will look after dash escaping for us, but we still have to be careful to ignore the last end-of-line in the file so lines 30 to 40 process one line first, and then only add a canonical line end, a '\r\n' to the signature once it is certain there is another line to process[22]. `aOut.endClearText()` signals the end of the cleartext and then the signature generator is used to generate a the final signature which is encoded to `aOut`. This last step is the one that produces the PGP signature armoring.

You can use the following code fragment to run the `createCleartextSignature` method and sign a simple input, the string "Hello, world!":

[22] As the signature is already defined as being of type CANONICAL_TEXT_DOCUMENT a single '\r' or a single '\n' would also do as the signature generator will canonicalize immediately. For the sake of the example it seemed best to make the end-of-line sequence explicit.

```
    byte[] message = Strings.toByteArray("Hello, world!\r\n");
    KeyPair dsaKp = DsaUtils.generateDSAKeyPair();
    PGPKeyPair dsaKeyPair = new JcaPGPKeyPair(PGPPublicKey.DSA, dsaKp, new Date());

    ByteArrayOutputStream sigOut = new ByteArrayOutputStream();
    createCleartextSignature(
        dsaKeyPair.getPrivateKey(), PGPUtil.SHA256,
        new ByteArrayInputStream(message), sigOut);

    System.out.println(Strings.fromByteArray(sigOut.toByteArray()));
```

Assuming all goes well, running the above fragment will produce something like the following:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

Hello, world!
-----BEGIN PGP SIGNATURE-----
Version: BCPG v1.65b04

iF4EAREIAAYFAl42au4ACgkQqjare2QW6Hij7wEAk0oLQGX11Gu0AleiYuZKK/Of
MCyx0DWRDqDDH0G6VpgBAIUG7wJ4/4P3pBTfVmcGJ/zxGzxlUyxS+7G6rqddQeWO
=Vvtg
-----END PGP SIGNATURE-----
```

Note: the signature block will be different as the DSA algorithm is not deterministic. As you can see the first armor headers and footers are not matched and there is no “END PGP SIGNED MESSAGE” footer. Fortunately, as we will see in the next section the Bouncy Castle ArmoredInputStream class will handle this for us.

Verifying Cleartext Signatures

The process of verifying a clear text message is very similar. The following code provides the utility methods needed to support a general purpose verification method. As you will see from looking at it, the main difference from before is that the PGPSignature class needs to be used rather than the PGPSignatureGenerator class.

```
// process a line, ignoring trailing white space.
private static void processLine(PGPSignature sig, byte[] line)
{
    // remove trailing white space.
    int length = getLengthWithoutWhiteSpace(line);
    if (length > 0)
    {
        sig.update(line, 0, length);
    }
}

// process the input data for signature verification.
static void processInput(PGPSignature sig, InputStream sigIn)
    throws IOException
{
    byte[] lookAhead = readInputLine(sigIn);

    processLine(sig, lookAhead);

    if (lookAhead != null)
    {
        while ((lookAhead = readInputLine(sigIn)) != null)
        {
            sig.update((byte)'\r');
            sig.update((byte)'\n');

            processLine(sig, lookAhead);
        }
    }

    sigIn.close();
}
```

Using the new input processing methods, we can put together a method for verifying a cleartext signed document. The following code shows how to write a verification method which assumes prior knowledge of the key used to sign the document.

```
/**
 * Verify the passed in cleartext signed document, returning true and a copy
 * of the data if successful, false otherwise.
 *
 * @param publicKey the public key to verify the signature with.
 * @param in the input containing the cleartext signed document.
 * @param msgOut the output stream to save the signed data in.
 * @return true if the document verifies, false otherwise.
 */
public static boolean verifyCleartextSignature(
    PGPPublicKey publicKey, InputStream in, OutputStream msgOut)
    throws IOException, PGPException
{
    ArmoredInputStream aIn = new ArmoredInputStream(in);

    // save a copy of the text for verification
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();

    int ch;
    while ((ch = aIn.read()) > 0 && aIn.isClearText())
    {
        bOut.write(ch);
    }

    //
    // load the signature record in the "PGP SIGNATURE" block.
    //
    JcaPGPObjectFactory pgpFact = new JcaPGPObjectFactory(aIn);
    PGPSignatureList p3 = (PGPSignatureList)pgpFact.nextObject();
    PGPSignature sig = p3.get(0);

    if (publicKey.getKeyID() != sig.getKeyID())
    {
        throw new PGPException("public key not for signature in document");
    }

    sig.init(
       new JcaPGPContentVerifierBuilderProvider().setProvider("BC"), publicKey);

    //
    // read the input, making sure we ignore the last newline.
    //
    InputStream sigIn = new ByteArrayInputStream(bOut.toByteArray());
44
45          processInput(sig, sigIn);
46
47          if (sig.verify())
48          {
49              msgOut.write(bOut.toByteArray());
50              msgOut.flush();
51
52              return true;
53          }
54
55          return false;
56      }
```

You can see on line 32 that the key ID in the signature is checked against the public key provided. For the simplicity of the example, we have assumed prior knowledge of the key as it is not until you are parsing the signature block associated with a cleartext signed document that you can work out what key was used to generate the signature. As you can see there is a key ID stored in the signature, so a simple modification to the method would be to pass a table of appropriately authorised public keys instead.

Other than requiring a public key, using the `verifyCleartextSignature()` really just requires an `OutputStream` to store the signed data in. The following code fragment demonstrates how to make use of the `verifyCleartextSignature()` method:

```
ByteArrayOutputStream msgOut = new ByteArrayOutputStream();

if (verifyCleartextSignature(
        dsaKeyPair.getPublicKey(),
        new ByteArrayInputStream(sigOut.toByteArray()),
        msgOut))
{
    System.out.println(Strings.fromByteArray(msgOut.toByteArray()));
}
else
{
    System.out.println("message failed to verify");
}
```

Assuming all has gone well, if you run it together with the previous code fragment for document creation you should be greeted with the following output.

```
Hello, world!
```

Of course, in the “real world”, things may not go according to plan. Assuming the public key is correct and everything else appears to be correct, it may just be that you are dealing with a legacy application. Where this is the case, it is worth checking that the sending application and yours are in agreement about the canonicalization method used (assuming the public key matches the signature and everything else looks correct). While RFC 4880 does document the standard approach that should be taken, it is not unusual for trailing spaces or new-lines to somehow be included in the signature when generated by legacy applications. Occasionally a bit of tweaking is required to get things to work.

Public Key Encryption

Public key encryption using PGP follows the usual pattern of generating a session key for a symmetric algorithm and then using the session key to encrypt the data and the recipient’s public key to encrypt the session key. The encrypted session key and encrypted data are then sent to recipient where the original plaintext is recovered using the recipient’s private key.

Here is a method for doing public key encryption. Have a read through it then we will go through it step by step.

```java
/**
 * Create an encrypted data blob using an AES-256 session key and the
 * passed in public key.
 *
 * @param encryptionKey the public key to use.
 * @param data the data to be encrypted.
 * @return a PGP binary encoded version of the encrypted data.
 */
public static byte[] createEncryptedData(
    PGPPublicKey encryptionKey,
    byte[] data)
    throws PGPException, IOException
{
    PGPEncryptedDataGenerator encGen = new PGPEncryptedDataGenerator(
        new JcePGPDataEncryptorBuilder(SymmetricKeyAlgorithmTags.AES_256)
            .setWithIntegrityPacket(true)
            .setSecureRandom(new SecureRandom()).setProvider("BC"));

    encGen.addMethod(
        new JcePublicKeyKeyEncryptionMethodGenerator(encryptionKey)
            .setProvider("BC"));

23          ByteArrayOutputStream encOut = new ByteArrayOutputStream();
24
25          // create an indefinite length encrypted stream
26          OutputStream cOut = encGen.open(encOut, new byte[4096]);
27
28          // write out the literal data
29          PGPLiteralDataGenerator lData = new PGPLiteralDataGenerator();
30          OutputStream pOut = lData.open(
31              cOut, PGPLiteralData.BINARY,
32              PGPLiteralData.CONSOLE, data.length, new Date());
33          pOut.write(data);
34          pOut.close();
35
36          // finish the encryption
37          cOut.close();
38
39          return encOut.toByteArray();
40      }
```

The code follows a few simple steps: setting up a `PGPEncryptedDataGenerator`, including adding an encryption method for encrypting the session key. Next writing out the data to be encrypted, and then finishing up.

The setup of the `PGPEncryptedDataGenerator` is over lines 14 to 21. The first section, from lines 14 to 17, specifies which symmetric key algorithm to use to encrypt the data we are trying to process and also specifies that an integrity packet should be used. Integrity packets are discussed in details in Section 5.13 of RFC 4880, but the short description is that specifying true adds a SHA-1 hash to the end of the encrypted data stream. In general you want to make use of integrity packets if you can, certainly if the data is not signed as well. PGP makes use of CFB8 mode for its symmetric encryption. CFB8 offers no assurances around protecting the encrypted data from tampering, and being a byte aligned xor mode, does make it quite easy to pick out particular bits in the ciphertext if the general format of the plaintext is known.

The second part of the `PGPEncryptedDataGenerator` setup is on lines 19-21 - the call to the `addMethod()` function. You might think this looks very similar to the Recipient Information we saw being added in CMS and you would be correct to do so. Specifying a method provides the generator with a mechanism for protecting the symmetric key used to encrypt the data so that our intended recipient can recover it. As with CMS you can add more than one method if you wish.

The steps that actually encrypt the data are between lines 26 and lines 37. The call to `encGen.open()` is used to initialise a stream that will encrypt any data we write to it - in this case it takes a buffer of size 4096[23] which is used to size the internal block size for an indefinite length encrypted stream. On lines 29 to 34, as this is PGP protocol we are concerned with, we construct a literal data object which

[23] Yes, it's an actual number, and no please don't do it like this real life! It's hard coded to make the mechanics of the example easier to follow.

we then write the data to. Finally on line 37 we finish by closing everything off and then return an encoding of the encrypted data message.

Decrypting the resulting data follows the same pattern we saw with CMS EncryptedData. Identify the recipient that matches the key you want to use and then read out the data, or in this case, the Literal Data structure wrapping the data.

The following code provides an example of how to recover the plaintext from the PGP encrypted data using a recipient private key.

```
/**
 * Extract the plain text data from the passed in encoding of PGP
 * encrypted data. The routine assumes the passed in private key
 * is the one that matches the first encrypted data object in the
 * encoding.
 *
 * @param privateKey the private key to decrypt the session key with.
 * @param pgpEncryptedData the encoding of the PGP encrypted data.
 * @return a byte array containing the decrypted data.
 */
public static byte[] extractPlainTextData(
    PGPPrivateKey privateKey,
    byte[] pgpEncryptedData)
    throws PGPException, IOException
{
    PGPObjectFactory pgpFact = new JcaPGPObjectFactory(pgpEncryptedData);

    PGPEncryptedDataList encList = (PGPEncryptedDataList)pgpFact.nextObject();

    // find the matching public key encrypted data packet.
    PGPPublicKeyEncryptedData encData = null;
    for (PGPEncryptedData pgpEnc: encList)
    {
        PGPPublicKeyEncryptedData pkEnc
            = (PGPPublicKeyEncryptedData)pgpEnc;
        if (pkEnc.getKeyID() == privateKey.getKeyID())
        {
            encData = pkEnc;
            break;
        }
    }

    if (encData == null)
    {
```

```
35             throw new IllegalStateException("matching encrypted data not found");
36         }
37 
38         // build decryptor factory
39         PublicKeyDataDecryptorFactory dataDecryptorFactory =
40             new JcePublicKeyDataDecryptorFactoryBuilder()
41                 .setProvider("BC")
42                 .build(privateKey);
43 
44         InputStream clear = encData.getDataStream(dataDecryptorFactory);
45         byte[] literalData = Streams.readAll(clear);
46         clear.close();
47 
48         // check data decrypts okay
49         if (encData.verify())
50         {
51             // parse out literal data
52             PGPObjectFactory litFact = new JcaPGPObjectFactory(literalData);
53             PGPLiteralData litData = (PGPLiteralData)litFact.nextObject();
54             byte[] data = Streams.readAll(litData.getInputStream());
55             return data;
56         }
57         throw new IllegalStateException("modification check failed");
58     }
```

As with signing, we use the PGP key ID to match the encrypted data block holding the wrapped session key with the private key we have. Note the iteration loop has a loop variable of type `PGPEncryptedData`. We cannot use `PGPPublicKeyEncryptedData` here as there is also a type for PBE based keys which we will look at in the next section.

Line 49 of the `extractPlainTextData()` is where we check the integrity packet to check that the encrypted data has not been tampered with. Once we have confirmed no tampering is evident, it is safe to parse the LiteralData structure and return the wrapped data to the user. If the `encData.verify()` statement returns false, we know something has gone wrong and through an exception. Keep in mind that as the Bouncy Castle PGP API is a streaming API and the integrity packet is at the end of the stream, the `encData.verify()` method can only be called once the encrypted data has been fully processed. If you call `verify()` early the method will read the stream internally for you, possibly not what you are after if you are planning to process the data yourself!

If you are dealing with PGP encrypted data which may not have an integrity protection packet on it, you can call the boolean `PGPEncryptedData.isIntegrityProtected()` method to tell you if the check is possible. If you call `PGPEncryptedData.verify()` on encrypted data that is not integrity protected, the API will assume the worst and throw an exception.

Public Key Encryption using Elgamal

PGP provides support for a range of public key algorithms. The oldest supported algorithms are RSA and Elgamal which both use PKCS#1 encoding for wrapping the symmetric key[24]. The following code fragment shows how to use the encryption and decryption methods using Elgamal.

```
byte[] msg = Strings.toByteArray("Hello, world!");

KeyPairGenerator kpGen = KeyPairGenerator.getInstance("DH", "BC");
kpGen.initialize(2048);

KeyPair kp = kpGen.generateKeyPair();

PGPKeyPair elgKp = new JcaPGPKeyPair(
    PGPPublicKey.ELGAMAL_ENCRYPT, kp, new Date());

byte[] encData = createEncryptedData(elgKp.getPublicKey(), msg);

byte[] decData = extractPlainTextData(elgKp.getPrivateKey(), encData);

System.out.println(Strings.fromByteArray(decData));
```

If you wanted to use RSA with the above methods you would just replace “DH” with “RSA” and use `PublicKey.RSA_ENCRYPT` instead of `PGPPublicKey.ELGAMAL_ENCRYPT`. Assuming all goes well, running the fragment should just produce the following output in both cases:

```
Hello, world!
```

Public Key Encryption using ECDH

While RFC 4880 does not talk about the use of Elliptic Curve cryptography, an update, RFC 6637 [85], offers public key encryption with key agreement using Elliptic Curves, originally for the purposes of meeting the NSA’s Suite B criteria [127]. As you would expect this is based on EC Diffie-Hellman, but unlike CMS does not require a special recipient, the PGP protocol allows the EC key type to appear to be used like any other asymmetric encryption key, so we just need a simple modification to the code fragment we looked at earlier for supporting Elgamal.

Here is the new code fragment for supporting EC in PGP encryption.

[24] Note the use of PKCS#1 encoding means this approach is not FIPS compliant.

```
        byte[] msg = Strings.toByteArray("Hello, world!");

        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");
        kpGen.initialize(new ECGenParameterSpec("P-256"));

        KeyPair kp = kpGen.generateKeyPair();

        PGPKeyPair ecdhKp = new JcaPGPKeyPair(PGPPublicKey.ECDH, kp, new Date());

        byte[] encData = createEncryptedData(ecdhKp.getPublicKey(), msg);

        byte[] decData = extractPlainTextData(ecdhKp.getPrivateKey(), encData);

        System.out.println(Strings.fromByteArray(decData));
```

As you can see the only real change is in the creation and initialization of the `KeyPairGenerator` and the algorithm specified in the encryption step. Run this fragment and you should also see:

```
Hello, world!
```

as we did before.

A Final Note on Public Key Encryption

It is actually fairly rare to find messages encrypted using public keys with only one `PGPEncryptedData` object inside them. Generally there are two keys as the sender often encrypts the message so that they can decrypt it later as well.

PBE Encryption

PGP also supports a method for password based encryption by using a different encryption method object to the one we have already seen with public key encryption. Bouncy Castle supports the PGP PBE method via the `JcePBEKeyEncryptionMethodGenerator` which takes a password in its constructor and does the necessary work internally to wrap the symmetric session key.

The following method shows the use of the `JcePBEKeyEncryptionMethodGenerator`. As you read the code you will note, while the literal data is processed differently, that it is really only the method being added to the `PGPEncryptedDataGenerator` that has changed from the previous example we saw with the public key method. The code for establishing the data encryptor being used and the session key is the same. Here is the code:

```java
/**
 * Create an encrypted data blob using an AES-256 key wrapped using
 * a PBE based key.
 *
 * @param passwd the password to apply to generate the PBE based key.
 * @param data the data to be encrypted.
 * @return a PGP binary encoded version of the encrypted data.
 */
public static byte[] createPBEEncryptedData(
    char[] passwd,
    byte[] data)
    throws PGPException, IOException
{
    // create the packet containing the plaintext.
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();

    PGPLiteralDataGenerator lData = new PGPLiteralDataGenerator();
    OutputStream pOut = lData.open(
        bOut, PGPLiteralData.BINARY, PGPLiteralData.CONSOLE,
        data.length, new Date());
    pOut.write(data);
    pOut.close();

    byte[] plainText = bOut.toByteArray();

    // set up the generator
    PGPEncryptedDataGenerator encGen = new PGPEncryptedDataGenerator(
        new JcePGPDataEncryptorBuilder(SymmetricKeyAlgorithmTags.AES_256)
            .setWithIntegrityPacket(true)
            .setSecureRandom(new SecureRandom())
            .setProvider("BC"));
    encGen.addMethod(
        new JcePBEKeyEncryptionMethodGenerator(passwd).setProvider("BC"));

    // encryption step.
    ByteArrayOutputStream encOut = new ByteArrayOutputStream();

    OutputStream cOut = encGen.open(encOut, plainText.length);
    cOut.write(plainText);
    cOut.close();

    return encOut.toByteArray();
}
```

Note that the only thing passed into the JcePBEKeyEncryptionMethodGenerator is the password, which in this case means the password is going to be turned into a secret key by using an algorithm based on SHA-1. It is possible to specify the algorithm used to convert the password by passing in a `PGPDigestCalculator` to the constructor as well[25].

The difference in the processing of the literal data is to show how to create a definite length encryption object, rather then the indefinite length one we used earlier. Precomputing the plaintext allows us to use the `open()` method which takes a length rather than a buffer. We could have used the indefinite length version of the `open()` method instead. Likewise, we could have used the defined length method earlier with the public key example.

In the case of the extraction code we need to recognise a different subclass for `PGPEncryptedData`, the one for PBE based data - `PGPPBEEncryptedData`. There is no ID associated with this type, so if you need to process public key encrypted and PBE encrypted data in the same code, you will need to rely on `instanceof` to tell them apart.

Here is the code for handling PBE based decryption.

```java
/**
 * Extract the plain text data from the passed in encoding of PGP
 * encrypted data. The routine assumes the password is matched
 * with the first encrypted data object in the encoding.
 *
 * @param passwd the password to apply to generate the PBE based key.
 * @param pgpEncryptedData the encoding of the PGP encrypted data.
 * @return a byte array containing the decrypted data.
 */
public static byte[] extractPlainTextPBEData(
    char[] passwd,
    byte[] pgpEncryptedData)
    throws PGPException, IOException
{
    PGPEncryptedDataList encList = new PGPEncryptedDataList(pgpEncryptedData);

    // assume the PBE encrypted data is first.
    PGPPBEEncryptedData encData = (PGPPBEEncryptedData)encList.get(0);
    PBEDataDecryptorFactory dataDecryptorFactory =
        new JcePBEDataDecryptorFactoryBuilder()
            .setProvider("BC")
            .build(passwd);

    InputStream clear = encData.getDataStream(dataDecryptorFactory);
    byte[] literalData = Streams.readAll(clear);
```

[25] While the possibilities do include the SHA-2 family of digests, the PBE algorithm is not FIPS compliant.

```
        clear.close();

        // check data decrypts okay
        if (encData.verify())
        {
            PGPLiteralData litData = new PGPLiteralData(literalData);
            return Streams.readAll(litData.getInputStream());
        }

        throw new IllegalStateException("modification check failed");
    }
```

Note that we have assumed the `PGPPBEEncryptedData` object is the first one in the `PGPEncryptedDataList` we have read in. The `JcePBEDataDecryptorFactoryBuilder` class, in this case, just requires the password and the provider to use. In some situations the provider for the digest services, required for password to key conversion, and the provider for cipher services, required for the actual ciphertext to plaintext conversion, might be different. If this is the case the `JcePBEDataDecryptorFactoryBuilder` can be passed a digest calculator provider on construction so that only the provider specified to it is only used for the creation of the cipher required.

The following sample code will exercise the PBE encryptor and decryptor.

```
    byte[] msg = Strings.toByteArray("Hello, PBE world!");
    char[] password = "secret".toCharArray();

    byte[] encData = createPBEEncryptedData(password, msg);

    byte[] decData = extractPlainTextPBEData(password, encData);

    System.out.println(Strings.fromByteArray(decData));
```

And when run should produce the following output:

```
Hello, PBE world!
```

⚠ Keep in mind when parsing an encrypted PGP data stream that it may have multiple `PGPEncryptedData` objects in it for different public keys and possibly even passwords.

Keybox Files

Keybox files were introduced in GnuPG 2.1, largely to address performance issues that occurred with large collections of public keys stored in a single pubring.gpg file. Where keybox files are being used you will find a pubring.kbx file instead of the old-style pubring.gpg. The format is described in a GnuPG source file keybox-blob.c [128].

Keybox Blobs

Keybox files store public key data as a series of blobs with associated meta-data. It is the meta-data that allows for the faster parsing. For our purposes the most interesting blobs generally contain one of two formats of public key data. Either public key data as an X.509 certificate or public key data in the original PGP encoded format for a PGP key ring.

There are 4 blob types: Empty blob, First blob, Open-PGP blob, and X.509 blob. Empty blob is what you would expect a blob without data. First blob is the first blob in a keybox file and contains header and version information. An Open-PGP blob contains an PGP encoded public key ring and its associated meta-data and finally an X.509 blob contains an X.509 public key certificate and its associated meta-data. In the Bouncy Castle API the last two are represented by `CertificateBlob` and `PublicKeyRingBlob` and share common ancestor `KeyBlob` which carries access methods for the meta-data in the respective blobs.

Loading a Keybox File

You can think of a keybox file as a list of blobs and an example method for loading all the key data from one could look as follows:

```
/**
 * Load all the key rings and certificates in a KeyBox file.
 *
 * @param keyBox the KeyBox to be loaded.
 * @return a list of the PGP public key rings and X.509 certificates
 *         found in the key box.
 */
public static List<Object> loadKeyBoxKeys(KeyBox keyBox)
    throws Exception
{
    List<Object> publicObjects = new ArrayList<Object>();

    for (KeyBlob keyBlob : keyBox.getKeyBlobs())
    {
        switch (keyBlob.getType())
        {
```

```
        case X509_BLOB:
            {
            byte[] certData = ((CertificateBlob)keyBlob).getEncodedCertificate();
            CertificateFactory factory = CertificateFactory.getInstance("X509");
            publicObjects.add(factory.generateCertificate(
                                            new ByteArrayInputStream(certData)));
            }
            break;
        case OPEN_PGP_BLOB:
            publicObjects.add(((PublicKeyRingBlob)keyBlob).getPGPPublicKeyRing());
            break;
        default:
            throw new IllegalStateException(
                          "Unexpected blob type: " + keyBlob.getType());
        }
    }

    return publicObjects;
}
```

Creating the actual KeyBox object is done via a builder - as the parsing of the public key information requires the use of cryptographic services you may want to specify a particular provider to use. After specifying the provider to the builder you can then produce a KeyBox by providing a byte encoding (or an InputStream) as follows:

```
KeyBox kbx = new JcaKeyBoxBuilder().setProvider("BC").build(kbxData);
```

where kbxData is either a byte[] or an InputStream containing the encoding of the keybox.

Once you have done that, you can just retrieve the key blobs as a list of objects like this:

```
List<Object> kbxObjs = loadKeyBoxKeys(kbx);
```

Keep in mind if you are writing code for parsing ".kbx" files, you can skip blobs without completely parsing them if you need to. So, if you are needing to deal with a large number of blobs, it will a lot quicker to skip through the stream and make use of the meta-data on each KeyBlob to select the correct blob and then scan the data.

Using S-expressions for Private Keys

In addition to the use of the ".kbx" file, GnuPG has also introduced the use of S-expressions for storing private keys, as an alternative to using the PGP encoding in the old "secring.gpg" file. As the name suggests S-expressions are a variant of Lisp S-expressions and were initially proposed for key encoding by Ron Rivest and Butler Lampson and others at MIT in the late 1990s [130]. Details of the format are provided in section 6.2 of the Libgcrypt Reference Manual [131] with some additional information in the file keyformat.txt [132].

A simple summary is that the fields in values in S-expression encoded keys follow are encoded as "length:value_bytes". Structures can be built up by containing the concatenation of fields in open and close parentheses. The length field is ascii (so, for example, the string "ecc" is encoded as "3:ecc"), but the value_bytes can be binary data so a PGP key encoded using S-expression syntax will not be full readable in a text editor, especially where encrypted.

Bouncy Castle now provides a parser, `SExprParser`, for reading these files. The class requires a `PGPDigestCalculatorProviderBuilder` on construction as some keys are written out with a message digest of the key included in the S-expression. Usage is fairly simple as each file can contain only one key. An example of calling the parser follows here:

```
PGPSecretKey secretKey = new SExprParser(
    new JcaPGPDigestCalculatorProviderBuilder().build()).parseSecretKey(
                inputStream,
                new JcePBEProtectionRemoverFactory("test".toCharArray()),
                new JcaKeyFingerprintCalculator());
```

The need for some of the method parameters are self-evident, `inputStream` is a stream containing the encrypted secret key, and the JcaFingerPrintCalculator is needed for calculating the key's fingerprint when the `PGPSecretKey` object is being built. The new class, `JcePBEProtectionRemoverFactory`, is required as, unlike the old encoding of PGP secret keys, the S-expression usually stores the PGP secret key as an encrypted blob so the password is needed to decrypt the PGP secret key. Usually this means that the `PGPPrivateKey` carried by the PGP secret key can then be extracted without having to make use of a `PBESecretKeyDecryptor` so a `null` can be passed in instead, as in:

```
PGPPrivateKey privateKey = secretKey.extractPrivateKey(null);
```

Summary

In this chapter we have examined another cryptographic protocol which is widely used across the Internet - OpenPGP.

We saw that OpenPGP was, in many ways, similar to CMS and also included its own certificate format and its own approaches for encoding and protecting private keys. Finally we also saw that recent developments in GnuPG, a widely used and well regarded tool for supporting Open PGP has also added some scope for introducing X.509 certificates into the new PGP KeyBox ".kbx" file format.

Chapter 14: Transport Layer Security

Introduction

This chapter is a guide to the use of the Legion of the Bouncy Castle DTLS/TLS APIs and JSSE provider. The BC DTLS/TLS APIs and JSSE provider follow a similar model to that of the BC JCA/JCE provider. The DTLS/TLS APIs provide a low-level mechanism for making use of the DTLS and TLS protocols which is more flexible than what is provided by the JSSE APIs, but requires more knowledge and care on the part of the developer. The BC JSSE provider is built on top of the TLS API and provides functionality to support the use of the JSSE API defined in javax.net.ssl.

Installation

The bctls jar can be installed in either jre/lib/ext for your JRE/JDK, or in many cases, on the general class path defined for the application you are running. In the event you do install the bctls jar on the general class path be aware that sometimes the system class loader may make it impossible to use the classes in the bctls jar without causing an exception depending on how elements of the JSSE provider are loaded elsewhere in the application.

BCJSSE Provider Installation Into the JRE

Once the bctls jar is installed, the provider class BouncyCastleJsseProvider may need to be installed if it is required in the application globally. Installation of the provider can be done statically in the JVM by adding it to the provider definition to the java.security file in in the jre/lib/security directory for your JRE/JDK. The provider can also be added during execution. If you wish to add the provider to the JVM globally during execution you can add the following imports to your code:

```
import java.security.Security
import org.bouncycastle.jsse.provider.BouncyCastleJsseProvider
```

Then insert the line

```
Security.addProvider(new BouncyCastleJsseProvider());
```

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

```
SSLContext clientContext = SSLContext.getInstance("TLS", “BCJSSE”);
```

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

```
SSLContext clientContext =
        SSLContext.getInstance("TLS", new BouncyCastleJsseProvider());
```

Configuring the BCJSSE Provider in FIPS Mode

The BCJSSE provider has a FIPS mode which helps restricts the provider to cipher suites and parameters that can be offered in a FIPS compliant TLS client, or server, setup. FIPS mode is indicated by passing the string “fips:<provider>”” to the constructor of the BouncyCastleJsseProvider class, either at runtime or via the java.security file for the JVM. Acceptable values of <provider> include “default” which indicates that providers will be used according to the provider prioty list for the JVM. Alternately a provider name, such as “BC”, or “BCFIPS” if you are using the Bouncy Castle FIPS provider, can be used. Where a specific provider name is used only algorithms from that provider will be accepted by the BCJSSE provider when encryption operations are done.

For example, if we wanted to use the BCFIPS provider and make use of the BCJSSE provider with it set in FIPS mode rather than just having the following in the security.provider section of the java.security file in the JVM:

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

a minimal install would instead also need the second line to include the “fips:BCFIPS” declaration at the end[26]:

```
security.provider.2=org...provider.BouncyCastleJsseProvider fips:BCFIPS
```

At runtime a similar configuration can be achieved using:

```
Security.addProvider(new BouncyCastleJsseProvider("fips:BCFIPS"));
```

[26] Note the full `org.bouncycastle.jsse.provider.BouncyCastleJsseProvider` needs to be there. We’ve used the dots to avoid a potentially confusing line-break.

Other Differences Between the BCJSSE and JSSE Provider

The `TrustManager` objects produced by the BCJSSE provider will ignore private keys in any KeyStores passed into to their factories. Only simple certificate entries in a KeyStore will be used to create a `TrustManager`. BCJSSE also supports ChaCha20 and Poly1305 in non-FIPS mode. Support for this cipher suite is available in the BCFIPS provider from 1.0.2.

Further on there is a list of security properties, for the regular JSSE if the properties are absent they are usually treated as unset. As the BCJSSE runs across a range of JVMs it will generally provide default settings based on what is used in the most recent JVM if a property the BCJSSE handles is not set.

Conventions

Use of getDefault()

Things like `SSLSocketFactory.getDefault()` will return the class from the BCJSSE provider only if the BCJSSE provider is either ahead of, or replacing the regular JRE JSSE provider.

Instance Names

The core `KeyManagerFactory` has the instance name "X.509", and the aliases "X509" and "PKIX". The core `TrustManagerFactory` has the instance name "PKIX", and the aliases "X.509" and "X509". `SSLContext` instances are available for "TLSv1", "TLSv1.1", and "TLSv1.2". In each case the name of the instance represents the lowest version of TLS supported and higher versions are supported as well. The `SSLContext` instance names "TLS" and DEFAULT provide support for TLSv1.2.

Some Preparation - a Simple Protocol

TLS is, of course, is not an end in itself. There is always the assumption that something is being communicated on top of it. The following class provides a simple protocol that our examples can work with where the messages are terminated with a '!' character.

```java
/**
 * Simple protocol to execute.
 */
public class Protocol
{
    private static Logger LOG = Logger.getLogger(Protocol.class.getName());

    /**
     * Carry out the '!' protocol - client side.
     */
    static void doClientSide(
        Socket cSock)
        throws IOException
    {
        OutputStream out = cSock.getOutputStream();
        InputStream in = cSock.getInputStream();
        out.write(Strings.toByteArray("world"));
        out.write('!');
        int ch = 0;
        while ((ch = in.read()) != '!')
        {
            System.out.print((char)ch);
        }
        System.out.println((char)ch);
    }

    /**
     * Carry out the '!' protocol - server side.
     */
    static void doServerSide(
        Socket sSock)
        throws IOException
    {
        LOG.log(Level.INFO, "session started.");
        InputStream in = sSock.getInputStream();
        OutputStream out = sSock.getOutputStream();
        out.write(Strings.toByteArray("Hello, "));
        int ch = 0;
        while ((ch = in.read()) != '!')
        {
            out.write(ch);
        }
        out.write('!');
        sSock.close();
        LOG.log(Level.INFO, "session closed.");
    }
}
```

As you can see the protocol is a two-way affair, where the client connects, sends a message, and then reads back what the server sends it, echoing it to stdout. After an initial message the server method echos what is sent to it down the socket until it receives a '!' and then sends a terminating '!' of its own. The client method sends its half of the message and then reads back what is sent echoing it to stdout until it also receives a '!' and then exits. Enough to prove a connection happened.

A Basic TLS Server

A basic server only requires a `KeyManager` to identify itself with. The actual `KeyManager` is then created by a `KeyManagerFactory` which has been initialised using the server's key store and key store password.

```
public static final int PORT_NO = 9090;
public static final char[] ID_STORE_PASSWORD = "passwd".toCharArray();


/**
 * Create a KeyStore containing a single key with a self-signed certificate.
 *
 * @return a KeyStore containing a single key with a self-signed certificate.
 */
public static KeyStore createIdentityKeyStore()
    throws Exception
{
    PrivateCredential cred = createSelfSignedCredentials();

    KeyStore store = KeyStore.getInstance("JKS");

    store.load(null, null);

    store.setKeyEntry("identity", cred.getPrivateKey(), ID_STORE_PASSWORD,
                            new Certificate[]{cred.getCertificate()});

    return store;
}
```

Note: in this case we assume that the BC and BCJSSE providers have just been added to the provider list - no attempt has been made to replace the regular JSSE provider, so SSLSocketFactory.getDefault() will still refer to the regular JSSE provider.

```java
/**
 * Basic TLS server - using the '!' protocol.
 */
public class BasicTLSServer
{
    private static Logger LOG = Logger.getLogger(BasicTLSServer.class.getName());

    private final CountDownLatch readyLatch = new CountDownLatch(1);

    private final KeyStore serverStore;
    private final char[] passwd;

    /**
     * Base server constructor.
     *
     * @param serverStore the public/private keys for the server.
     * @param passwd the password to unlock the private keys in the serverStore.
     */
    BasicTLSServer(KeyStore serverStore, char[] passwd)
    {
        this.serverStore = serverStore;
        this.passwd = passwd;
    }

    /**
     * Task for bringing up a server-side TLS connection.
     */
    public void start()
        throws InterruptedException
    {
        new Thread(new ServerTask()).start();

        readyLatch.await();
    }

    /**
     * Task for bringing up a server-side of the TLS connection.
     */
    private class ServerTask
        implements Runnable
    {
        public void run()
        {
            try
            {
                SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");

                KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance(
                    "PKIX", "BCJSSE");
                keyMgrFact.init(serverStore, passwd);

                sslContext.init(keyMgrFact.getKeyManagers(), null, null);

                SSLServerSocketFactory fact = sslContext.getServerSocketFactory();
                SSLServerSocket sSock =
                    (SSLServerSocket)fact.createServerSocket(PORT_NO);

                readyLatch.countDown();

                SSLSocket sslSock = (SSLSocket)sSock.accept();

                Protocol.doServerSide(sslSock);
            }
            catch (Exception e)
            {
                LOG.log(Level.SEVERE, "server: " + e.getMessage(), e);
            }
        }
    }
}
```

You will notice that the server makes use of a `CountDownLatch` in its `start()` method. The reason for this is that the server needs to be ready to accept connections before it can signal to the caller of `start()` that the start process has completed. It is worth keeping this trick in mind - writing tests that rely on multi-threading to emulate clients and servers becomes very difficult otherwise.

A Basic TLS Client

In the case of a basic client the only requirement is that is has some way of identifying the server. This is done by providing a `TrustManager` to validate incoming credentials. The `TrustManager` is created using a `TrustManagerFactory` which has been initialized with a key store containing the certificates needed to validate the certificate path the server will present. In our case we can create a suitable key store from our server key store as follows:

```java
  /**
   * Create a key store suitable for use as a trust store, containing only
   * the certificates associated with each alias in the passed in
   * credentialStore.
   *
   * @param credentialStore key store containing public/private credentials.
   * @return a key store containing only certificates.
   */
  public static KeyStore createTrustStore(KeyStore credentialStore)
      throws Exception
  {
      KeyStore store = KeyStore.getInstance("JKS");

      store.load(null, null);

      for (Enumeration<String> en = credentialStore.aliases(); en.hasMoreElements();)
      {
          String alias = en.nextElement();

          store.setCertificateEntry(alias, credentialStore.getCertificate(alias));
      }

      return store;
  }
```

Now that we have our trust store we need to create a client to use it with. The following code implements a `Runnable` that provides a TLS client that will work with the ‘!’ protocol that the server implements.

```java
/**
 * Basic TLS client - using the '!' protocol.
 */
public class BasicTLSClient
    implements Runnable
{
    private static Logger LOG = Logger.getLogger(BasicTLSClient.class.getName());

    private final KeyStore trustStore;

    /**
     * Base client constructor.
     *
     * @param trustStore the certificates we are willing to trust from a server.
     */
    public BasicTLSClient(KeyStore trustStore)
    {
        this.trustStore = trustStore;
    }

    /**
     * Task for bringing up a TLS client.
     */
    public void run()
    {
        try
        {
            SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");

            TrustManagerFactory trustMgrFact =
                            TrustManagerFactory.getInstance("PKIX", "BCJSSE");
            trustMgrFact.init(trustStore);

            sslContext.init(null, trustMgrFact.getTrustManagers(), null);

            SSLSocketFactory fact = sslContext.getSocketFactory();
            SSLSocket cSock = (SSLSocket)fact.createSocket("localhost", PORT_NO);

            Protocol.doClientSide(cSock);
        }
        catch (Exception e)
        {
            LOG.log(Level.SEVERE, "client: " + e.getMessage(), e);
        }
    }
}
```

We can see that the code only differs from the server in that rather than providing a `KeyManager` to the `SSLContext.init()` method we provide `TrustManager` object built using a key store containing the server's certificate.

The next step is to put both classes together and see the server and client in action. The following code fragment will set up the key store for our server and then create the server and client tasks to execute the protocol:

```
        KeyStore serverStore = createIdentityKeyStore();

        BasicTLSServer server = new BasicTLSServer(serverStore, ID_STORE_PASSWORD);

        server.start();

        new Thread(new BasicTLSClient(createTrustStore(serverStore))).start();
```

We need to start the server first in the example code so that the client has something to connect to and we know this is safe to do once `server.start()` returns[27]. As both our client and server handle only one connection before exiting no clean up is required.

Assuming everything has gone to plan, and in this case that assumes cipher suites built around Elliptic Curve are available, you should get the following output from executing the sample above[28]:

```
Jan 04, 2019 7:02:19 PM chapter14.Protocol doServerSide
INFO: session started.
Hello, world!
Jan 04, 2019 7:02:19 PM chapter14.Protocol doServerSide
INFO: session closed.
```

indicating that the server started correctly, the client connected correctly, they executed the protocol and then both exited.

Using Client-Side Authentication

Once client-side authentication is involved the `SSLContext` for the client also needs to be initialised with key managers, and we need to introduce trust managers, for validating the client, to the server side.

A TLS Server with Client-Side Authentication

Client authentication can be compulsory or optional. Where the server should require client authentication the `SSLServerSocket.setNeedClientAuth()` method needs to be called with `true`.

Other than that, the only changes are related to the setting up of the `SSLContext` in its `init()` method, we need to configure a `TrustManagerFactory` and then pass the created `TrustManager` objects to the `init()` in addition to the server's `KeyManager`.

Here is updated server code incorporating the use of the `TrustManagerFactory`.

[27] Remember, we know `server.start()` will not return too early as we have used the `CountDownLatch`.

[28] Depending on the timing, the output from the client and the log output might be intermingled as we are looking at the output from two distinct threads.

```
/**
 * TLS server with client-side authentication - using the '!' protocol.
 */
public class TLSServerWithClientAuth
{
    private static Logger LOG = Logger.getLogger(BasicTLSServer.class.getName());

    private final CountDownLatch readyLatch = new CountDownLatch(1);

    private final KeyStore serverStore;
    private final char[] passwd;
    private final KeyStore trustStore;

    /**
     * Base server constructor.
     *
     * @param serverStore the public/private keys for the server.
     * @param passwd the password to unlock the private keys in the serverStore.
     * @param trustStore the trust store for validating any certificates presented.
     */
    public TLSServerWithClientAuth(
        KeyStore serverStore, char[] passwd, KeyStore trustStore)
    {
        this.serverStore = serverStore;
        this.passwd = passwd;
        this.trustStore = trustStore;
    }

    /**
     * Start our server thread.
     */
    public void start()
        throws InterruptedException
    {
        new Thread(new ServerTask()).start();

        readyLatch.await();
    }

    /**
     * Task for bringing up a server-side of the TLS connection.
     */
    private class ServerTask
```

```
        implements Runnable
    {
        public void run()
        {
            try
            {
                SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");

                KeyManagerFactory keyMgrFact =
                                KeyManagerFactory.getInstance("PKIX", "BCJSSE");
                keyMgrFact.init(serverStore, passwd);

                TrustManagerFactory trustMgrFact =
                                TrustManagerFactory.getInstance("PKIX", "BCJSSE");
                trustMgrFact.init(trustStore);

                sslContext.init(
                    keyMgrFact.getKeyManagers(),
                    trustMgrFact.getTrustManagers(), null);

                // create the server socket and make client auth mandatory.
                SSLServerSocketFactory fact = sslContext.getServerSocketFactory();
                SSLServerSocket sSock =
                    (SSLServerSocket)fact.createServerSocket(PORT_NO);
                sSock.setNeedClientAuth(true);

                readyLatch.countDown();

                SSLSocket sslSock = (SSLSocket)sSock.accept();

                Protocol.doServerSide(sslSock);
            }
            catch (Exception e)
            {
                LOG.log(Level.SEVERE, "server: " + e.getMessage(), e);
            }
        }
    }
}
```

Note that, in this case, we are calling `SSLServerSocket.setNeedClientAuth()` method with `true`, meaning that this server will only accept connections from a client that can authenticate itself.

A TLS Client with Client-Side Authentication

Just as a TrustManager needs to be introduced to the server to allow it to identify the client a KeyManager also needs to be added to the client's SSLContext to allow the client to identify itself to the server.

The code example that follows shows a basic client that is capable of authenticating itself to a server. Note that the trust anchor for the certificate path identifying the client must also have been made available to the TrustManager created for the server.

```
/**
 * TLS client with client-side authentication - using the '!' protocol.
 */
public class TLSClientWithClientAuth
    implements Runnable
{
    private static Logger LOG = Logger.getLogger(BasicTLSClient.class.getName());

    private final KeyStore trustStore;
    private final KeyStore clientStore;
    private final char[] passwd;

    /**
     * Base client with authentication constructor.
     *
     * @param trustStore the certificates we are willing to trust from a server.
     * @param clientStore the public/private keys for the client.
     * @param passwd the password to unlock the private keys in the clientStore.
     */
    public TLSClientWithClientAuth(
        KeyStore trustStore, KeyStore clientStore, char[] passwd)
    {
        this.trustStore = trustStore;
        this.clientStore = clientStore;
        this.passwd = passwd;
    }

    /**
     * Task for bringing up a TLS client with authentication.
     */
    public void run()
    {
        try
        {
            SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");

            KeyManagerFactory keyMgrFact =
                            KeyManagerFactory.getInstance("PKIX", "BCJSSE");
            keyMgrFact.init(clientStore, passwd);

            TrustManagerFactory trustMgrFact =
                            TrustManagerFactory.getInstance("PKIX", "BCJSSE");
            trustMgrFact.init(trustStore);

            sslContext.init(
                keyMgrFact.getKeyManagers(),
                trustMgrFact.getTrustManagers(), null);

            SSLSocketFactory fact = sslContext.getSocketFactory();
            SSLSocket cSock = (SSLSocket)fact.createSocket("localhost", PORT_NO);

            Protocol.doClientSide(cSock);
        }
        catch (Exception e)
        {
            LOG.log(Level.SEVERE, "client: " + e.getMessage(), e);
        }
    }
}
```

As you can see in the example, from the client's point of view, simply providing the KeyManager is enough. No additional settings for this are required.

Another Look a TrustManagers

In many cases the way the server is handling client validation in our example is not enough. Usually client certificates will be allocated under a CA, and the CA might also periodically need to revoke some.

In order to give developers more control over the validation of client certificates the `TrustManagerFactory.init()` method can also take an implementation of the `ManagerFactoryParameters` interface. One of the provided implementations of `ManagerFactoryParameters` is the `CertPathTrustManagerParameters` which allows you to provide a `PKIXParameters` object which will be used to manage the certificate path validation done on any presented certificate that a trust manager needs to check. For the simple server example with client authentication we looked at earlier, you can get the same result we saw earlier by replacing the `TrustManagerFactory` creation code with the following fragment:

```
        TrustManagerFactory trustMgrFact =
                        TrustManagerFactory.getInstance("PKIX", "BCJSSE");

        PKIXParameters pkixParameters = new PKIXParameters(trustStore);

        trustMgrFact.init(new CertPathTrustManagerParameters(pkixParameters));
```

Internally the `PKIXParameters` object will just convert `trustStore` into a set of trust anchors. In the simple example case we do not need to do much more as the scenario is built around self-signed certificates. In the more general case you would want to also initialise the `PKIXParameters` object with CA certificates and, optionally, CRLs or a `PKIXCertPathChecker` for managing revocation instead.

Properties

Configuration of the JSSE provider with properties is constantly evolving so the list of properties is regularly updated, at the time of writing here is the current list, but it would also be worth checking Appendix A and Appendix B of the latest version of "Java (D)TLS API and JSSE Provider User Guide" which you should be able to find on the bouncycastle.org website.

Security Properties

Some regular JVM security properties are also used by the BCJSSE provider.

> Keep in mind that, as the BCJSSE runs across a range of JVMs, the BCJSSE will generally provide default settings based on what is used in the most recent JVM if a property the BCJSSE handles is not set.

Here are the ones that are currently respected:

keystore.type: The default keystore type (via `java.security.KeyStore.getDefaultType()`).

keystore.type.compat: Controls compatibility mode for JKS and PKCS12 keystore types.

jdk.certpath.disabledAlgorithms: Algorithm restrictions for certification path (CertPath) processing.

jdk.tls.disabledAlgorithms: Algorithm restrictions for TLS processing.

ssl.KeyManagerFactory.algorithm: The default key manager factory algorithm (via `javax.net.ssl.KeyManagerFactory.getDefaultAlgorithm()).`

ssl.TrustManagerFactory.algorithm: The default trust manager factory algorithm (via `javax.net.ssl.TrustManagerFactory.getDefaultAlgorithm()`).

Standard System Properties

The following standard system properties are currently used by the BCJSSE provider:

com.sun.net.ssl.checkRevocation: Used to enable revocation checking when the PKIX trust manager creates default PKIX parameters.

com.sun.net.ssl.requireCloseNotify: Whether to require a close_notify warning alert is sent before a connection is closed, as specified in TLS but not always implemented correctly. BCJSSE uses a default value of "true".

java.home: Java installation directory.

javax.net.ssl.keyStore: Name of the key store file that holds the endpoint's private key. This is used to create a KeyManager internally.

javax.net.ssl.keyStorePassword: Password to use to get access to the key store file holding the endpoint's private key.

javax.net.ssl.keyStoreType: Type of the key store file holding the endpoint's private key.

javax.net.ssl.keyStoreProvider: The name of the provider which provides implementation support for the key store specified in the javax.net.ssl.keyStore property.

javax.net.ssl.sessionCacheSize: Configures the default value of the 'sessionCacheSize' property for session contexts; see SSLSessionContext.setSessionCachSize(int).

javax.net.ssl.trustStore: Name of the key store file that holds the trust anchors for validating the end points we are talking to. This is used to create a TrustManager internally.

javax.net.ssl.trustStorePassword: Password to use to get access to the key store file holding the trust anchors.

javax.net.ssl.trustStoreType: Type of the key store file holding the trust anchors.

javax.net.ssl.trustStoreProvider: The name of the provider which provides implementation support for the key store specified in the javax.net.ssl.trustStore property.

jdk.tls.allowLegacyMasterSecret: Whether to permit connections that did not negotiate the extended_master_secret extension.

jdk.tls.allowLegacyResumption: Whether to permit resumption of a session where the original connection did not negotiate the extended_master_secret extension.

jdk.tls.client.cipherSuites: Specify the default enabled cipher suites for TLS clients, in a comma-separated list.

jdk.tls.client.enableCAExtension: If set to "true", enables the certificate_authorities extension, an optional ClientHello extension introduced in TLS 1.3.

jdk.tls.client.enableStatusRequestExtension: Enables "OCSP stapling" per the status_request extension from RFC 6066.

jdk.tls.client.protocols: To enable specific TLS protocols, specify them in a comma-separated list within quotation marks and all other supported protocols will be disabled. For example, if the value of this property is "TLSv1,TLSv1.1", then the default protocol settings is only for TLSv1 and TLSv1.1 with nothing else available.

jdk.tls.client.SignatureSchemes: Contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS connections on the client side.

jdk.tls.ephemeralDHKeySize: Set this value to customize the size of ephemeral DH keys (in bits). This value defaults to 2048, with a minimum of 1024.

jdk.tls.maxCertificateChainLength: Specifies the maximum allowed length of the certificate chain in TLS handshaking.

jdk.tls.maxHandshakeMessageSize: Specifies the maximum allowed size, in bytes, for the handshake message in TLS handshaking.

jdk.tls.namedGroups: Defines the set of enabled curves and/or finite-field groups, in order of preference. The format is a comma-separated list of group names e.g. "secp256r1, secp384r1, secp521r1". See RFC 4492 \cite{RFC4492}, RFC 7919 [98], and/or the IANA's TLS Supported Groups Registry for the full list of group names.

jdk.tls.server.cipherSuites: Specify the default enabled cipher suites for TLS servers, in a comma-separated list.

jdk.tls.server.enableCAExtension: If set to "true", enables the certificate_authorities extension, an optional CertificateRequest extension introduced in TLS 1.3.

jdk.tls.server.protocols: To enable specific TLS server protocols, specify them in a comma-separated list within quotation marks and all other supported protocols will be disabled. For example, if the value of this property is "TLSv1,TLSv1.1", then the default protocol settings is only for TLSv1 and TLSv1.1 with nothing else available.

jdk.tls.server.SignatureSchemes: Contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS connections on the client side .

jdk.tls.trustNameService: Can be used to switch on reverse name lookup where it is needed for end point identification. Note: enabling reverse name lookup may be susceptible to MITM attacks when a name service is not trustworthy - see also section 3.5 before you use this.

jdk.tls.useExtendedMasterSecret: If set to "false", disables negotiation of the extended_master_secret extension.

jsse.enableFFDHE: If set to "false", disables the TLS FFDHE mechanisms defined in RFC 7919 [98].

jsse.enableSNIExtension: If set to "true" enable the Server Name Indication (SNI) extension.

sun.security.ssl.allowLegacyHelloMessages: If set to "true", allows connections to peers that do not implement secure renegotiation per RFC 5746.

BCJSSE Specific System Properties

The following properties are BCJSSE specific:

org.bouncycastle.jsse.client.dh.minimumPrimeBits: Integer property, default 2048, can be configured in the range 1024 to 16384.

org.bouncycastle.jsse.client.dh.unrestrictedGroups: boolean property, default false, if set (exact string) to "true" will accept any DH group meeting the size requirement (i.e. minimumPrimeBits above).

org.bouncycastle.jsse.client.enableSessionResumption: Controls whether automatic session resumption is enabled in TLS clients (when supported).

org.bouncycastle.jsse.client.enableTrustedCAKeysExtension: Controls whether the trusted_ca_-keys extension from RFC 6066 is enabled for TLS clients.

org.bouncycastle.jsse.config: an alternate method of passing a config string to the JSSE provider. Set to "default" to behave as normal, otherwise other configs, such as "fips:BCFIPS" can also be used.

org.bouncycastle.jsse.fips.allowRSAKeyExchange: boolean property, default true, if set to "true" will allow the use of cipher suites based on the use of RSA key exchange in fips mode. Note: the default value for this property will become "false" when the SP 800-131A transition away from RSA key exchange in TLS takes effect.

org.bouncycastle.jsse.ec.disableChar2: boolean property, default false, if set (exact string) to "true" will disable the use of any characteristic 2, or F2m, curves in TLS handshakes and key exchanges.

org.bouncycastle.jsse.keyManager.checkEKU: Controls whether BCJSSE key managers will check for an appropriate Extended Key Usage when selecting a set of credentials to present to the TLS peer. Note that a certificate with an invalid EKU may be rejected by the peer.

org.bouncycastle.jsse.server.enableSessionResumption: Controls whether automatic session resumption is enabled in TLS servers (when supported).

org.bouncycastle.jsse.server.enableTrustedCAKeysExtension: Controls whether the trusted_ca_-keys extension from RFC 6066 is enabled for TLS servers.

org.bouncycastle.jsse.trustManager.checkEKU: Controls whether BCJSSE trust managers will check for an appropriate Extended Key Usage extension as part of checking whether a given certificate chain is trusted.

BC JSSE Extensions

Following the TLS protocol is quite a challenge, for anyone, and as result the Bouncy Castle JSSE provider makes several extensions available to fill gaps in older version of the JSSE, as well as a couple of things in the current versions which have not quite made it into the public API.

Recovering the TLS unique ID

The BC JSSE returns SSLSocket objects which implement the `org.bouncycastle.jsse.BCSSLSocket` interface. The `BCSSLSocket` is defined as follows:

```java
/**
 * A BCJSSE-specific interface to expose extended functionality on
 * javax.net.ssl.SSLSocket implementations.
 */
public interface BCSSLSocket
{
    BCExtendedSSLSession getBCHandshakeSession();

    /**
     * Returns an accessor for extended SSL connection data. This method will
     * initiate the initial handshake if necessary and then block until the
     * handshake has been established. If an error occurs during the initial
     * handshake, this method returns null.
     *
     * @return A BCSSLConnection instance.
     */
    BCSSLConnection getConnection();

    /**
     * Returns a BCSSLParameters with properties reflecting the current
     * configuration.
     * @return the current BCSSLParameters parameters
     */
    BCSSLParameters getParameters();

    /**
     * Allows explicit setting of the 'host' String when the SocketFactory
     * methods that include it as an argument are not used.
     *
     * Must be called prior to attempting to connect the socket to have any
     * effect.
     *
     * @param host the server host name with which to connect, or
     *             null for the loopback address.
     */
    void setHost(String host);

    /**
```

```
     * Sets parameters according to the properties in a {@link BCSSLParameters}.
     *
     * Note that any properties set to null will be ignored, which will leave
     * the corresponding settings unchanged.
     *
     * @param parameters the {@link BCSSLParameters parameters} to set
     */
    void setParameters(BCSSLParameters parameters);
}
```

with `org.bouncycastle.jsse.BCSSLConnection` defined as:

```
/**
 * A BCJSSE-specific interface providing access to extended connection-specific
 * functionality.
 */
public interface BCSSLConnection
{
    /**
     * Returns the application protocol negotiated for this connection, or an empty
     * String if none was negotiated. See https://tools.ietf.org/html/rfc7301 for
     * details.
     *
     * @return The negotiated application protocol, or an empty {@code String}.
     */
    String getApplicationProtocol();

    /**
     * Request TLS Channel Bindings for this connection. See
     * https://tools.ietf.org/html/rfc5929 for details.
     *
     * @param channelBinding An IANA-registered "Channel-binding unique prefix"
     *                  valid for TLS e.g. "tls-unique" or "tls-server-end-point".
     * @return A copy of the channel binding data as a byte[], or null if the
     *     binding is unavailable for this connection.
     */
    byte[] getChannelBinding(String channelBinding);

    /**
     * Returns the SSL session in use by this connection
     * @return The BCExtendedSSLSession.
     */
    BCExtendedSSLSession getSession();
}
```

The channelBinding string represents an IANA registered “Channel-binding unique prefix” which is listed as valid for TLS, for example: “tls-unique”. You can make use of these BCJSSE specific interfaces to create a function to retrieve a channel binding like “tls-unique” by following the example that follows.

```
public byte[] getTlsUnique(Socket sock)
{
    BCSSLConnection bcon = ((BCSSLSocket)sock).getConnection();
    if (bcon != null)
    {
        return bcon.getChannelBinding("tls-unique");
    }
    return null;
}
```

Endpoint Identification

Endpoint identification was introduced in bctls 1.0.7 in a manner that tried to be as compatible with SunJSSE as possible. Unfortunately, when using HttpsURLConnection SunJSSE uses some magic (reflection and/or internal API) to tell the socket about the “original hostname” used for the connection, and we cannot use that same magic as it is internal to the JVM. To allow the endpoint validation to work properly you need to make use of one of two workarounds:

1. Set the system property “jdk.tls.trustNameService” to “true”. However, note this advice from the 8u51 release notes: *“If an application does need to perform reverse name lookup for raw IP addresses in SSL/TLS connections, and encounter endpoint identification compatibility issue, System property “jdk.tls.trustNameService” can be used to switch on reverse name lookup. Note that if the name service is not trustworthy, enabling reverse name lookup may be susceptible to MITM attacks.”*
2. The second (and recommended) alternative is to set a customized SSLSocketFactory on the HttpsURLConnection, then intercept the socket creation call and manually set the SNI host_name on the created socket. Please see the example code below. This is a general pattern that people typically use anyway when they want to e.g. adjust what ciphersuites are enabled.

```
// main code block
{
    SSLContext sslContext = ...;
    final URL serverURL = ...;
    HttpsURLConnection conn = (HttpsURLConnection)serverURL.openConnection();

    conn.setSSLSocketFactory(new CustomSSLSocketFactory(
                                    sslContext.getSocketFactory())
    {
        @Override
        protected Socket configureSocket(Socket s)
        {
            if (s instanceof SSLSocket)
            {
                SSLSocket ssl = (SSLSocket)s;

                SNIHostName sniHostName = getSNIHostName(serverURL);
                if (null != sniHostName)
                {
                    SSLParameters sslParameters = new SSLParameters();

                    sslParameters.setServerNames(
                        Collections.<SNIServerName>singletonList(sniHostName));
                    ssl.setSSLParameters(sslParameters);
                }
            }
            return s;
        }
    });
}

private SNIHostName getSNIHostName(URL url)
{
    String host = url.getHost();
    if (null != host && host.indexOf('.') > 0
            && !org.bouncycastle.util.IPAddress.isValid(host))
    {
        try
        {
            return new SNIHostName(host);
        }
        catch (RuntimeException e)
        {
                // ignore - return null
        }
    }
    return null;
}
```

Using the low level (D)TLS API

The BCJSSE provider implements TLS over a low-level (D)TLS API located in the org.bouncycastle.tls package. This API can be used directly instead of through the JSSE standard API. It is designed to allow great flexibility to the programmer to modify or extend all aspects of the implementation. This could mean simple configuration options like cipher suites, versions, elliptic curves, or more complex changes like adding support for a new TLS extension. For most users, using Bouncy Castle's TLS functionality via the BCJSSE provider will be the best choice, especially when working with third party applications that will usually be written to the JSSE standard API. However there may be use cases for which the JSSE does not expose the necessary functionality, where you need to use DTLS, or for some other reason the JSSE is not appropriate. In some cases a BC-specific extension may be available to cover the use case, but it is also possible to dispense with the JSSE altogether. Some familiarity with the TLS specification (as described in RFC 5246 and other related RFCs) is advisable. If you are planning to go down this path it may also be useful to refer to the source code for the TLS test suite for example usage, and to study how the BCJSSE provider itself uses the API.

TlsCrypto

The low-level (D)TLS API delegates all cryptographic operations to the TlsCrypto API, which is located in the org.bouncycastle.tls.crypto package. Use of this API is via an instance of the TlsCrypto interface; a single instance can be shared across many connections. e.g. the BCJSSE provider uses a single instance for each SSLContext. There are 2 existing implementations of TlsCrypto available in the regular TLS distribution:

- org.bouncycastle.tls.crypto.impl.bc.BcTlsCrypto
- org.bouncycastle.tls.crypto.impl.jcajce.JcaTlsCrypto

Another option is to write a completely new implementation, or perhaps more realistically to subclass one of these in order to tweak its behaviour. Just keep in mind that TlsCrypto objects should be shareable.

BcTlsCrypto

BcTlsCrypto relies on the lightweight BC crypto API for implementations of cryptographic primitives. The BcTlsCrypto is not available in the BCFIPS TLS jar.

JcaTlsCrypto

JcaTlsCrypto delegates operations to any installed cryptographic providers via the Java Cryptography Architecture (JCA). Although we expect that this will typically be used with the “BC”, or “BCFIPS”, provider, it is not a requirement. However, be aware that the set of cipher suites available to the TLS API is constrained by the capabilities of the available providers.

The Low-level TLS API

The low-level TLS API provides the foundation on which the BCJSSE provider is built, so anything that can be done with the BCJSSE provider can also be done in the low-level TLS API.

Outline of a simple TLS Client

Sample code for a simple TLS client:

```
// TlsCrypto to support client functionality
TlsCrypto crypto = new BcTlsCrypto(new SecureRandom());
…
InetAddress address = InetAddress.getByName("www.example.com");
int port = 443;
Socket s = new Socket(address, port);
TlsClient client = new DefaultTlsClient(crypto) {
 // MUST implement TlsClient.getAuthentication() here
};
TlsClientProtocol protocol = new TlsClientProtocol(
 s.getInputStream(), s.getOutputStream());
// Performs a TLS handshake
protocol.connect(client);
// Read/write to protocol.getInputStream(), protocol.getOutputStream()
…
protocol.close();
```

DefaultTlsClient provides a reasonable default set of cipher suites, like what a typical web browser might support. Other options are PSKTlsClient if you are using pre-shared keys, or SRPTlsClient if using the Secure Remote Password protocol for authentication. Note that before this code can be used, an implementation of TlsClient.getAuthentication() will have to be provided (see code comment above). That implementation should return an instance of the TlsAuthentication interface, and that instance MUST perform certificate validation in TlsAuthentication.notifyServerCertificate. It could be a simple check that the server certificate is exactly the expected one, or a full path validation as specified in RFC 5280, or something in between, but it is vital to understand that the security of the resulting TLS connection depends on this step. TlsAuthentication.getClientCredentials will also need to be implemented if you wish for your client to authenticate with the server.

Outline of a simple TLS Server

Writing a TLS server can be rather more work than a TLS client, depending on the range of functionality needed. The top-level code should be quite similar though:

```
// TlsCrypto to support server functionality
TlsCrypto crypto = new BcTlsCrypto(new SecureRandom());
…
int port = 443;
ServerSocket ss = new ServerSocket(port);
Socket s = ss.accept();
…
TlsServer server = new DefaultTlsServer(crypto) {
 // Override e.g. TlsServer.getRSASignerCredentials() or
 // similar here, depending on what credentials you wish to use.
};
TlsServerProtocol protocol = new TlsServerProtocol(
 s.getInputStream(), s.getOutputStream());
// Performs a TLS handshake
protocol.accept(server);
// Read/write to protocol.getInputStream(), protocol.getOutputStream()
…
protocol.close();
```

PKSTlsServer or SRPTlsServer are other options here, similar to the client case. Here also, before this can be useful for anything, the issue of what credentials the server will use has to be addressed (see code comments). To support client authentication requires overriding TlsServer.getCertificateRequest() and TlsServer.notifyClientCertificate, and the latter MUST perform certificate validation before the client is considered authenticated. The TlsClient or TlsServer implementation is where most customisation will likely take place, though it is possible to subclass TlsClientProtocol or TlsServerProtocol, if necessary.

DTLS

DTLS clients and servers uses different protocol classes and a different transport than the TLS ones, but the same TlsClient and/or TlsServer implementation will usually work fine with the DTLS protocol classes. There are some minor exceptions, e.g. DTLS does not support cipher suites based on stream ciphers like RC4. Example of client code differences from the TLS client example:

```
DatagramSocket socket = new DatagramSocket();
socket.connect(InetAddress.getByName("www.example.com"), 5556);
int mtu = 1500;
DatagramTransport transport = new UDPTransport(socket, mtu);
TlsClient client = ...;
DTLSClientProtocol protocol = new DTLSClientProtocol();
DTLSTransport dtls = protocol.connect(client, transport);
// Send/receive packets using dtls methods
…
dtls.close();
```

The server is again very similar:

```
DatagramTransport transport = ...;
TlsServer server = ...;
DTLSServerProtocol protocol = new DTLSServerProtocol();
DTLSTransport dtls = protocol.accept(server, transport);
// Send/receive packets using dtls methods
…
dtls.close();
```

Low-level TLS API Overrides

Making use of the low-level TLS classes usually requires extending the base classes and then overriding methods from the parent class to provide the functionality required. Some of the more useful overrides follow.

Note: in a lot of cases where you need to perhaps add something to an operation by overriding a method in one of the base classes it is usually a good idea to make sure you call the method in the base class via super as well.

Useful Overrides Based on TLSPeer

The TLSPeer class forms the basis of both TLS clients and servers in the low-level API.

Handshakes - notifyHandshakeBeginning(), notifyHandshakeComplete()

The `notifyHandshakeBeginning()` and `notifyHandshakeComplete()` provide the ability to capture the start and completion of a handshake. The `notifyHandshakeComplete()` method will only be called if a handshake has completed successfully. In the case of an alert you need to override the `notifyAlertRaised()` and `notifyAlertReceived()` methods.

Alerts - notifyAlertRaised(), notifyAlertReceived()

The `notifyAlertRaised()` and `notifyAlertReceived()` methods provide the ability to listen in on alerts that are raised either locally or remotely during the execution of the TLS protocol. The `notifyAlertRaised()` method will be called when an alert is raised locally. The `notifyAlertReceived()` will be called when an alert is received from the remote peer.

Useful Overrides Based on TLSClient

`TLSClient` is the client extension of `TLSPeer`. The usual base implementation class is `DefaultTLSClient` but there are also some additional overrides available.

Session Information - notifySessionID()

The `notifySessionID()` method is called to notify the client of the session ID sent in the ServerHello. The session ID is passed in as a `byte[]`.

Session Resumption - getSessionToResume(), notifySessionToResume()

Usually this method will return null, indicating a new session should be used. A non-null return can be used to indicate the session that the client wants to resume. Assume all is in place to make resumption possible, this session will be passed to the `notifySessionToResume()` method indicating success. If it was not possible to resume the session `notifySessionToResume()` will be called with `null` and the new session ID will be passed to the `notifySessionID()` method.

Useful Overrides Based on TLSServer

`TLSServer` is the server extension of `TLSPeer`. The usual base implementation class is `DefaultTLSServer` but there are also some additional overrides available.

Session Information - notifySession()

The `notifySession()` method is called on the server object to pass in all the session information as well as the ability to prevent the session from being resumed. The required data is passed in using an object implementing the `TlsSession` interface. The `TlsSession` interface has methods on it returning the session parameters, the session ID, and whether the session is resumable. An `invalidate()` method is also provided which can be used to prevent the session from being resumed.

Specialist Protocols: PSK and SRP

There is some support for both PSK and SRP in the low-level APIs. At the moment there is no support in the JSSE level - this may change in the future, but at the moment the amount of work required seems to be more than the amount of work people face just using the low-level API in the first place.

TLS-PSK - Transport Layer Security Preshared-Key

Use of a preshared-key in TLS is covered in RFC 4279 [60]. The low-level TLS API provides support for implementing TLS-PSK with the `PSKTlsClient` class. You can implement a TLS-PSK client by extending the `PSKTlsClient` class and at a minimum overriding the `getAuthentication()` method. The `PSKTlsClient` class takes the TLS-PSK identity information in its constructor and objects implementing the `TlsPSKIdentity` can also be used to pass in the TLS-PSK identity information.

TLS-PSK server support is provided by the `PSKTlsServer` class where the implementer needs to override `getRSASignerCredentials()` and provide an implementation of `TlsPSKIdentityManager` when the server object is constructed.

TLS-SRP - Transport Layer Security Secure Remote Password

The use of the Secure Remote Password (SRP) Protocol for TLS Authentication, or TLS-SRP, is documented in RFC 5054 [65]. Bouncy Castle provides support for TLS-SRP with the `SRPTlsClient` class. To implement an SRP-TLS client you need to extend the SRPTlsClient class and, a minimum override the `getAuthentication()` method to supply support for the protocol. As with TLS-PSK, the `SRPTlsClient` takes the identity information in its constructor, with `TlsSRPIdentity` providing the basic interface that any objects carrying identity need to conform to.

TLS-SRP server support is provided by the `SRPTlsServer` class. In this case the important methods are `getDSASignerCredentials()` and `getRSASignerCredentials()` - one of which needs to be overridden in order for the server to provide credentials. The server class also requires an implementation of `TlsSRPIdentityManager` when the server object is constructed. The low-level API also provides the `SimulatedTlsSRPIdentityManager` to allow a server to simulate the existence of “unknown” identities by providing an implementation the algorithm from RFC 5054, Section 2.5.1.3. Using this class will allow a server to hide the fact a user name does not have a verifier.

Summary

The Bouncy Castle API provides both a JSSE provider and a low-level API for in order to support TLS, DTLS, and some variations on the protocols, such as PSK and SRP. For specialist variations like PSK and SRP you need to make use of the low-level API.

For debugging purposes, the TLS APIs provide support for Java Logging. The BCJSSE provider also can make use of many of same system properties the JSSE provider does, just keep in mind that the BCJSSE provider has defaults for many properties based on the latest JVM, whereas the regular JSSE provider will regard undefined properties as simply unset without defaults.

Chapter 15: The Future

This chapter attempts to provide a window on current developments in cryptography as well as how they are starting to affect the direction of the Bouncy Castle APIs. The greatest activity in the area at the moment are around post-quantum algorithms - these cover application areas as diverse as key exchange, signatures, and key transport. One of the reasons this development is of interest is that many of these algorithms are not “business as usual” - we see stateful signature algorithms and key exchange mechanisms where only one party has a private key. These differences will probably see the introduction of new protocols and also change the way we, as developers, look at solving some of the problems we deal with to support our applications and services.

Installing the Bouncy Castle Post-Quantum Provider

The Bouncy Castle Post-Quantum Provider (BCPQC) is included in the general BC provider jars (the ones starting with bcprov) starting at the Java 1.5 editions. It is not installed automatically if the BC provider is and needs to be installed either statically or at runtime as well.

At the moment there is no equivalent to use along side the BCFIPS provider, although there are some tools for BCFIPS which make use of post-quantum algorithms.

Static Installation

How you install a provider statically depends on whether you are dealing with a JVM that is pre-Java 1.9 or not.

For Java 1.5 to Java 1.8 you can add the provider by inserting its class name into the list in the `java.security` file in `$JAVA_HOME/jre/lib/security`. The provider list is a succession of entries of the form “security.provider.n” where n is the precedence number for the provider, with 1 giving the provider the highest priority. To add the Bouncy Castle Post-Quantum provider in this case you need to add a line of the form:

```
security.provider.N=org.bouncycastle.pqc.jcajce.provider.BouncyCastlePQCProvider
```

Where N represents the precedence you want the BCPQC provider to have. Make sure you adjust the values for the other providers if you do not add the BCPQC provider to the end of the list. After adding that entry, ensure the provider is on the class path, preferably in the `$JAVA_HOME/jre/lib/ext` directory so that it will always be loaded by a class loader the JCA trusts.

For Java 1.9 onwards the list is in the file `$JAVA_HOME/conf/security`. At the moment the BC providers do not support the `META-INF/service` feature of jar files, so the provider needs to be added in the same way as before, by using the full class name, to the list of providers.

Java 1.9 no longer supports `$JAVA_HOME/jre/lib/ext` either. In the case of Java 1.9 you just need to make sure BCPQC is included on the class path.

Runtime Installation

There are two ways of making use of a provider at runtime.

The first method is to use `java.security.Security` - for the BCPQC provider this looks like:

```
java.security.Security.addProvider(
            new org.bouncycastle.pqc.jcajce.provider.BouncyCastlePQCProvider());
```

After the above statement executes the situation in the JVM is almost equivalent to what will happen if the provider is statically installed. Services such as signatures can be accessed by using `getInstance()` methods using the provider's name.

With the second method, the provider object is passed into the service being requested instead, for example, to create an XMSS signature using the BCPQC provider, you can write:

```
Signature sig = Signature.getInstance("XMSS", new BouncyCastlePQCProvider());
```

Stateful Signature Algorithms

Most signature algorithms we deal with are stateless. That is to say, once a signing operation is carried out, everything is as it was before the signing operation, the same private key can be used again with the same results.

Stateful signature algorithms differ in this regard - the private key is modified each time a signature is generated. The reason for this is that the stateful schemes are based on one-time signature schemes, and the many-time signature scheme that a stateful signature scheme provides is actually working through a set of one-time private keys creating signatures, all of which can be verified by the same public key.

In some ways stateful signature schemes are actually an advantage, as the private key has a limited life, the use of a stateful scheme forces people to rotate their keys and deal with the 'hard' issues around cryptography and security properly at the start of development, rather than the usual process of "deploy first, panic later". The algorithms we will look at here also include the signature index in the signature. This makes it easy to tell how many signatures you have handed out, even if you do not have ready access to the private key.

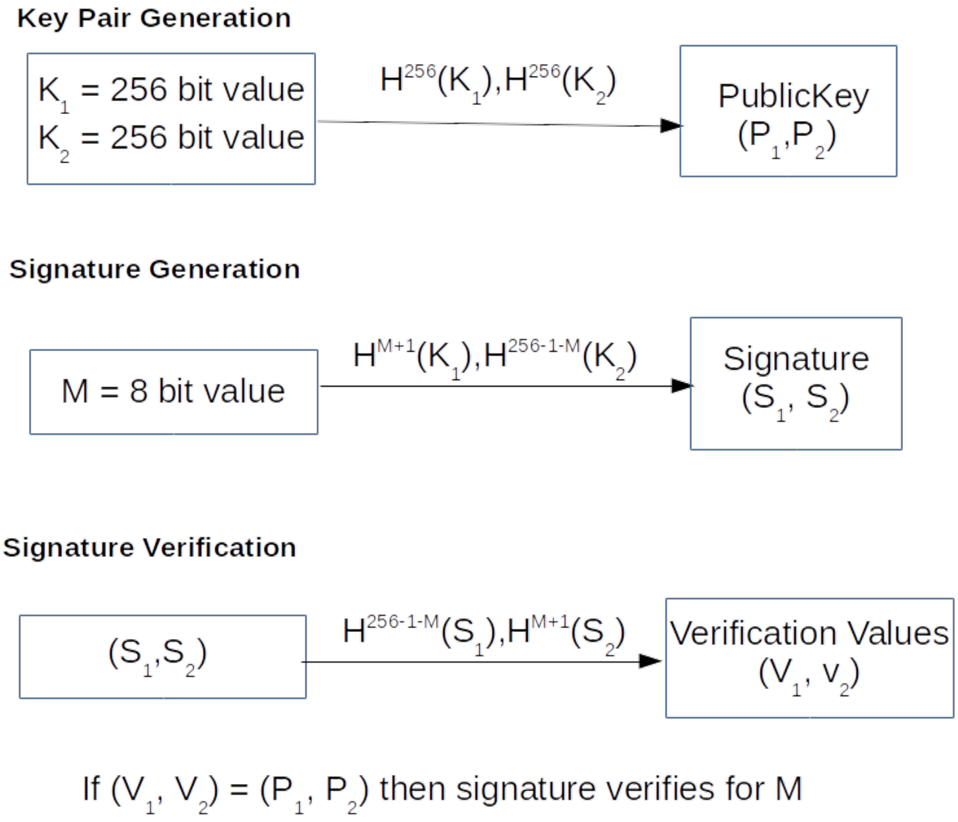
That said, stateful signature schemes present some new problems that we have not previously encountered. It is important to take these new problems into account, particularly where you need to persist the private key.

Origins

Stateful signature algorithms trace there origins back to early work on hash-based signature algorithms which originated from the work of Leslie Lamport [133] and Robert Winternitz [134] in 1979. It is worth having a quick look at one of these schemes in some detail as it helps explain both how these schemes actually work and why there is such a need to make sure the state in the key is not reused.

Basic Winternitz One-time Signatures

The Winternitz scheme is of interest as it is both representative of hash-based signature generators and variants of it (W-OTS+) are used in algorithms like XMSS, as well as SLH-DSA, now one of the NIST Post-Quantum Standards. The scheme is also relatively simple to describe and implement. Have a look at the following diagram and you will see how it would work for signing a single byte using SHA-256 as the message digest.



Processes around signing a single byte using W-OTS

You can see in the diagram that we get a 64 byte signature as we are also hashing a checksum based on the complement of the byte being signed. While the first 32 byte value does provide us with a signature on the value that we can verify, it is also trivial to hash that value again and, in the absence of additional information, we would accept the first 32 bytes as the signature of $(M + 1)$. In order to avoid this we rely on the compliment of M, as in $(w-1-M)$ for the checksum. Doing this means that any prospective forger trying to present $(M+1)$ as valid is then faced with the problem of computing $SHA-256(w-1-(M+1))$ which is the pre-image of $SHA-256(w-1-M)$ - extremely difficult, if not impossible if the digest function in use is any good.

Unfortunately the need to protect pre-images means that we can also use the W-OTS private key once - if we do not follow this rule and someone can fool us into signing the byte ‘0x00’ and the byte ‘0xFF’ they will be able to forge signatures for any value.

Implementing the scheme in a more general sense is reasonably straightforward. The Winternitz scheme relies on a parameter w to determine how many bits it can process with each part of the private key - w represents 2^bits. For simplicity we will just use the following setup parameters:

```
private static final String DIGEST = "SHA-256";
private static final int DIGEST_SIZE = 32;

private static final int W = 256;
// CHK_SIZE = floor(log2(DIGEST_SIZE * (W - 1))/log2(W)) + 1
private static final int CHK_SIZE = 2;
private static final int KEY_ELEMENTS = W / 8 + CHK_SIZE;
```

I say simplicity as setting w to 256 means we can just do everything based on bytes. You could increase w to a higher power of 2 though, for example 65536 would allow the message to be processed in 2 byte chunks and halve the signature size and the size of the public and private keys.

The checksum size has been set to 2 bytes as we can describe the calculation we want as follows:

```
/**
 * Calculate a compliment checksum to prevent forgeries based on
 * upping the values of msg[i].
 *
 * @param msg the message to calculate the checksum on.
 * @return the checksum as a byte[].
 */
private static byte[] calcCheckSum(byte[] msg)
{
    // compute msg checksum
    int checkSum = 0;
    for (int i = 0; i < msg.length; i++ )
    {
        checkSum += W - 1 - (msg[i] & 0xff);
    }
    // note: this is dependent on CHK_SIZE
    return Pack.shortToBigEndian((short)checkSum);
}
```

We can return this as a 2 byte checksum as $32 * (w - 1)$, the maximum value of that the checksum can be will fit in 2 bytes. Using the full compliment of the message would also work but it will double the signature size - the smaller calculation also works as it creates the same problem for a forger. Any forged signature can only be calculated by increasing one of the bytes in the hash of the message we are signing, it is not possible for a forger to use a smaller byte value at a particular index as they cannot recover the pre-image for the hash. Increasing a byte value will have the affect of decreasing at least one byte in the checksum, so the prospective forger is in trouble again.

Calculating the one-time key pair is then just a matter of creating an array of secrets representing the private key and then doing the necessary number of hashes to create the public key.

```java
/**
 * Generate a W-OTS key pair.
 *
 * @param random randomness source for key generation.
 * @return an array in the form of {publicKey, privateKey}
 */
private static byte[][] computeWotsKeyPair(SecureRandom random)
    throws GeneralSecurityException
{
    MessageDigest digest = MessageDigest.getInstance(DIGEST);

    // create a 32 * 256 bit private key
    byte[] privateKey = new byte[KEY_ELEMENTS * DIGEST_SIZE];

    random.nextBytes(privateKey);

    byte[] publicKey = new byte[privateKey.length];
    System.arraycopy(privateKey, 0, publicKey, 0, publicKey.length);
    for (int i = 0; i != KEY_ELEMENTS; i++)
    {
        for (int j = 0; j != W; j++)
        {
            digest.update(publicKey, i * DIGEST_SIZE, DIGEST_SIZE);
            digest.digest(publicKey, i * DIGEST_SIZE, DIGEST_SIZE);
        }
    }

    return new byte[][]{publicKey, privateKey};
}
```

You can see looking at this that increasing w to 65536 would also mean while there will be less public key calculations, the ones that stay will take 256 times longer to create the public key. Quite a jump!

To calculate the signature: we first hash the message, append the checksum to the result, and then perform the hash calculations for each byte. The following method shows how this is done.

```java
/**
 * Calculate a signature based on a digest of the message and the private key.
 *
 * @param privateKey the W-OTS private key to use.
 * @param message the message to be signed.
 * @return the calculated signature.
 */
private static byte[] computeWotsSignature(byte[] privateKey, byte[] message)
    throws GeneralSecurityException
{
    MessageDigest digest = MessageDigest.getInstance(DIGEST);

    byte[] msgDigest = digest.digest(message);
    byte[] checkMsg = Arrays.concatenate(msgDigest, calcCheckSum(msgDigest));

    byte[] signature = new byte[privateKey.length];
    System.arraycopy(privateKey, 0, signature, 0, signature.length);
    for (int i = 0; i != KEY_ELEMENTS; i++)
    {
        for (int j = 0; j <= (checkMsg[i] & 0xff); j++)
        {
            digest.update(signature, i * DIGEST_SIZE, DIGEST_SIZE);
            digest.digest(signature, i * DIGEST_SIZE, DIGEST_SIZE);
        }
    }

    return signature;
}
```

You can see from the code that if w was increased the same issue would apply to signature generation that applies to key generation.

Verifying the signature also requires hashing the message and then iterating the hash values in the signature forward according the message hash. Assuming everything is correct doing this should produce the same value as the public key.

```
/**
 * Verify a passed in W-OTS signature.
 *
 * @param publicKey the W-OTS public key to verify the signature with.
 * @param signature the signature to be verified.
 * @param message the message that the signature should verify.
 * @return true if message verifies, false otherwise.
 */
private static boolean isVerifiedWotsSignature(
    byte[] publicKey, byte[] signature, byte[] message)
    throws GeneralSecurityException
{
    MessageDigest digest = MessageDigest.getInstance(DIGEST);

    byte[] msgDigest = digest.digest(message);

    byte[] calcPubKey = new byte[publicKey.length];
    byte[] checkMsg = Arrays.concatenate(msgDigest, calcCheckSum(msgDigest));

    System.arraycopy(signature, 0, calcPubKey, 0, calcPubKey.length);
    for (int i = 0; i != KEY_ELEMENTS; i++)
    {
        for (int j = (checkMsg[i] & 0xff); j != W - 1; j++)
        {
            digest.update(calcPubKey, i * DIGEST_SIZE, DIGEST_SIZE);
            digest.digest(calcPubKey, i * DIGEST_SIZE, DIGEST_SIZE);
        }
    }

    // calcPubKey should now equal publicKey
    return Arrays.constantTimeAreEqual(calcPubKey, publicKey);
}
```

Again you can note that increasing w will also result in an increase in the verification time.

You can put all these methods together using the following code fragment:

```
      byte[] msg = Strings.toByteArray("Hello, world!");
      byte[][] wotsKp = computeWotsKeyPair(new SecureRandom());

      byte[] signature = computeWotsSignature(wotsKp[1], msg);

      System.out.println("verifies: "
          + isVerifiedWotsSignature(wotsKp[0], signature, msg));
```

Running the example fragment should produce the following output:

```
verifies : true
```

hopefully showing that key generation, signing, and verification all worked correctly and were in sync. If you are inclined to, it is worth trying to increase w to see what happens, just be careful with the size of the checksum as it will increase. The smaller signatures are nice, but you will notice a difference in the running time.

Moving to Multiple-Signatures - XMSS and LMS

We can make use of algorithms like W-OTS as the basis of algorithms for generating multiple signatures by adding an index, or counter, into the mix and using the index and the secret value in something like a HMAC or other pseudo-random function. This still allows us to take advantage of the difficulties in recovering a pre-image, to keep the algorithm "quantum-hard", but you can also see that while the introduction of the index does allow us to re-use the secret values, re-using the index will put us in the same position we were in with W-OTS as reusing the state will make it possible to start recovering pre-images that could be used for forging signatures.

These issues all stem from reusing a private key, and carelessness aside, the simplest way for an outsider to cause mischief is to simply steal a copy of your private key store and then come back later and replace it with the earlier one at some later date, resulting in private key reuse as the key has been reset to a previous state. An attack like this does not require knowing your key password, and it will not necessarily leave an obvious trail in the beginning, only making itself obvious when it is too late. NIST regard this issue so seriously that the current draft for SP 800-208 [17] (currently in draft) mandates the use of HSMs for key generation and signature generation.

Considering this, assuming you cannot take advantage of a HSM, it's worth taking some extra steps to protect the private keys, and also to check that signature indexes are not being reused. With the private key, you might want to consider changing the password on each persist and/or storing some information about its state elsewhere. On the signature front, for the algorithms we look at here, the index used in the signature construction is available in the signature, and in the case of the Bouncy Castle APIs the next index to be used is also available from the private key, so for example, in the case of certificate production you could use the index number as the source for the serial number for your certificate and rely on any constraints you have in place to make sure you are not handing out a certificate with the same serial number twice to raise the alarm.

We will look at two stateful signature algorithms here, LMS and XMSS. These algorithms are of interest as both are based on well understood methods, both have been standardised by the IETF, and both are also in the process of being standardised by NIST in SP 800-208.

LMS

Leighton-Micali Signatures (LMS) and the Hierarchical Signature System (HSS) which is built on LMS are defined in RFC 8554 [103]. Additional standards work has been approved on these signatures as well. RFC 8708 [104] now provides guidance for CMS and X.509 public key encoding and RFC 8778 [105] for use in messages built around the Concise Binary Object Representation (CBOR).

The scheme is based on the use of trees of hashes and was originally patented under U.S. Patent 5,432,852 which has now expired.

Workflow with LMS

After a couple of attempts at making stateful signature algorithms more useful and safer to deal with, including an attempt at providing a special interface for stateful signatures `StateAwareSignature` we ended up with thread safe private keys which auto-rotate indexes on each signature operation and what we refer to as "key-shards".

Key-shards represent short time private keys that occupy a part of the key space for the LMS private key. For LMS, and HSS, a key shard is created by using the `extractKeyShard()` method on the interface `LMSPrivateKey`.

Here is the definition of the `LMSPrivateKey` interface:

```java
/**
 * Base interface for an LMS private key
 */
public interface LMSPrivateKey
    extends LMSKey, PrivateKey
{
    /**
     * Return the index of the next signature.
     *
     * @return the index number for the next signature.
     */
    long getIndex();

    /**
     * Return the number of usages left for the private key.
     *
     * @return the number of times the key can be used before it is exhausted.
     */
    long getUsagesRemaining();

    /**
     * Return a key representing a shard of the key space that can be used
     * usageCount times.
     * <p>
     * Note: this will use the range [index...index + usageCount) for the
     * current key.
     * </p>
     * @param usageCount the number of usages the key should have.
     * @return a key based on the current key that can be used usageCount times.
     */
    LMSPrivateKey extractKeyShard(int usageCount);
}
```

Note that the interface also allows you to retrieve the number of usages remaining for the private key as well as the next index that will be used in certificate generation. We will see a concrete example of the index usage further on.

The following example code provides a function for generating an LMS key pair.

```
/**
 * Generate a LMS key pair with a tree height of 5, based around SHA-256.
 *
 * @return an LMS KeyPair
 */
public static KeyPair generateLMSKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("LMS", "BCPQC");

    kpg.initialize(new LMSKeyGenParameterSpec(
                            LMSigParameters.lms_sha256_n32_h5,
                            LMOtsParameters.sha256_n32_w1));

    return kpg.generateKeyPair();
}
```

Note that in the case of the LMS key pair generation, the hash function definition used is a property of the tree.

In the following code we see how to sign data and verify the resulting signature.

```
/**
 * Basic example of LMS with SHA-512 as the tree digest.
 */
public class LMSExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        Security.addProvider(new BouncyCastlePQCProvider());

        byte[] msg = Strings.toByteArray("hello, world!");

        KeyPair kp = generateLMSKeyPair();

        Signature lmsSig = Signature.getInstance("LMS", "BCPQC");

        lmsSig.initSign(kp.getPrivate());

        lmsSig.update(msg, 0, msg.length);

        byte[] s = lmsSig.sign();

        lmsSig.initVerify(kp.getPublic());

        lmsSig.update(msg, 0, msg.length);

        System.err.println("LMS verified: " + lmsSig.verify(s));
    }
}
```

Certificate Generation with LMS

The following code produces an X.509 certificate using LMS as outlined in RFC 8708 [104]. In this case we have made a change to the way we previously generated certificates as the LMS private key can provide a unique supply of serial numbers. You can see this where 'privKey.getIndex()' is used as the source of the certificate serial number.

Here is the sample code:

```
/**
 * Example of building an LMS based certificate - self-signed in this case.
 */
public class LMSCertExample
{
    /**
     * Create an X.509 certificate signed by an LMS key, using
     * the signature index as the serial number.
     *
     * @param privKey the LMS private key to use.
     * @param issuer the issuer name for the certificate.
     * @param pubKey the public key to be held in the certificate.
     * @param subject the name to associate with the public key.
     * @return an X.509 certificate holder.
     */
    public static X509CertificateHolder createLMSCertificate(
        LMSPrivateKey privKey, X500Name issuer,
        PublicKey pubKey, X500Name subject)
        throws OperatorCreationException
    {
        X509v1CertificateBuilder certBldr = new JcaX509v1CertificateBuilder(
                                                issuer,
                                                BigInteger.valueOf(privKey.getIndex()),
                                                calculateDate(0),
                                                calculateDate(24 * 31),
                                                subject,
                                                pubKey);

        ContentSigner signer = new JcaContentSignerBuilder("LMS")
                                        .setProvider("BCPQC").build(privKey);

        return certBldr.build(signer);
    }

    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastlePQCProvider());

        X500Name name = new X500Name("CN=LMS Demo Root Certificate");

        KeyPair kp = generateLMSKeyPair();
        LMSPrivateKey privKey = (LMSPrivateKey)kp.getPrivate();
44
45         // we throw away our signature with index 0 - RFC 5280 requires
46         // certificate serial numbers > 0.
47         privKey.extractKeyShard(1);
48
49         X509CertificateHolder certHldr = createLMSCertificate(
50                                             privKey, name, kp.getPublic(), name);
51
52         System.out.println("key index: " + privKey.getIndex());
53         System.out.println("serial no: " + certHldr.getSerialNumber());
54         System.out.println("sig index: "
55                        + getLMSSignatureIndex(certHldr.getSignature()));
56         System.out.println("verifies : " + certHldr.isSignatureValid(
57             new JcaContentVerifierProviderBuilder()
58                 .setProvider("BCPQC").build(kp.getPublic())));
59     }
60 }
```

You will note that at the 52/53 the index number of the signature is printed out. The code for doing this is quite simple as the index number is the first 4 bytes of the signature. Here is the function definition:

```
/**
 * Return the index used in creating an LMS signature.
 *
 * @return the index used for the encoded signature.
 */
public static long getLMSSignatureIndex(byte[] lmsSig)
{
    return Pack.bigEndianToInt(lmsSig, 0) & 0xFFFFFFFFL;
}
```

The presence of the index number in the signature also opens up other paths to indicate if something has gone awry and an index is somehow getting reused. In the case of the example we are looking at tracking the index number publicly as it is also used in the serial number of the certificate. It never hurts to keep a watchful eye on things, even if you are using a HSM.

Of course, depending on what certificate profile you are following, there might be a catch. Some profiles require the certificate serial number to be generated using a random bit generator. If you are dealing with something like this assign a random seed to a CA and then generate the serial number for the certificate using either a HMAC, or an XOF, using both the seed and the key index as input. While this creates a primitive DRBG so the output will be unpredictable and irreversible it will mean you should never get the some serial number twice - unless you've given the function the same inputs, which would indicate a second use of a serial number.

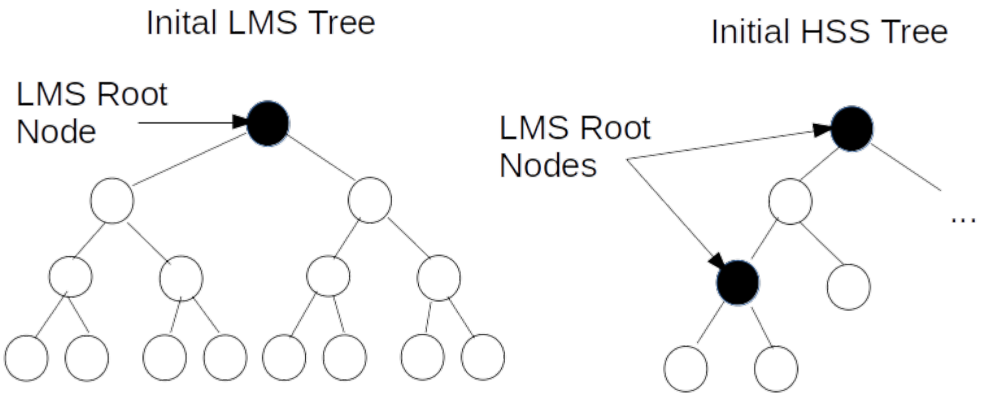
If you run the example and all goes well, you should see the following output:

```
key index: 2
serial no: 1
sig index: 1
verifies : true
```

showing that the signature index stored in the signature is the same one used as the serial number for the certificate.

HSS

The Hierarchical Signature System (HSS) provides a way to increase the number of signatures that a single public key can verify at the expense of signature size and increasing the size of the private key. Essentially a hierarchy of trees is created, but the leaf node trees are used to sign the actual data and the parent trees are used to sign the public keys of the leaf node trees.



The multi-tree approach of HSS means the lower level root nodes of the tree can be calculated as needed.

Superficially, from an API point of view this means that the only difference in usage is the generation of a key pair as the additional sub-trees need to be specified. The following example code provides a function for generating an HSS key pair which uses 2 levels of trees, both with a depth of 5, giving a total depth of 10:

```
/**
 * Generate a HSS key pair with a tree height of 10, based around SHA-256.
 *
 * @return an HSS KeyPair
 */
public static KeyPair generateHSSKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("LMS", "BCPQC");
```

```

        kpg.initialize(new LMSHSSKeyGenParameterSpec(
            new LMSKeyGenParameterSpec(LMSigParameters.lms_sha256_n32_h5,
                                            LMOtsParameters.sha256_n32_w1),
            new LMSKeyGenParameterSpec(LMSigParameters.lms_sha256_n32_h5,
                                            LMOtsParameters.sha256_n32_w1)));

        return kpg.generateKeyPair();
    }
```

The interesting thing about this approach is that if you try this and look at the size of the public key encoding, you will find that the encoding is the same size as the single level LMS key. The reason for was hinted at earlier and lies in the fact that the 2 level HSS key signature is really the encoding of the public key of the sub-tree signed using the HSS public key and then the encoding of the signature generated by the single use private key of the sub-tree. Consequently all the sub-tree information required to verify the HSS signature is contained in the full encoding with the only missing piece of information being the public key of the base tree making up the HSS hierarchy.

At any rate, what this all means is that you will find that you can apply the HSS keys in the same way as you applied the basic LMS key before. There is no need to even change the name of the signature algorithm.

XMSS

XMSS is described in RFC 8391 “XMSS: eXtended Merkle Signature Scheme” [102] which was finalised in May 2018.

The XMSS scheme is hash based like LMS, but uses the WOTS+ primitive (an extension of Winternitz One-Time Signature). The WOTS+ primitive is an operation which calculates a signature by taking the input message and a single use private key and produces a chain of hashes forming the signature. In this case the private key is a set of random byte strings and the public key is also represented as a chain of hashes calculated against the random byte strings making up the private key. Unlike previous schemes we have seen where verification usually uses the public key and the data to do a computation that matches the signature, a WOTS+ signature is calculated so that verification uses the signature and the data to calculate another chain of hashes which should correspond to the hash chain making up the public key, if the signature and the data really are related to one another and the original private key.

XMSS comes in two variants; the first is basic XMSS, which operates off a single tree of hashes, and the second is XMSS^MT which operates off several layers of XMSS reduced height trees. An equivalent XMSS^MT configuration to an XMSS configuration will allow for quicker key generation (exponentially) and reduced signature generation time (linearly). On the other hand, the signatures generated for the XMSS^MT key pair will be scaled up linearly in size. Which scheme you choose is a question of what trade offs you wish to make.

In terms of limitations with XMSS the number of uses a private key can have is limited to 2^h, where h is the height of the tree. XMSS^MT is configured using both the height h and the number of layers d, with each layer been made up of a number of XMSS trees of height h/d (d must divide h without remainder). At the top layer there is a single XMSS tree and the bottom layer has $2^{(h-(h/d))}$ trees. The trees above the bottom layer are used to sign the next layer down and the trees on the bottom layer is used to sign actual messages, giving 2^h possible signatures as well.

The following example code provides a function for generating an XMSS key pair, where h is set to 10. The hash function used to construct the public key and any signatures generated will be SHA-512.

```
  /**
   * Generate a XMSS key pair with a tree height of 10, based around SHA-512.
   *
   * @return an XMSS KeyPair
   */
  public static KeyPair generateXMSSKeyPair()
      throws GeneralSecurityException
  {
      KeyPairGenerator kpg = KeyPairGenerator.getInstance("XMSS", "BCPQC");

      kpg.initialize(XMSSParameterSpec.SHA2_10_512);

      return kpg.generateKeyPair();
  }
```

Note that in the case of the XMSS key pair generation, the hash function definition used is a property of the tree.

In the following code we see how to sign actual data and verify the resulting signature as well.

```
/**
 * Basic example of XMSS with SHA-512 as the tree digest.
 */
public class XMSSExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        byte[] msg = Strings.toByteArray("hello, world!");

        KeyPair kp = generateXMSSKeyPair();

        Signature xmssSig = Signature.getInstance("XMSS", "BCPQC");

```

```java
        xmssSig.initSign(kp.getPrivate());

        xmssSig.update(msg, 0, msg.length);

        byte[] s = xmssSig.sign();

        xmssSig.initVerify(kp.getPublic());

        xmssSig.update(msg, 0, msg.length);

        System.err.println("XMSS verified: " + xmssSig.verify(s));
    }
}
```

You can also see the algorithm is described as “XMSS”, in this case the message will be hashed directly into the signature, pre-hashing is specified in the usual JCA style, such as “SHA512withXMSS”, where the digest name used for calculating the message hash is specified at the start of the signature algorithm. Bouncy Castle currently limits the pre-hash digests to the same digest the hash tree is constructed with. While you could specify a different digest to that used for the key generation for calculating the data digest that forms the message signed by the XMSS algorithm, it is better to use the same one. Apart from potential differences in security level, odd things could become possible if you allow for a different algorithm. If you also parameterise the digest, someone may decide that a different parameterisation might suit their purposes - SHA-256 is not the only thing that can be used to produce a 256 bit digest.

Assuming everything is working correctly you should get the following output:

```
XMSS verified: true
```

If you want to try tweaking the tree height in the key pair generation, you will find that it does make a difference to the amount of time the example takes to run. Having a sensible lifetime for a private key with XMSS, or XMSS^MT, will also help with system performance.

Workflow with XMSS

Like LMS, XMSS private keys support the `extractKeyShard()` method and also have a usages remaining count associated with private key which will eventually go down to zero. If you try this next example, you will see what happens when the usage count on a key drops to zero.

```
/**
 * Example of use and effect of a private key running out of usages with XMSS.
 */
public class XMSSExceptionExample
{
    public static void main(String[] args)
        throws GeneralSecurityException
    {
        byte[] msg = Strings.toByteArray("hello, world!");

        KeyPair kp = generateXMSSKeyPair();
        XMSSPrivateKey privKey = (XMSSPrivateKey)kp.getPrivate();

        Signature xmssSig = Signature.getInstance("XMSS", "BCPQC");

        // this uses extractKeyShard() to get a single use key.
        XMSSPrivateKey sigKey = privKey.extractKeyShard(1);
        //Uncomment this line back in to make the example complete
//        sigKey = privKey.extractKeyShard(2);

        xmssSig.initSign(sigKey);

        xmssSig.update(msg, 0, msg.length);

        byte[] s1 = xmssSig.sign();

        // a 0 value here tells us the key can no longer be used.
        System.out.println("xmssSig usages remaining: "
                                     + sigKey.getUsagesRemaining());

        xmssSig.update(msg, 0, msg.length);

        //this line will only work if extractKeyShard() was called with more than 1.
        byte[] s2 = xmssSig.sign();

    }
}
```

Running the example as currently implemented will produce an exception with the following message:

```
Exception in thread "main" java.security.SignatureException: no usages of private ke\
y remaining
```

If you remove the comment characters on line 19, which reinitializes the key so that it has 2 usages possible rather than the existing 1, you should find the example runs to conclusion without throwing an exception and just produces the usage count message.

Certificate Generation with XMSS

The following code produces an X.509 certificate using XMSS as outlined in [44]. As with the LMS example this example makes use of `privKey.getIndex()` as the source of the certificate serial number.

```java
/**
 * Example of building an XMSS based certificate - self-signed in this case.
 */
public class XMSSCertExample
{
    /**
     * Create an X.509 certificate signed by an XMSS key, using
     * the signature index as the serial number.
     *
     * @param privKey the XMSS private key to use.
     * @param issuer the issuer name for the certificate.
     * @param pubKey the public key to be held in the certificate.
     * @param subject the name to associate with the public key.
     * @return an X.509 certificate holder.
     */
    public static X509CertificateHolder createXMSSCertificate(
        XMSSPrivateKey privKey, X500Name issuer,
        PublicKey pubKey, X500Name subject)
        throws OperatorCreationException
    {
        X509v1CertificateBuilder certBldr = new JcaX509v1CertificateBuilder(
                                                issuer,
                                                BigInteger.valueOf(privKey.getIndex()),
                                                calculateDate(0),
                                                calculateDate(24 * 31),
                                                subject,
                                                pubKey);

        ContentSigner signer = new JcaContentSignerBuilder("XMSS")
                                .setProvider("BCPQC").build(privKey);

        return certBldr.build(signer);
    }

    public static void main(String[] args)
        throws Exception
    {
        X500Name name = new X500Name("CN=XMSS Demo Root Certificate");

        KeyPair kp = generateXMSSKeyPair();
        XMSSPrivateKey privKey = (XMSSPrivateKey)kp.getPrivate();

        // we throw away our signature with index 0 - RFC 5280 requires
        // certificate serial numbers > 0.
        privKey.extractKeyShard(1);

        X509CertificateHolder certHldr = createXMSSCertificate(
                                            privKey, name, kp.getPublic(), name);

        System.out.println("key index: " + privKey.getIndex());
        System.out.println("serial no: " + certHldr.getSerialNumber());
        System.out.println("sig index: "
                       + getXMSSSignatureIndex(certHldr.getSignature()));
        System.out.println("verifies : " + certHldr.isSignatureValid(
            new JcaContentVerifierProviderBuilder()
                .setProvider("BCPQC").build(kp.getPublic())));
    }
}
```

Like LMS, the index number is the first 4 bytes of the signature. Here is the definition for the `getXMSSSignatureIndex()` method:

```
/**
 * Return the index used in creating an XMSS signature.
 *
 * @return the index used for the encoded signature.
 */
public static long getXMSSSignatureIndex(byte[] xmssSig)
{
    return Pack.bigEndianToInt(xmssSig, 0) & 0xFFFFFFFFL;
}
```

If you run the example and all goes well, you should see the following output:

```
key index: 2
serial no: 1
sig index: 1
verifies : true
```

showing that the signature index stored in the signature is the same one used as the serial number for the certificate.

XMSS^MT

We have concentrated on XMSS in the examples here. Usage of XMSS^MT is basically the same, except the "XMSS" algorithm name being replaced with the "XMSSMT". The only difference applies to the use of the `KeyPairGenerator` which requires the use of the `XMSSMTParameterSpec`, which includes both the height h and the number of layers d. The following code fragment shows the creation of an XMSS^MT key pair with the same limit of 2^{20} signatures that we saw for the XMSS key pair we generated earlier:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("XMSSMT", "BCPQC");

kpg.initialize(new XMSSMTParameterSpec(20, 2, XMSSMTParameterSpec.SHA512));

KeyPair kp = kpg.generateKeyPair();
```

The second parameter to the `XMSSMTParameterSpec` is the number of layers, so this example will generate a key pair that works with 2 layers of XMSS trees that have a height of 20/2 each. This is one of the standard parameter sets and could also be written as `XMSSMTParameterSpec.XMSSMT_SHA2_20d2_512`.

Stateless Signature Algorithms

As the phrase "one-time signature" implies (at least in this case), you do not want to sign with the same private key twice as it weakens the overall security of the scheme as applied with the private key. The kind of issues that might arise here go from something as simple as signing a message and having a server crash before the updated key is persisted, or someone restoring a system off a backup not realising that they are also "winding back" the private key. Depending on what you are doing these are all things that can usually be managed, either by techniques like generating a new key pair on every sever startup, or persisting periodically and skipping a fixed number of keys before generating any published signatures (assuming a new key pair on a "restore from back up" event).

Stateless Hash-Based Signature Algorithms

For Hash-Based signature algorithms, one approach to eliminating state issues was proposed by Oded Goldreich [126] is to use random branches of the tree in the private key, and allow the security to degrade "gracefully" by ceasing to use the private key when the risk of a branch re-use becomes unacceptably high. This approach was built on the NIST PQC finalist SPHINCS+ [135], now called SLH-DSA and standardized in FIPS PUB 205 [20].

A look at SLH-DSA

SLH-DSA brings us back to more familiar terrain. The algorithm is stateless, in the sense that the private key does not change with each signature generated. SLH-DSA does this by combining a one-time signature mechanism over a very large tree, with a “few times” signature mechanism, and selected branches at random.

At the time of writing SLH-DSA is fully described in FIPS PUB 205. At the time of writing there is also a proposal to provide new parameter sets which are appropriate for people wanting to generate a reduced number of signatures. The accepted maximum number of signatures that can be generated with SLH-DSA is 2^{64}, which is quite a lot. This maximum does affect the size of the keys and signatures though, and a one of the areas of interest for SLH-DSA is code signing, so there is a discussion as to whether such a big range is always required - assuming a firmware update every month, 1200 signatures would get you through a 100 years, multiply that by 100 again to allow for accidents and you are still a long way from $2^6$4.

The following method will generate a SLH-DSA key pair, given a passed in parameter spec.

```java
/**
 * Generate a SLH-DSA key pair.
 *
 * @param slhdsaParameterSpec SLH-DSA parameter set to use.
 * @return a SLH-DSA key pair.
 */
public static KeyPair generateSLHDSAKeyPair(
    SLHDSAParameterSpec slhdsaParameterSpec)
    throws GeneralSecurityException
{
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("SLH-DSA", "BC");
    kpg.initialize(slhdsaParameterSpec, new SecureRandom());

    return kpg.generateKeyPair();
}
```

Once we have our key pair we can use the private key for signature generation. The below method allows a simple message to be signed.

```
/**
 * Generate an encoded SLH-DSA signature using the passed in SLH-DSA private key
 * and input data.
 *
 * @param slhdsaPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateSLHDSASignature(
    PrivateKey slhdsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SLH-DSA", "BC");

    signature.initSign(slhdsaPrivate);

    signature.update(input);

    return signature.sign();
}
```

You can see from the above, it is no more complicated than using any other signature algorithm we have seen before. Dealing with, or worrying about state, is now well behind us.

Lastly, we have a simple method for verifying the generated signature with our private key's corresponding public key.

```
/**
 * Return true if the passed in SLH-DSA signature verifies against
 * the passed in SLH-DSA public key and input.
 *
 * @param slhdsaPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifySLHDSASignature(
    PublicKey slhdsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SLH-DSA", "BC");

    signature.initVerify(slhdsaPublic);

    signature.update(input);

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

SLH-DSA parameters are all named (at least at the moment), so putting the above methods to generate a signature is just a matter of providing the named parameter spec that we want to use and we can make use of the methods described before as with the following sample code:

```
KeyPair spxKp = generateSLHDSAKeyPair(
                            SLHDSAParameterSpec.slh_dsa_sha2_128f);

byte[] sphincsPlusSig = generateSLHDSASignature(
                spxKp.getPrivate(), Strings.toByteArray("hello, world!"));

System.out.println("SLH-DSA verified: " + verifySLHDSASignature(
    spxKp.getPublic(), Strings.toByteArray("hello, world!"), sphincsPlusSig));
```

Running the sample code should produce the following output:

```
SLH-DSA verified: true
```

showing that the signature verified correctly.

Lattice-Based Signature Algorithms

The other two algorithms selected for standardisation in the signature category were CRYSTALS-Dilithium [140] (often just referred to as Dilithium, now called ML-DSA [19]) and Falcon [139] (not yet published but will apparently be FN-DSA). These algorithms, like SLH-DSA, are both stateless and also have named parameters. You can find the available parameters for ML-DSA in `org.bouncycastle.jcajce.spec.MLDSAParameterSpec` and the available parameters for Falcon in `org.bouncycastle.pqc.jcajce.spec.FalconParameterSpec`. Once again keep in mind these algorithms are not yet official standards, but what is currently implemented in Bouncy Castle will give you a good idea of where it is all heading.

As the usage for both algorithms just diverges in the choice of parameters, we will just look at an example of using ML-DSA here. In the case of Falcon, you can just substitute "ML-DSA" with "Falcon" and use one of the predefined FalconParameterSpec classes.

The following method will generate a ML-DSA key pair, given a passed in parameter spec.

```
/**
 * Generate a ML-DSA key pair.
 *
 * @param mldsaParameterSpec ML-DSA parameter set to use.
 * @return a ML-DSA key pair.
 */
public static KeyPair generateMLDSAKeyPair(
    MLDSAParameterSpec mldsaParameterSpec)
    throws GeneralSecurityException
{
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("ML-DSA", "BC");
    kpg.initialize(mldsaParameterSpec, new SecureRandom());

    return kpg.generateKeyPair();
}
```

Once we have our key pair we can use the private key for signature generation. The below method allows a simple message to be signed.

```
/**
 * Generate an encoded ML-DSA signature using the passed in ML-DSA private
 * key and input data.
 *
 * @param mldsaPrivate the private key for generating the signature with.
 * @param input the input to be signed.
 * @return the encoded signature.
 */
public static byte[] generateMLDSASignature(
    PrivateKey mldsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("ML-DSA", "BC");

    signature.initSign(mldsaPrivate);

    signature.update(input);

    return signature.sign();
}
```

Lastly, we have a simple method for verifying the generated ML-DSA signature with our private key's corresponding public key.

```
/**
 * Return true if the passed in ML-DSA signature verifies against
 * the passed in MLDSA public key and input.
 *
 * @param mldsaPublic the public key of the signature creator.
 * @param input the input that was supposed to have been signed.
 * @param encSignature the encoded signature.
 * @return true if the signature verifies, false otherwise.
 */
public static boolean verifyMLDSASignature(
    PublicKey mldsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("ML-DSA", "BC");

    signature.initVerify(mldsaPublic);

    signature.update(input);

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

ML-DSA parameters are all named (at least at the moment), so putting the above methods to generate a signature is just a matter of providing the named parameter spec that we want to use and we can make use of the methods described before as with the following sample code:

```
KeyPair dilKp = generateMLDSAKeyPair(
                            MLDSAParameterSpec.ml_dsa_65);

byte[] mldsaSignature = generateMLDSASignature(
                dilKp.getPrivate(), Strings.toByteArray("hello, world!"));

System.out.println("ML-DSA verified: " + verifyMLDSASignature(
    dilKp.getPublic(), Strings.toByteArray("hello, world!"),
                                          mldsaSignature));
```

Running the sample code should produce the following output:

```
ML-DSA verified: true
```

showing that the signature verified correctly.

Key Encapsulation Mechanisms

Key Encapsulation Mechanisms (KEM) are supported in Bouncy Castle in two ways. One is as a transport mechanism for other keys, which is done via the `Cipher` class and its key wrapping functionality. The other is as a seeding mechanism, where the seed key generated by the KEM is available for use directly. This last one is done via the `KeyGenerator` class. Both these approaches can be used with all the KEM algorithms listed here.

Classic McEliece

Classic McEliece [137] has now been moved to Round 4 of the NIST competition for algorithms that represent key encapsulation mechanisms. Apart from probably being one of the best researched algorithms in the PQC competition, due to its age (it goes back to the 1970s), it is also of interest as the round 3 version of the algorithm has also been accepted by BSI in Germany [136].

Key pair generation for Classic McEliece allows for keys to be generated with named parameter types representing 128 bit, 192 bit, and 256 bits of security. There are a total of 10 named parameter sets, with their security strength and key size in the table below.

Parameter Set	**Security Strength**	**Public Key Size**	**Cipher Text Size**
	(bits)	**(bytes)**	**(bytes)**
mceliece348864	128	261120	128
mceliece348864f	128	261120	128
mceliece460896	192	524160	188
mceliece460896f	192	524160	188
mceliece6688128	256	1044992	240
mceliece6688128f	256	1044992	240
mceliece6960119	256	1047319	226
mceliece6960119f	256	1047319	226
mceliece8192128	256	1357824	240
mceliece8192128f	256	1357824	240

It is worth pausing for a moment to contemplate the key sizes, yes, they really are in bytes! That said it is also worth contemplating the cipher text size as well - the biggest 240 bytes (1920 bits), even smaller than what you will get with a 2048-bit RSA key.

Classic McEliece is referred to in the BC provider with the algorithm name “CMCE”. Before we can do anything with the algorithm we first need to generate a key pair. We can do this by using one of the named parameter specifications provided as follows:

```
/**
 * Generate a Classic McEliece key pair.
 *
 * @param cmceParameters the Classic McEliece parameter set to use.
 * @return a Classic McEliece key pair.
 */
public static KeyPair cmceGenerateKeyPair(CMCEParameterSpec cmceParameters)
    throws GeneralSecurityException
{
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("CMCE", "BCPQC");
    kpg.initialize(cmceParameters, new SecureRandom());

    return kpg.generateKeyPair();
}
```

The `CMCEParameterSpec` has entries for each of the 10 given parameter sets. In addition to being able to make use of the `CMCEParameterSpec`, the CMCE `KeyPairGenerator` can also accept the `NamedParameterSpec` class that was added to the Java Security APIs with Java 11.

For Classic McEliece we will look at using the KEM via the `Cipher` class, so in this case the internally generated secret is used to generate a symmetric key wrapping key, which is used to encrypt the key being wrapped. The two cipher texts are then appended and sent. As you can see in the following example, written for the initiator, the concatenation will happen internally and a single byte array is returned.

```
public static byte[] cmceGeneratePartyU(PublicKey vPubKey, SecretKey secKey)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("CMCE", "BCPQC");

    cipher.init(Cipher.WRAP_MODE, vPubKey, new SecureRandom());

    return cipher.wrap(secKey);
}
```

Now that we generated the cipher text containing our encrypted secret and the wrapped key we pass the cipher text onto the recipient, and they reverse the operation, decrypting the secret, generating the wrapping key and then finally unwrapping the wrapped key and returning it. All this work is also performed hidden inside the `Cipher` which means the unwrapping code can be written as follows:

```
public static SecretKey cmceGeneratePartyV(
    PrivateKey vPriv, byte[] encapsulation, String keyAlgorithm)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("CMCE", "BCPQC");

    cipher.init(Cipher.UNWRAP_MODE, vPriv);

    return (SecretKey)cipher.unwrap(
                    encapsulation, keyAlgorithm, Cipher.SECRET_KEY);
}
```

Finally, here is an example of the full process which shows the generation of a Classic McEliece key pair, wrapping and unwrapping of a 128-bit AES key.

```
KeyPair kp = cmceGenerateKeyPair(CMCEParameterSpec.mceliece348864);
SecretKey aesKey1 = new SecretKeySpec(
            Hex.decode("000102030405060708090a0b0c0d0e0f"), "AES");

byte[] encKey = cmceGeneratePartyU(kp.getPublic(), aesKey1);

SecretKey aesKey2 = cmceGeneratePartyV(kp.getPrivate(), encKey, "AES");

System.out.println("keys match: "
    + Arrays.areEqual(aesKey1.getEncoded(), aesKey2.getEncoded()));
```

And, assuming nothing has gone wrong, running the fragment above should produce:

```
keys match: true
```

Showing that AES key has been transported correctly via the cipher text.

The parameter set used in the example, mceliece348864, has a security strength of 128 bits and produces a cipher text of 1024 bits. The combined cipher texts returned by the `cmceGeneratePartyU` will be a total 1216 bits long - 1024 for the encrypted secret, 192 for the wrapped key. As you are probably realising it is the small ciphertexts associated with Classic McEliece, as well as its robustness, which are the reason for the interest in it. If you swap to a more secure parameter set, such as mceliece6960119, and wrap a 256 bit AES key instead, you will find the total cipher text has only increased in size to 2128 bits, just slightly 2048 bits, but offering a higher level of security that 2048-bit RSA, which is currently estimated at 112 bits of security.

We will now look at an example of how to use the `KeyGenerator` class with a KEM. Classic McEliece supports this as well, but for interests sake we will look at an example using FrodoKEM.

FrodoKEM

FrodoKEM [138] was originally one of the NIST Alternative Candidate algorithms for key encapsulation mechanisms. It is also of interest as the round 3 version of the algorithm has also been accepted by BSI in Germany [136], so while it has been dropped from the NIST process it is likely there is still some life in it. There are two versions of the algorithm, which differ in terms of how the public matrix associated with the key pair involved is generated - in one case using the AES primitive with the other based on SHAKE primitive.

Key pair generation for FrodoKEM allows for keys to be generated with named parameter types representing 128 bit, 192 bit, and 256 bits of security, which yields public keys with sizes 9616, 25632, and 21520 bytes respectively. Taking into account the two versions, this makes for six parameter sets which can be used for generating key pairs.

As the following example shows, generating a key pair is just a matter of passing in the appropriate parameter set.

```
public static KeyPair frodoGenerateKeyPair(FrodoParameterSpec frodoParameters)
    throws GeneralSecurityException
{
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("Frodo", "BCPQC");
    kpg.initialize(frodoParameters, new SecureRandom());

    return kpg.generateKeyPair();
}
```

The FrodoKEM `KeyPairGenerator` can also accept the `NamedParameterSpec` class that was added to the Java Security APIs with Java 11.

While FrodoKEM also supports the mechanism we saw with Classic McEliece previously, in this example we are going to look at the initiator of a key exchange using the KEM operations where the secret seed generated by the KEM will be used for some other purpose later. As we want direct access to the secret generated by the KEM we use the `KeyGenerator` class and the `KEMGenerateSpec` which takes the recipient's public key and uses the KEM to generate an encapsulation and secret which is returned to us as an object implementing `SecretKeyWithEncapsulation`.

```
public static SecretKeyWithEncapsulation frodoGeneratePartyU(PublicKey vPubKey)
    throws GeneralSecurityException
{
    KeyGenerator keyGen = KeyGenerator.getInstance("Frodo", "BCPQC");

    keyGen.init(new KEMGenerateSpec(vPubKey, "AES"), new SecureRandom());

    return (SecretKeyWithEncapsulation)keyGen.generateKey();
}
```

The getEncoded() method on SecretKeyWithEncapsulation returns the secret seed generated by the encapsulation. The SecretKeyWithEncapsulation class also has a getEncapsulation() method which returns the encryption of the source of the seed for sending to the recipient. SecretKeyWithEncapsulation also implements Destroyable so the contents of the object can be erased when no longer required.

On the recipient's side we still use the KeyGenerator class but this time we make use of the KEMExtractSpec which uses the recipient's private key and the encapsulation we received from the initiator.

```
public static SecretKeyWithEncapsulation frodoGeneratePartyV(
    PrivateKey vPriv, byte[] encapsulation)
    throws GeneralSecurityException
{
    KeyGenerator keyGen = KeyGenerator.getInstance("Frodo", "BCPQC");

    keyGen.init(new KEMExtractSpec(vPriv, encapsulation, "AES"));

    return (SecretKeyWithEncapsulation)keyGen.generateKey();
}
```

As you can see, for consistency, the method again returns a SecretKeyWithEncapsulation implementation.

We can put all these methods together to provide a short example of a secret seed exchange using FrodoKEM. If you want to try it for yourself, have a go at the following:

```
      KeyPair kp = frodoGenerateKeyPair(FrodoParameterSpec.frodokem640aes);

      SecretKeyWithEncapsulation secEnc1 = frodoGeneratePartyU(kp.getPublic());

      SecretKeyWithEncapsulation secEnc2 = frodoGeneratePartyV(
                          kp.getPrivate(), secEnc1.getEncapsulation());

      System.out.println("secrets match: "
          + Arrays.areEqual(secEnc1.getEncoded(), secEnc2.getEncoded()));
```

Assuming everything has gone according to plan you should see:

```
secrets match: true
```

as the output, indicating the encapsulation has been generated and decrypted successfully. If you want to generate a secret key from the secret, and you are not just encrypting otherwise “random data” (so another secret or private key) as is the case with the key transport mechanism with we saw with Classic McEliece earlier, the secret does make excellent input into a KDF or an XOF like SHAKE. We will see an example of this when we look at Hybrid Encryption a bit later. It is also worth modifying the example to try a few different parameters. In Bouncy Castle’s case you will find the size of the secret changes with the security level of the parameters used, with the bit strength of the secret key reflecting the security strength of the parameters.

ML-KEM

The last algorithm we will look at in this category is ML-KEM [18], a KEM relative of ML-DSA.

ML-KEM was originally based on CRYSTALS-Kyber. At the time of writing it is the only PQC KEM algorithm that has been standardized by NIST, but the round 4 algorithm HQC [110], which is code based, has also been selected for standardization. Keep in mind “selected” does not mean the same as “standardized” so expect the algorithm parameters to change (a bit).

As the following example shows, other than the choice of parameter spec and algorithm name generation of a key pair for ML-KEM is virtually identical the process followed with FrodoKEM:

```
public static KeyPair mlkemGenerateKeyPair(MLKEMParameterSpec mlkemParameters)
    throws GeneralSecurityException
{
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("ML-KEM", "BC");
    kpg.initialize(mlkemParameters, new SecureRandom());

    return kpg.generateKeyPair();
}
```

ML-KEM can also be used with the `Cipher` class, but in this case we will restrict ourselves to the use of the KeyGenerator. We are doing this as we will make use of the PartyU and PartyV methods when we look at hybrid encryption later in the chapter.

Here is the PartyU, or initiator, method for ML-KEM. As before the method uses the ML-KEM public key and a SecureRandom to generate the shared secret and an encapsulation derived from the public key and the shared secret value.

```
public static SecretKeyWithEncapsulation mlkemGeneratePartyU(PublicKey vPubKey)
    throws GeneralSecurityException
{
    KeyGenerator keyGen = KeyGenerator.getInstance("ML-KEM", "BC");

    keyGen.init(new KEMGenerateSpec(vPubKey, "AES"), new SecureRandom());

    return (SecretKeyWithEncapsulation)keyGen.generateKey();
}
```

Next we have the PartyV, or responder/receiver, method.

```
public static SecretKeyWithEncapsulation mlkemGeneratePartyV(
    PrivateKey vPriv, byte[] encapsulation)
    throws GeneralSecurityException
{
    KeyGenerator keyGen = KeyGenerator.getInstance("ML-KEM", "BC");

    keyGen.init(new KEMExtractSpec(vPriv, encapsulation, "AES"));

    return (SecretKeyWithEncapsulation)keyGen.generateKey();
}
```

As in the FrodoKEM example we recover the shared secret generated by the initiator using the ML-KEM private key.

Finally here is a simplified example of the secret seed exchange using ML-KEM - as you have likely realised it is mostly identical to FrodoKEM. If you want to try it for yourself, try running the following:

```
    KeyPair kp = mlkemGenerateKeyPair(MLKEMParameterSpec.ml_kem_1024);

    SecretKeyWithEncapsulation secEnc1 = mlkemGeneratePartyU(kp.getPublic());

    SecretKeyWithEncapsulation secEnc2 = mlkemGeneratePartyV(
                        kp.getPrivate(), secEnc1.getEncapsulation());

    System.out.println("secrets match: "
        + Arrays.areEqual(secEnc1.getEncoded(), secEnc2.getEncoded()));
```

If the derivation and decapsulation process has gone correctly, you should see:

```
secrets match: true
```

when running the code fragment - indicating that both sides have, again, been able to derive the same secret.

Hybrid Encryption

While the presence of new algorithms does mean that help is on the way, another idea that has been getting a bit of attention is the area of hybrid encryption, where a post-quantum is used to quantum harden an existing encryption scheme. This can be very useful where longer term protection is wanted for messages so adding a touch of quantum hardness now is seen as hopefully removing the need to do re-encryption later.

To give an idea of how this works, it is worth revisiting EC key-agreement, as the mechanism does lend itself readily to using an additional algorithm in parallel. With the arrival of NIST SP 800-56C Rev. 2 [11] there are also now two accepted certified compliant mechanisms for adding in an external PQC algorithm as well, of particular importance now as we are probably looking at 2024 before certifiable KEM algorithms arrive. We will have a look at the approach using the OtherInfo structure first and the NIST SP 800-56C

Hybrid Encryption Using OtherInfo - Setup.

To provide a ready example of doing hybrid encryption we will provide a simple class to generate an EC key pair to provide as a foundation for the hybrid scheme.

```java
public class KasParty
{
    protected final KeyPair kp;

    KasParty(SecureRandom random)
        throws GeneralSecurityException
    {
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("EC", "BC");

        kpGen.initialize(new ECGenParameterSpec("P-256"), random);

        this.kp = kpGen.generateKeyPair();
    }

    public PublicKey getPublic()
    {
        return kp.getPublic();
    }
}
```

We will use the ML-KEM PQC encryption algorithm in this case, although the other ones, such as FrodoKEM, can also be used.

The generator needs two extension classes, not one, to implement the two sides of agreement between the parties U and V. The reason for this is that the initiator side (or party U) will produce a public key and the responder side, or party V, will produce the encapsulation containing the encapsulated secret value which party U must then extract.

We will look at the U side first.

Hybrid Encryption Using OtherInfo - Party U

The following code implements the U side of the agreement.

```java
public class PartyU
    extends KasParty
{
    PQCOtherInfoGenerator.PartyU infoGenerator;

    PartyU(SecureRandom random, byte[] uInfo, byte[] vInfo)
        throws GeneralSecurityException
    {
        super(random);

        infoGenerator = new PQCOtherInfoGenerator.PartyU(
            MLKEMParameters.ml_kem_512,
            new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256),
            uInfo, vInfo, random);
    }

    /**
     * Return the initiator stage 1 value for the NewHope agreement.
     * @return a byte[] based on a NewHope public key.
     */
    public byte[] doStage1()
    {
        return infoGenerator.getSuppPrivInfoPartA();
    }

    /**
     * Return the KDF generated agreed value based on the responses from V.
     *
     * @param otherKey the V party EC key.
     * @param partBInfo the V party NewHope response.
     * @return the KDF generated secret from the agreement.
     */
    public byte[] doAgreement(PublicKey otherKey, byte[] partBInfo)
        throws GeneralSecurityException, IOException
    {
        KeyAgreement agreement = KeyAgreement.getInstance(
                                    "ECCDHwithSHA256KDF", "BC");

        agreement.init(kp.getPrivate(),
            new UserKeyingMaterialSpec(
                    infoGenerator.generate(partBInfo).getEncoded()));

        agreement.doPhase(otherKey, true);

        return agreement.generateSecret();
    }
}
```

You can see the core difference between this and a regular EC agreement is the use of the PQCOtherInfoGenerator class - in this case the PartyU branch. The PQCOtherInfoGenerator.PartyU class is used to create an infoGenerator object which is used both to provide an additional “stage1” value to pass to party V during the exchange. The infoGenerator object is also used to process the party V’s response to produce a user keying material to include in the KDF which produces the final agreed value.

Hybrid Encryption Using OtherInfo - Party V

Here is the code implementing the V side of the agreement.

```
public class PartyV
    extends KasParty
{
    PQCOtherInfoGenerator.PartyV infoGenerator;

    PartyV(SecureRandom random, byte[] uInfo, byte[] vInfo)
        throws GeneralSecurityException
    {
        super(random);

        infoGenerator = new PQCOtherInfoGenerator.PartyV(
            MLKEMParameters.ml_kem_512,
            new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256),
            uInfo, vInfo, random);
    }

    /**
     * Return the NewHope based response to U's initial value.
     *
     * @param partAInfo the NewHope based agreed response.
     * @return the response value to pass to U.
     */
    public byte[] doStage2(byte[] partAInfo)
    {
        return infoGenerator.getSuppPrivInfoPartB(partAInfo);
    }

    /**
     * Return the KDF generated agreed value based on the U public key.
     *
     * @param otherKey the U party EC key.
     * @return the KDF generated secret from the agreement.
     */
    public byte[] doAgreement(PublicKey otherKey)
        throws GeneralSecurityException, IOException
    {
        KeyAgreement agreement = KeyAgreement.getInstance(
                                             "ECCDHwithSHA256KDF", "BC");

        agreement.init(kp.getPrivate(), new UserKeyingMaterialSpec(
                                 infoGenerator.generate().getEncoded()));

        agreement.doPhase(otherKey, true);

        return agreement.generateSecret();
    }
}
```

As you probably already expected, the class also makes use of an `infoGenerator` object. This time created from `PQCtherInfoGenerator.PartyV`. What you will notice that is different in this case is that the `infoGenerator` object requires the input from party U that comes from its `doStage1()` method in order to provide a stage 2 response via the `doStage2()` method. Again the `infoGenerator` object is also used to produce a user keying material spec to include in the KDF which produces the final agreed value. Nothing is passed to the `infoGenerator` in this case as the processing of party U's stage 1 data was carried out in the `doStage2()` method.

Hybrid Encryption Using OtherInfo - Putting it Together

Here is an example of the two classes used together to conduct an agreement which also includes use of the `PQCOtherInfoGenerator`.

```
SecureRandom random = new SecureRandom();

byte[] partyUId = Strings.toByteArray("PartyU");
byte[] partyVId = Strings.toByteArray("PartyV");

PartyU partyU = new PartyU(random, partyUId, partyVId);

PartyV partyV = new PartyV(random, partyUId, partyVId);

System.out.println(Hex.toHexString(
    partyU.doAgreement(partyV.getPublic(),
                partyV.doStage2(partyU.doStage1()))));
System.out.println(Hex.toHexString(
    partyV.doAgreement(partyU.getPublic())));
```

The example is a bit oversimplified for the sake of brevity - in this situation you would expect the stage 1 and stage 2 values to be exchanged across the wire during the agreement. The implication of this is that if you wanted to do a one pass agreement it could only from V to U as U would need to pre-publish its stage in addition to its public key. That said, the additional effort should make it significantly harder to attack the key agreement as part of the information mixed into the KDF used to generate the shared secret will include the agreement value calculated using ML-KEM (in this case).

As you can see from the code, the sample just prints the shared secrets as a hex string. As this algorithm is using random seeding you will likely not see the output values we present here. You should see output which is a similar length and consists of two lines which are the same though. Our latest run produced this:

```
cd40dc777d3ce109f84660132ebeb52268c2a5aaa2b17a232c397b6d3b33e94c
cd40dc777d3ce109f84660132ebeb52268c2a5aaa2b17a232c397b6d3b33e94c
```

indicating that the result of the U party agreement was the same as the result of the V party agreement. While you will see different values you should see both lines have the same different value to what we did.

Hybrid Encryption - What's going on

It is useful to be able to describe this process in the same terms as the standards. In this case what we are really doing is taking advantage of a field defined in the NIST document SP 800-56A [9] Section 5.6.2 as FixedInfo. FixedInfo forms part of the extra data that is provided to a KDF for generating a secret key from a key agreement algorithm know as OtherInput. One of the components of FixedInfo can be a shared secret value, shared through some other mechanism to the agreed value Z calculated by the key agreement.

In the example given the ML-KEM algorithm is used to calculate this agreed upon secret. We are able to rely on the traditional EC algorithm for our known security level and the worst case scenario, should it turn out that ML-KEM is in some way broken, is that the Key Agreement performed will have the same security level as the traditional EC algorithm. On the other hand if ML-KEM turns out to really be quantum hard, then recovering the agreed symmetric key will involve attacking both the EC algorithm and ML-KEM - leaving even the user of a quantum computer with half the puzzle to solve.

By now there should be no "if" associated with ML-KEM. At the time of writing the NIST post-quantum competition is still running so you can expect ML-KEM to be joined by other algorithms as well with eventual standards coming out in the middle of 2024.

Hybrid Encryption: SP 800-56C Style

With the release of SP 800-56C Rev 2 [11] and some new Two-Step KDFs, NIST provided a slightly different approach to the idea of adding other shared secret data to a key agreement. Section 2 of the recommendation allows for the use of a "hybrid" shared secret of the form $Z' = Z \parallel T$ meaning Z' is a concatenation of the standard shared secret value Z and an auxiliary shared secret T which is generated through some other agreed mechanism by the parties involved in the agreement.

As of Bouncy Castle 1.71, KAS support is provided for this approach using the `HybridValueParameterSpec` class which allows for the addition of T value with a regular KAS parameter spec. In some ways the approach appears cleaner than using the KAS OtherInfo field, although it should be pointed out that the OtherInfo field does server other purposes, so you may end up with something similar regardless. That said, when used with a Two-Step KDF, such as HKDF, this approach is believed to offer better security as the everything making up Z' will be double hashed. In any case, the SP 800-56C approach is certainly easier to demonstrate.

Here is a demonstration of the use of `HybridValueParameterSpec` which should illustrate the last point:

```java
/**
 * Generate an agreed AES key value of 256 bits in length
 * using both UKM and a T value.
 *
 * @param aPriv Party A's private key.
 * @param bPub Party B's public key.
 * @param tValue T value to create Z || T.
 * @param keyMaterial user keying material to mix with Z || T.
 * @return the generated AES key (256 bits).
 */
public static SecretKey ecGenerateHybridAESKey(
    PrivateKey aPriv, PublicKey bPub, byte[] tValue, byte[] keyMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement =
        KeyAgreement.getInstance("ECCDHwithSHA256KDF", "BC");

    agreement.init(aPriv, new HybridValueParameterSpec(
             tValue, new UserKeyingMaterialSpec(keyMaterial)));

    agreement.doPhase(bPub, true);

    return agreement.generateSecret("AES");
}
```

As you can see the `HybridValueParameterSpec` is just a wrapper class, internally the Bouncy Castle KAS implementation will retrieve the T value and then strip off the `HybridValueParameterSpec`, passing the inner algorithm parameter spec down to the KDF generator inside the implementation.

Calculating the T still requires the use of another mechanism, be it a KAS or a KEM. The following sample code shows the use of the `HybridValueParameterSpec` with the ML-KEM algorithm.

```
// PartyU
KeyPair ecKpU = EcUtils.generateECKeyPair("P-256");

// PartyV, in this case has a PQC key pair as well.
KeyPair ecKpV = EcUtils.generateECKeyPair("P-256");
KeyPair qKpV = MLKEMExample.mlkemGenerateKeyPair(
                   MLKEMParameterSpec.ml_kem_512);

// User keying material - shared.
byte[] ukm = Hex.decode("030f136fa7fef90d185655ed1c6d46bacdb820");

SecretKeyWithEncapsulation secEncU = MLKEMExample.mlkemGeneratePartyU(
                                                      qKpV.getPublic());

SecretKey secKeyU = ecGenerateHybridAESKey(
      ecKpU.getPrivate(), ecKpV.getPublic(), secEncU.getEncoded(), ukm);

SecretKeyWithEncapsulation secEncV = MLKEMExample.mlkemGeneratePartyV(
                         qKpV.getPrivate(), secEncU.getEncapsulation());

SecretKey secKeyV = ecGenerateHybridAESKey(
    ecKpV.getPrivate(), ecKpU.getPublic(), secEncV.getEncoded(), ukm);

System.out.println("U and V Shared Key equal: " +
        Arrays.areEqual(secKeyU.getEncoded(), secKeyV.getEncoded()));
```

While, in this case, the recipient V is the one with the PQC key pair, the only requirement for this to work is for at least one party to have a PQC public key that can be used. Keep in mind that this does mean there is potentially an extra key exchange in the protocol as well and that PQC public keys are often a bit bigger than what we are used to. If you are looking for 256 bit security with Classic McEliece for example, while the cipher texts are fairly small, the public key size is over one meg! Some thought is required.

Summary

There is a major update for public key encryption heading towards us in the shape of Quantum Computers. Unfortunately, even if you are a “Quantum Sceptic” and do not think you will see a Quantum Computer soon, the reality is classical attacks on algorithms like RSA and EC are also improving.

In order to prepare for this change NIST have been running a Post Quantum Cryptography (PQC) competition to produce new algorithms which can protect against Quantum Computer based attacks.

This competition will start producing actual standards in 2024. In the interim, for signatures, there are two Quantum Safe signature algorithms, LMS and XMSS which can be used (with caution as they are also stateful) now. The competition has also produced some useful KEM (Key Encapsulation Mechanism) algorithms which can be used for secret exchange now for short term encryption protocols, such as those making use of hybrid cryptography. While the stateless signature algorithms and KEMs currently published are still evolving, the algorithms currently published provide a very good introduction if you wish to experiment with PQC cryptography.

Yes, this is the end of the book, but the topic is by no means closed. There is no doubt that there will be further algorithms, and insights, coming. Thanks for reading, best luck for all of us!

Appendix A: ASN.1 and Bouncy Castle

The vast majority of standards involving cryptography use ASN.1 (or Abstract Syntax Notation 1), which is primarily a language for describing how objects are encoded for transmission. Algorithm parameters, keys, signatures, signed and encrypted messages, you name it! There is a good chance there will be an ASN.1 structure somewhere for encoding it.

The ASN.1 standards (X.680 and its associates X.681, X.682, X.683, X.684, X.691, and X.693) came out of standards work done jointly by ISO and CCITT (now ITU-T) that started in the early 1980s. ASN.1 is big, but fortunately a full understanding of the standard is not required to be able to apply it. One way around this complexity is to make use of ASN.1 parsers for generating code which then allows objects to be encoded, another approach is to provide an API which allows specific encodings to be defined, encoded, and parsed. Here at Bouncy Castle we used the second approach - we provide an API.

We have stayed with an API, rather than trying to go down the compiler path, as while a lot of IETF standards (for example) use ASN.1, they don't all use the same version the ASN.1 language (in some cases in the same document, RFC 5755 [77] for example). Over the passage of time, various things have been tweaked, and, as a result, sometimes a bit of flexibility is required interpreting a structure to allow for the production of correct encodings, or on the occasion where compatibility is the biggest problem, compatible encodings[29].

Being able to use the API effectively means having some understanding of three things. You need to have a basic understanding of what you are looking at when you read an ASN.1 module definition. You need to have some knowledge of how the module definition can then be expressed using the BC API. Finally, you need to know how the encoding rules need to be used in the case you are looking at - this is particularly important in the case of anything which is signed or has a MAC calculated for it. ASN.1 provides a set of encoding rules, the Distinguished Encoding Rules, which is used in these cases - the idea being that if the encoding system is followed the ASN.1 structures and objects encoded into the output stream will always produce the same MAC or signature digest. The other encoding rules that are of interest to us are the Basic Encoding Rules, in particularly the subset of Basic Encoding Rules which only uses definite length encoding.

Basic ASN.1 Syntax

Fortunately the bits of ASN.1 syntax we have to deal with are relatively simple, the fun really starts with using the encoding rules. We will take a look at the basic elements here.

[29] Naturally we would not encourage bending any existing standards of encoding, but as anyone who has faced a legacy system that "simply must be worked with" will attest, the choice between correctly meeting ISO/ITU-T restrictions and total project failure is one that sometimes has to be made. There is no prize for guessing how the choice needs to turn out.

Comment Syntax

ASN.1 has two distinct commenting styles: one for blocks, which are delimited by /* and */, and one for single-line comments, which start and end with --.

Unlike Java, block comments can be nested, however as the block syntax is also a more recent addition to ASN.1, you will often see multiple lines of single-line comments used where you might otherwise expect to see a block comment.

Object Identifiers

The ASN.1 Object Identifiers (OID) was introduced to allow the construction of a globally unique namespace. An OID is made up of a list of numbers separated by dots ('.') and the best way to think of how this works is that the structure of an OID follows a path, often referred to as an arc, through a tree which is extended of 3 primary branches. Procedures exist for organisations under any of the 3 primary branches to extend the tree further and this allows for any organisation, or individual, to be allocated an OID which they can then extend themselves in order to provide a unique identifier for things like algorithms, data types, and algorithm parameters which can then be shared with others.

The three primary branches are allocated according to where an organisation initially fits in the ISO/ITU-IT universe, with the number 0 allocated for ITU-T, the number 1 allocated for ISO, and the number 2 allocated to joint ISO/ITU-T organisations. After that the numbering system is arbitrary and depends on the allocation made by the owner of the previous OID that represents the branch the new OID is being allocated on.

For example, the NIST OID for ECDSA using SHA3-512 is 2.16.840.1.101.3.4.3.12 which in long form is described as:

```
{ joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101) csor(3)
                     nistAlgorithms(4) sigAlgs(3) id-ecdsa-with-sha3-512(12) }
```

Which is to say that the Computer Security Objects Register (CSOR) at NIST has been allocated an OID under the joint ISO/ITU-T (2) US country branch (16) for organizations (1) in the government (101). The CSOR then allocated a further branch for NIST algorithms (4) with a subbranch for signature algorithms (3). Finally, we arrive at the actual signature algorithm type ECDSA with SHA3-512 (12).

Generally, the best practice if an OID is already allocated for a particular algorithm is to always use the already allocated one. Occasionally, as the branches are based around organisations rather than algorithms, you will find that sometimes a given algorithm has more than one OID associated with it. As this can lead to unexpected failures in processing, it is worth keeping this in mind.

The Module Structure

The basic structure for an ASN.1 module is as follows:

```
module_name { object_identifier }
DEFINITIONS tagging TAGS ::=
BEGIN
EXPORTS export_list ;
IMPORTS import_list ;

body

END
```

Where *module_name* and *object_identifier* serve to identify the module being defined, with the expectation being that the OID specified is unique. The *tagging* field defines the kind of tagging to be used in the module. The *body* section contains the local definitions of structures that are defined in the module.

As an example the module defined in A.1 in RFC 5280 (PKIX) starts with:

```
PKIX1Explicit88 { iso(1) identified-organization(3) dod(6) internet(1)
  security(5) mechanisms(5) pkix(7) id-mod(0) id-pkix1-explicit(18) }

DEFINITIONS EXPLICIT TAGS ::=

BEGIN

-- EXPORTS ALL --

-- IMPORTS NONE --
```

So the module name is `PKIX1Explict88` and its unique identifier is `1.3.6.1.5.5.7.0.18`. The

```
DEFINITIONS EXPLICIT TAGS ::=
```

tells us that the tagging field is `EXPLICIT` and by default any tagged items in the module are `EXPLICIT` tagged unless specified otherwise. The alternative is `IMPLICIT`, we will see what the difference between these two is a bit further on.

In this case the `EXPORTS` keyword is missing (it is in a comment), which means that anything in the module can be exported to another one. If you ever see that *export_list* is missing, so the header has just `EXPORTS ;` then nothing can be exported from the module.

The `IMPORTS` keyword is also missing, and as the comment implies this means that no imports are required, so everything you might need to handle is already described in the module. If you look further along in RFC 5280 at the module in A.2, `PKIXImplicit88` you will see an example of an import clause:

```
IMPORTS
      id-pe, id-kp, id-qt-unotice, id-qt-cps,
      -- delete following line if "new" types are supported --
      BMPString, UTF8String,  -- end "new" types --
      ORAddress, Name, RelativeDistinguishedName,
      CertificateSerialNumber, Attribute, DirectoryString
      FROM PKIX1Explicit88 { iso(1) identified-organization(3)
            dod(6) internet(1) security(5) mechanisms(5) pkix(7)
            id-mod(0) id-pkix1-explicit(18) };
```

In this case the `PKIXImplicit88` module is importing a list of definitions from a single module, the `PKIXExplicit88` module which is also defined in RFC 5280.

The Types

ASN.1 types are all in one of three categories: simple types, string types, and structured types. String types are further subdivided into two more categories, raw bit types and those that represent specific character encodings. The structured types represent two container types: an order sequence of objects and an unordered set of objects. These types are all defined in the package `org.bouncycastle.asn1`.

Simple Types

The simple types in ASN.1 are

BOOLEAN

represents a true or false value. Represented by `ASN1Boolean`, with `ASN1Boolean.TRUE` and `ASN1Boolean.FALSE` representing the two constants.

ENUMERATED

is a special case of INTEGER, conventionally from a restricted set. Represented by `ASN1Enumerated`.

INTEGER

an INTEGER can be used to represent integers of any magnitude. Note: this means they are signed and have only one form of encoding (two's-complement). INTEGER is represented by `ASN1Integer`.

NULL

an explicit null value. It is important not to get NULL confused with the Java `null`, the Java `null` is really closer to the ASN.1 idea of absent, meaning no encoding is present. NULL has a real encoding to represent it. It is represented by `ASN1Null` and a legacy `DERNull`, with `DERNull.INSTANCE` providing a constant value.

OBJECT IDENTIFIER

we covered this type in the section on Object Identifiers above. This is the actual type definition they have and they are represented by the `ASN1ObjectIdentifier` class.

UTCTime

a UTCTime is used to define a “Coordinated Universal Time” with a two-digit year. Generally the two-digit year is interpreted as representing the years from 1950 to 2049, but there are apparently some systems out there that use a different window, so it is always worth checking how to convert these. The lowest level of resolution for a UTCTime is seconds. UTCTime is represented by the `ASN1UTCTime` and `DERUTCTime` classes.

GeneralizedTime

a GeneralizedTime has a 4 digit year and can be used to represent an arbitrary precision of seconds. UTCTime is represented by the `ASN1GeneralizedTime` and `DERGeneralizedTime` classes.

Most of the primitive types have only one encoding, so whether you are using DER, or the more relaxed BER, they will always encode the same way. The two exceptions are UTCTime and GeneralizedTime. Both these primitives encode a time as an ASCII string. For CER and DER, in the case of GeneralizedTime this means the seconds element will always be present and fractional seconds, where present, will omit trailing zeroes and the string will terminate with a Z, so with UTC/GMT as the timezone. UTCTime follows a similar pattern. For CER and DER a UTCTime value is encoded with seconds always present and also terminating with a Z. If you can always try and follow DER for these, Bouncy Castle does in general, as it is really easy to encode a primitive in a piece of data that may end up being used inside a signature and unexpected failures will show up on verification if the signature verifier is, probably correctly, enforcing DER encoding.

String Types

String types are again divided into two further categories. Bit string types and character string types.

Bit String Types

There are two bit string types:

BIT STRING

BIT STRING allows for the storage of an arbitrarily long string of bits. They are encoded in two parts with the first part being the number of pad bits followed by a string of octets making up the actual bit string. In DER encoding the pad bits should all be zero. BIT STRING is represented by `ASN1BitString` and `DERBitString`.

OCTET STRING

OCTET STRING allows for the storage of an arbitrarily string of octets (for the more modern amongst us, a byte array). OCTET STRING is represented by `ASN1OctetString`, `BEROctetString`, and `DEROctetString`. There are also streaming classes for BER OCTET STRING processing - `BEROctetStringGenerator` and `BEROctetStringParser`.

Character String Types

ASN.1 supports a wide variety of character string types, some of which might bring tears of nostalgia (or perhaps shrieks of horrors remembered) to more “seasoned” readers of this book. Some of these string types represent character sets that have since been retired so you will rarely, if ever, see them (VideotexString and Teletex are the two best examples of this), others such as PrintableString, IA5String, BMPString and UTF8String are probably the most common. As with the bit string types, all these allow for the storage of arbitrarily long strings, but in the case of what we do at Bouncy Castle we have only found situations where DER and direct-length encodings are required.

The full list of string types and what they represent is as follows:

BMPString

BMPString is represented by the `DERBMPString` class. It takes its name from the “Basic Multilingual Plane” - a construction that contains all the characters associated with the “living languages”. The character set is represented by ISO 10646, the same set represented by Unicode.

GeneralString

GeneralString is represented by the `DERGeneralString` class. It can contain any of the characters described in the International Register of Coded Character Sets described in ISO 2375 including control characters.

GraphicString

GraphicString is represented by the `DERGraphicString` class. This string type is derived from the same character set as GeneralString but without the control characters. Only the printing characters are allowed.

IA5String

An IA5String is made up of characters from “International Alphabet 5”, an old ITU-T recommendation. These days they are considered to cover the whole of the ASCII character set. The type is represented by `DERIA5String`.

NumericString

A NumericString contains only the digits 0 to 9 and the space character. It is represented by `DERNumericString`.

PrintableString

The PrintableString is represented by `DERPrintableString`. PrintableString draws its character set from a subset of ASCII: the letters “A” to “Z”, “a” to “z”, “0” to “9”, and the additional characters “ “, “’”, “(“, “)”, “+”, “-“, “.”, “:”, “=”, “?”, “/”, and “,”.

TeletexString

Originally known as the T61String and represented by the `DERT61String`. It can be difficult to interpret properly as while it is an 8 bit character type it supports changing character set by using a sequence of characters starting with ASCII ESC - the escape character.

UniversalString

UniversalString is represented by `DERUniversalString`. This string type was also added for internationalization, but in this case uses a character size of 32 bits. By default Bouncy Castle will convert these to strings as HEX strings showing the character encodings.

UTF8String
UTF8String represents the encoding "Universal Transformation Format, 8 bit" and is the recommended string type for full internationalization. It is represented by the `DERUTF8String` and is now very widely used.

VideotexString
As the name suggests, VideotexString was designed for use with videotext systems and to accommodate 8-bit characters that can build simple images using control codes.

VisibleString
VisibleString is represented by `DERVisibleString`. Originally this type was meant to contain only characters from ISO 646, but since 1994 it has been interpreted as containing plain ASCII without the control characters.

Container Types

There are two container types in ASN.1 - the SET and the SEQUENCE. These are supported in the BC ASN.1 API using the `ASN1Set` and the `ASN1Sequence` class at the top most level.

Both the `ASN1Set` and the `ASN1Sequence` classes are abstract and have implementation classes that hint at a particular encoding style. ASN.1 SEQUENCE structures are ordered so the choice of encoding style does not affect the order of the elements in the encoding. ASN.1 SET structures on the other hand are not ordered, but in order to meet the requirements of DER, are written out with the elements sorted according to the numeric value of their encodings (calculated by encoding the element and then converting it into an unsigned integer after adding enough trailing zeroes to make each encoding the same length).

The CHOICE Type

CHOICE is a special type which indicates that an ASN.1 field, or variable, will be one of a group of possible ASN.1 types or structures. There isn't really a parallel to this in Java, but if you think of a union in C or Pascal you would be close.

Bouncy Castle provides a marker interface `ASN1Choice` which can be implemented on ASN.1 objects which represent choice types. This marker interface should be used where you can as it helps to alert the Bouncy Castle encoders to enforce the correct encoding rule for CHOICE (more on this below).

Encoding Rules

ASN.1 supports a range of different ways of encoding things, all the way from a carefully crafted binary format which is unambiguous and minimal in size to a general encoding using XML. From the point of view of what we currently do at Bouncy Castle there are three sets of encoding rules which are most relevant: the Basic Encoding Rules (BER), the Distinguished Encoding Rules (DER), and to a lesser extent the Canonical Encoding Rules (CER).

Basic Encoding Rules

The core of the ASN.1 encoding system is built around the Basic Encoding Rules (BER). BER encoding follows the tag-length-value (TLV) convention. The type is identified using the tag, a value giving the length of the content is next, and then a length's worth of octets describing the content comes next. Both DER and CER are a subset of BER.

BER encoding offers three methods for encoding an ASN.1 structure or primitive:

Primitive definite-length
in this case the whole primitive is described in a single chunk - the length and data following the type tag tell you everything you need to know about size and data in the encoded structure.

Constructed definite-length
constructed definite-length is used both to represent ASN.1 structures containing multiple primitives (possibly constructed themselves), and also to represent primitives which are defined as the aggregate of a list of primitive definite length encodings. In the case of one of these you can only determine the real size of the data, as well as the octets making it up, by reading all the primitive encodings make up the constructed encoding.

Constructed indefinite-length
a constructed indefinite-length encoding is also made up of other encodings - possibly indefinite-length themselves. It is indicated with by a length octet of `0x80` and terminated by an end-of-contents marker made up of two octets with the value `0x00`. This encoding appears a lot - it is the only way to encode data that may be larger than memory resources allow to be comfortably dealt with, or where other considerations make it impossible to work out how long the encoded data may be initially.

Tagging

Being a TLV format, BER has a standard set of predefined tags for common primitives and structured types. BER also allows you to define your own tag values for situations where a definition for a container type like a SEQUENCE might otherwise be ambiguous.

There are a couple of twists on defining your own tag values though.

The first twist is that an ASN.1 object can be tagged EXPLICIT which means that your tag value and the object's original tag value are included in the encoding or the ASN.1 object can be tagged IMPLICIT which means that your tag value replaces the object's original tag value in the encoding. Of course, as the original tag value is lost, you may ask yourself how you can interpret the encoding correctly if you do not know what the original tag value is and you do not have any documentation on hand telling you what it should have been. Sadly, you cannot tell. This becomes particularly important with structured types as an explicitly tagged primitive will generate the same encoding as an implicitly tagged SET or SEQUENCE containing a single primitive.

All of the standard API types in the Bouncy Castle ASN.1 library support static `getInstance()` methods which take `ASN1TaggedObject` and a `boolean` indicating implicit or explicit tagging. Because of the ambiguity around IMPLICIT and structured types, it is very important to use these methods when interpreting objects of type `ASN1TaggedObject` and to make sure you use the correct value for the `boolean` concerning the type of tagging used.

When interpreting encodings containing IMPLICIT tagging, you must write code to interpret each IMPLICIT tag correctly.

The second twist is that where the ASN.1 object is a CHOICE. For CHOICE type you cannot afford to lose any tag associated with the CHOICE value as it is the only clue you have as to what the CHOICE value really represents.

If you tag an ASN.1 object of type CHOICE, the tagging is always done using the EXPLICIT style, regardless of what the ASN.1 module default might be.

Bouncy Castle provides a marker interface `ASN1Choice` which can be implemented on ASN.1 objects to alert the Bouncy Castle encoders to enforce the correct encoding rule. We recommend using this when you define an object representing an ASN.1 CHOICE.

Distinguished Encoding Rules Encoding

With the provision for structured primitive types, unordered sets and the like, there might be a few ways of encoding a given ASN.1 structure. The Distinguished Encoding Rules (DER) are designed to make sure that identical ASN.1 data will always result in identical encodings.

This is particularly important for signed or MAC'd data as the ability to verify signed ASN.1 data relies on the ability of the party doing the verification to be able to accurately recreate correctly the encoding of the data that was signed.

To make sure this is always the case, DER adds the following restrictions to BER:

- Only definite-length encoding is allowed.
- Only the structured types SEQUENCE, SET, IMPLICIT tagged SEQUENCE or SET, and EXPLICIT tagged objects may use constructed definite-length.
- The length encodings must always be encoded in the minimum number of bytes (no leading zeroes allowed).
- Fields that are set to the DEFAULT value are not included in the encoding.
- The objects contained in a SET are sorted by adjusted encoded value (with zeroes are appended where necessary to make all encodings the same length) before encoding.

Note: these last two restrictions apply for CER as well.

While the DER SET is sorted, remember ASN.1 SET itself is unordered. Other than for DER encoding purposes do not rely on a particular order in a SET.

Octet Encoding Rules

In 2014, a new set of encoding rules were introduced which are now starting to make themselves felt, mainly in standards where the size of the final encoding is regarded as important. The new set, the Octet Encoding Rules (OER), is defined in X.696 [33] and the rules work side by side with BER and DER. The foundation rules are the Basic Octet Encoding Rules (BOER) and the DER equivalent is the Canonical Octet Encoding Rules (COER). All OER types are byte aligned, but not all of them include length tags as the length tag is left out where the ASN.1 definition of the type already provides the ability to determine length.

OER expands the list of ASN.1 types to include fixed length unsigned integers (UInt) of varying bit-lengths as well fixed length signed integers (Int). String types can also be fixed length, for example OCTET STRING types can be fixed length, in which case the OCTET STRING encodes directly as its value octets, or they can be variable length, in which case the regular length tag is included at the start. Finally, when a SEQUENCE has optional items in it, a presence bitmap, rather than a length is encoded at the start of the sequence which flags which of the optional fields are present.

Bouncy Castle does not currently support OER, but watch this space, we are pretty sure it is coming!

Other Encoding Rules

Other than the “recently” arrived OER rules, there are other encoding rules available for ASN.1 as well. The most likely one to run into in this line of work is the Canonical Encoding Rules (CER) which is also a subset of BER but places restrictions on how big the components of indefinite length encodings can be. In general, it is worth being aware of CER, as it gives you a good idea of the limits people will implement against. In ASN.1, and indeed Java, it is technically possible to construct a valid indefinite length constructed OCTET STRING made up of substrings that are 2^{31} octets in length - you may have a very hard time finding someone that could parse such a creation though.

The last two encodings (at the time of writing) are the Packed Encoding Rules (PER) and the XML Encoding Rules (XER). If you are interested in more information on these please see the relevant standards X.961 [31] and X.963 [32].

Basic Guidance

One of the difficulties of working with a protocol layer like ASN.1 in an object-orientated language is it is sometimes not clear where it is safe to be dealing with an ASN.1 structure such as an AlgorithmIdentifier as the object `org.bouncycastle.asn1.x509.AlgorithmIdentifier` or as the structure in its primitive composition as a SEQUENCE (an `org.bouncycastle.asn1.ASN1Sequence` in Java parlance). Casting does not actually work very well in this case - objects that come “off the wire” are generally primitive compositions, but objects passed down from high level functions in an application tend to be actual objects referring to types in specific standard modules. A cast may easily cause a

ClassCastException if a method is used in an unexpected way. Simple constructors could be used but would probably result in many objects being created unnecessarily and additional complexity.

In Bouncy Castle this problem dealt with by using static getInstance() methods which make casting unnecessary. For example AlgorithmIdentifier.getInstance() can be passed either an AlgorithmIdentifier object or an ASN1Sequence object that represents an AlgorithmIdentifier and will always return an AlgorithmIdentifier or null if a null value was passed in. Likewise ASN1Sequence.getInstance() can be passed an AlgorithmIdentifier object or an ASN1Sequence object and will always return an ASN1Sequence or null if a null value was passed in[30].

Defining Your Own Objects

Normally if you are defining an ASN.1 based object you should define a class extending off org.bouncycastle.asn1.AS This will provide equals() and hashCode() definitions as well as require the definition of a toASN1Primitive() method in your extending class. After that if you wish to follow the existing pattern (which we strongly recommend) you should define a static getInstance() method which is capable of accepting either the class type, whatever is returned by toASN1Primitive(), or null. Once you have done that the rest of the class will be almost self-writing - with constructors falling out from the types making up the object.

The example code that follows provides a definition of a class for the ASN.1 structure:

```
SimpleStructure ::= SEQUENCE {
    version INTEGER DEFAULT 0,
    created GeneralizedTime,
    data OCTET STRING,
    comment [0] UTF8String OPTIONAL
}
```

The class is complete, but goes for a couple of pages. Have a look through the code first and then read the commentary that follows it.

```
/**
 * Implementation of SimpleStructure - an example ASN.1 object (tagging
 * IMPLICIT).
 * <pre>
 *     SimpleStructure ::= SEQUENCE {
 *         version INTEGER DEFAULT 0,
 *         created GeneralizedTime,
 *         data OCTET STRING,
 *         comment [0] UTF8String OPTIONAL
 *     }
```

[30] The getInstance() methods can also be passed a byte array containing the DER/BER encoding of a single object.

```java
 * </pre>
 */
public class SimpleStructure
    extends ASN1Object
{
    private final BigInteger version;
    private final Date created;
    private final byte[] data;

    private String comment;

    /**
     * Convert, or cast, the passed in object into a SimpleStructure
     * as appropriate.
     *
     * @param obj the object of interest.
     * @return a SimpleStructure
     */
    public static SimpleStructure getInstance(
        Object  obj)
    {
        if (obj instanceof SimpleStructure)
        {
            return (SimpleStructure)obj;
        }
        else if (obj != null)
        {
            return new SimpleStructure(ASN1Sequence.getInstance(obj));
        }

        return null;
    }

    /**
     * Create a structure with a default version.
     *
     * @param created creation date.
     * @param data encoded data to contain.
     */
    public SimpleStructure(Date created, byte[] data)
    {
        this(0, created, data, null);
    }

    /**
     * Create a structure with a default version and the optional comment.
     *
     * @param created creation date.
     * @param data encoded data to contain.
     * @param comment the comment to use.
     */
    public SimpleStructure(Date created, byte[] data, String comment)
    {
        this(0, created, data, comment);
    }

    /**
     * Create a structure with a specific version and the optional comment.
     *
     * @param version the version number to use.
     * @param created creation date.
     * @param data encoded data to contain.
     * @param comment the comment to use.
     */
    public SimpleStructure(int version, Date created,
                           byte[] data, String comment)
    {
        this.version = BigInteger.valueOf(version);
        this.created = new Date(created.getTime());
        this.data = Arrays.clone(data);

        if (comment != null)
        {
            this.comment = comment;
        }
        else
        {
            this.comment = null;
        }
    }

    // Note: private constructor for sequence - we want users to get
    // into the habit of using getInstance(). It's safer!
    private SimpleStructure(ASN1Sequence seq)
    {
        int index = 0;

        if (seq.getObjectAt(0) instanceof ASN1Integer)
        {
            this.version = ASN1Integer.getInstance(
                                 seq.getObjectAt(0)).getValue();
            index++;
        }
        else
        {
            this.version = BigInteger.ZERO;
        }

        try
        {
            this.created = ASN1GeneralizedTime.getInstance(
                                seq.getObjectAt(index++)).getDate();
        }
        catch (ParseException e)
        {
            throw new IllegalArgumentException(
                "exception parsing created: " + e.getMessage(), e);
        }

        this.data = Arrays.clone(
            ASN1OctetString.getInstance(seq.getObjectAt(index++)).getOctets());

        for (int i = index; i != seq.size(); i++)
        {
            ASN1TaggedObject t = ASN1TaggedObject.getInstance(
                                                     seq.getObjectAt(i));

            if (t.getTagNo() == 0)
            {
                comment = DERUTF8String.getInstance(t, false)
                                                            .getString();
            }
        }
    }

    public BigInteger getVersion()
    {
        return version;
    }
```

```java

    public Date getCreated()
        throws ParseException
    {
        return new Date(created.getTime());
    }

    public byte[] getData()
    {
        return Arrays.clone(data);
    }

    public String getComment()
    {
        return comment;
    }

    /**
     * Produce a DER representation of the object.
     *
     * @return an ASN1Primitive made up of DER primitives.
     */
    @Override
    public ASN1Primitive toASN1Primitive()
    {
        ASN1EncodableVector v = new ASN1EncodableVector();

        // DER encoding rules specify that fields with
        // the value of their specified DEFAULT are left out
        // of the encoding
        if (!version.equals(BigInteger.ZERO))
        {
             v.add(new ASN1Integer(version));
        }

        v.add(new DERGeneralizedTime(created));
        v.add(new DEROctetString(data));

        if (comment != null)
        {
            v.add(new DERTaggedObject(false, 0, new DERUTF8String(comment)));
        }

        return new DERSequence(v);
    }
}
```

As you can see the class follows the standard pattern. It extends `ASN1Object`, provides a `getInstance()` method, and also takes into account DER encoding rules (the version is only encoded if it is not the default value). The `getInstance()` method is able to outsource most of its work to `ASN1Sequence.getInstance()` and the constructor taking an `ASN1Sequence` is marked as private so that any attempts to use it have to go through the `SimpleStructure.getInstance()` method.

The other thing to note is that the class is immutable. The main thing to note here is that unlike almost all ASN.1 objects, `ASN1OctetString` are not immutable - largely for (sigh) performance reasons. This is reason why the Arrays.clone() method is being called on the `byte[]` returned by the `getOctets()` method. Ideally as `ASN1Object` provides both `equals()` and `hashCode()` an `ASN1Object` should be immutable!

Appendix B: The Bouncy Castle Kotlin API

Periodically we get asked if there is a way of using Bouncy Castle for writing scripts, or at least something that is a nearest equivalent. The Bouncy Castle Kotlin API takes advantage of Kotlin's Domain Specific Language (DSL) capabilities to provide support for generating things like certificates and certification requests as well as providing additional methods for `java.io.OutputStreamWriter` to produce PEM files.

A Note on Packaging

While it was quite tempting to use org.bouncykastle, in the interests of not cluttering up the Internet further, we ended up using org.bouncycastle.kcrypto as the parent package for Kotlin specific cryptography classes and org.bouncycastle.kutil as the parent package for general utilities.

The Kotlin API is available at https://github.com/bcgit/bc-kotlin We will look at the two "classic" examples of use here, but there is also an examples directory included with the distribution that shows other use cases.

Generating a PKCS#10 Certification Request

The following Kotlin DSL creates an RSA key pair and then generates a PKCS#10 Certification Request from it. The signature used in the request is equivalent to what the JCA would provide if you asked for "SHA256withRSA". At the end the script writes out both the private key and the PKCS#10 request as PEM files.

```
fun main() {

    using(new BouncyCastleProvider())

    var kp = signingKeyPair {
        rsa {
            keySize = 2048
        }
    }

    val name = x500Name {
        rdn(BCStyle.C, "AU")
        rdn(BCStyle.O, "The Legion of the Bouncy Castle")
        rdn(BCStyle.L, "Melbourne")
        rdn(BCStyle.CN, "Eric H. Echidna")
        rdn(BCStyle.EmailAddress, "feedback-crypto@bouncycastle.org")
    }

    val extensions = extensions {
        critical(extension {
            extOid = Extension.keyUsage
            extValue = KeyUsage(KeyUsage.keyCertSign or KeyUsage.cRLSign)
        })
    }

    var pkcs10 = pkcs10Request {
        subject = name
        subjectKey = kp.verificationKey
        attributes = attributes {
             attribute {
                 attrType = PKCSObjectIdentifiers.pkcs_9_at_extensionRequest
                 attrValue = extensions
             }
        }
        signature {
            PKCS1v1dot5 with sha256 using kp.signingKey
        }
    }

    OutputStreamWriter(System.out).writePEMObject(kp.signingKey)

    OutputStreamWriter(System.out).writePEMObject(pkcs10)
}
```

Note the use of the `attributes` block in the certification request. While the block is optional, in the above example we are using it to request that the CA included a KeyUsage extension in the certificate it issues. You can find more details on the PKCS#10 request in Chapter 12.

One of the other interesting things to note about the script is that the signature algorithm is not specified using a string. The Kotlin API tries to take full advantage of the Kotlin language to make sure that everything is checked at compile time. The result of this is that if you need to disable an algorithm, and be sure you have, you can just remove the BC Kotlin API's definition of it and any scripts using the disabled algorithm will not even compile.

Generating a Certificate

The other task that is often scripted is the generation of certificates. The following example generates a self-signed certificate adding a few extensions all marked as critical.

```
fun main() {

    using(new BouncyCastleProvider())

    var expDate = Date(System.currentTimeMillis() + 365L * 24 * 60 * 60 * 1000)

    var caKp = signingKeyPair {
        rsa {
            keySize = 2048
        }
    }

    val caName = x500Name {
        rdn(BCStyle.C, "AU")
        rdn(BCStyle.O, "The Legion of the Bouncy Castle")
        rdn(BCStyle.L, "Melbourne")
        rdn(BCStyle.CN, "Eric's CA")
    }

    var caExtensions = extensions {
        critical(extension {
            extOid = Extension.basicConstraints
            extValue = BasicConstraints(true)
        })
        critical(extension {
            extOid = Extension.keyUsage
            extValue = KeyUsage(KeyUsage.cRLSign or KeyUsage.keyCertSign)
        })
        critical(subjectKeyIdentifierExtension {
            subjectKey = caKp.verificationKey
        })
    }

    var caCert = certificate {
        issuer = caName

        serialNumber = BigInteger.valueOf(1)

        notAfter = expDate
        subject = caName
        subjectPublicKey = caKp.verificationKey

        extensions = caExtensions

        signature {
            PKCS1v1dot5 with sha256 using caKp.signingKey
        }
    }

    OutputStreamWriter(System.out).writePEMObject(caKp.signingKey)

    OutputStreamWriter(System.out).writePEMObject(caCert)
}
```

As with the PKCS#10 example the certificate in question will be signed using SHA256withRSA. Differences wise, the fields in a certificate setup are obviously different, but the feeling of the API is still the same.

Of course, you might want to encrypt the new private key with a password. The `writePEMObject()` override for `OutputStreamWriter` can recognise a range of objects for PEM encoding. So, another way (clearly, given the password, not the greatest) of saving the private key is given below:

```
    var encCaKey = encryptedPrivateKey {
        privateKey = caKp.signingKey
        encryption {
            AESKWP using SCRYPT {
                saltLength = 20
                costParameter = 1048576
                blockSize = 8
                parallelization = 1
                keySize = 256
            } with "Test".toCharArray()
        }
    }

    OutputStreamWriter(System.out).writePEMObject(encCaKey)
```

This code fragment will encrypt the private key using the AES key wrap algorithm for encryption and the scrypt algorithm for deriving the secret key used from the password. The result of this is you should receive a blob with the PEM header:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
```

rather than the previous one for a general RSA private key:

```
-----BEGIN RSA PRIVATE KEY-----
```

These examples only touch on some of the things that can be done with the Kotlin API. If you are curious to see what else can be done, don’t forget to check out the examples directory in the BC Kotlin distribution.

Appendix C: Algorithms provided by the Bouncy Castle Providers

Ciphers (Block)

Algorithm	Key Size	BC General	BC FIPS
AES	128, 192, 256	X	X
Blowfish	0 .. 448	X	X
Camellia	128, 192, 256	X	X
CAST-5	0 .. 128	X	X
CAST-6	0 .. 256	X	X
DES	56	X	X
DSTU7624	128, 256, 512	X	
GOST 28147	256	X	X
GOST 3412-2015	256	X	
IDEA	128	X	X
RC2	0 .. 1024	X	X
Rijndael	0 .. 256	X	X
SEED	128	X	X
Serpent	128, 192, 256	X	X
Shacal2	512	X	
Skipjack	0 .. 128	X	
SM4	128	X	
TEA	128	X	
Threefish	256, 512, 1024	X	
Triple-DES	112, 168	X	X
Twofish	128, 192, 256	X	X
XTEA	128	X	

Ciphers (Public Key)

Algorithm	BC General	BC FIPS
RSA	X	X
ElGamal	X	X

Ciphers (KEMS)

Algorithm	BC General	BC FIPS
BIKE	X	
Classic McEliece	X	
FrodoKEM	X	
HQC	X	
Kyber	X	
NTRU	X	
NTRU Prime	X	
RSA-KTS-KEM-KWS		X
SABER	X	

Ciphers (Stream)

Algorithm	Key Size	BC General	BC FIPS
ARC4	40 .. 2048	X	X
HC128	128	X	
HC256	256	X	
ChaCha	128, 256	X	
Salsa20	128, 256	X	
XSalsa20	256	X	
ISAAC	32 .. 8192	X	
VMPC	8 .. 6144	X	
Grain-V1	80	X	
Grain-128	128	X	

Key Agreement Algorithms

Algorithm	BC General	BC FIPS
DH	X	X
DHU	X	
MQV	X	X
ECDH	X	X
ECCDH	X	X
ECCDHU	X	
ECMQV	X	X
XDH	X	X

Message Authentication Codes (MACs)

Algorithm	Output Size	BC General	BC FIPS
CBC-MAC	blocksize/2 unless specified	X	X
CFB-MAC	blocksize/2, in CFB 8 mode, unless specified	X	X
CMAC	24 to cipher block size bits	X	X
GMAC	32 to 128 bits	X	X
GOST28147-MAC	32 bits	X	X
ISO9797Alg3-MAC	multiple of 8 bits up to underlying cipher size	X	
HMac	digest length	X	X
DSTU7564	256, 384, 512 bits	X	
DSTU7624	128, 256, 512 bits	X	
Poly1305	128 bits	X	
SkeinMac	any byte length	X	
SipHash	64 bits	X	

Message Digests and Expandable Output Functions

Algorithm	Output Size	BC General	BC FIPS
Blake2s	128, 160, 224, 256	X	
Blake2b	224, 256, 384, 512	X	
Blake3	224, 256, 384, 512	X	
cSHAKE	Any	X	X
DSTU-7564	256, 384, 512	X	
GOST-3411	256	X	X
GOST-3411-2012-256	256	X	
GOST-3411-2012-512	512	X	
Haraka	256, 512	X	
Kangaroo	Any	X	
Keccak	224, 256, 288, 384, 512	X	
KMAC	Any	X	X
MD2	128	X	
MD4	128	X	
MD5	128	X	X
ParallelHash	Any	X	X
RipeMD-128	128	X	X
RipeMD-160	160	X	X
RipeMD-256	256	X	X
RipeMD-320	320	X	X
SHA-1	160	X	X
SHA-224	224	X	X
SHA-256	256	X	X
SHA-384	384	X	X
SHA-512	512	X	X

Message Digests and Expandable Output Functions

Algorithm	Output Size	BC General	BC FIPS
SHA-3	224, 256, 384, 512	X	X
SHAKE-128	Any	X	X
SHAKE-256	Any	X	X
Skein	Any	X	
SM3	256	X	
TupleHash	Any	X	X
Tiger	192	X	X
Whirlpool	512	X	X

Signature Algorithms

Algorithm	BC General	BC FIPS
Dilithium	X	
DSA	X	X
ECDSA	X	X
EdDSA	X	X
Falcon	X	
GeMSS	X	
LMS	X	X
Picnic	X	
Rainbow	X	
RSA	X	X
RSA-PSS	X	X
SPHINCS+	X	
XMSS	X	

Acronyms and Definitions

Acronym	Definition
AES	Advanced Encryption Standard
ASN.1	Abstract Syntax Notation 1
ACVP	Automated Cryptographic Validation Protocol
CAVP	Cryptographic Algorithm Validation Program
CBC	Cipher-Block Chaining
CCM	Counter with CBC-MAC
CFB	Cipher Feedback Mode
CMAC	Cipher-based Message Authentication Code
CMVP	Cryptographic Module Validation Program
CRL	Certificate Revocation List
CS	Cipher text Stealing
CTS	Cipher Text Stealing
CTR	CounTeR-mode
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
DRBG	Deterministic Random Bit Generator
EC	Elliptic Curve
ECB	Electronic CodeBook
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Authority
FIPS	Federal Information Processing Standards
GCM	Galois/Counter Mode
GOST	Gosudarstvennyi Standard Soyuza SSR/Government Standard of the Union of Soviet Socialist Republics
HMAC	key-Hashed Message Authentication Code
HSM	Hardware Security Module
IV	Initialization Vector
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KAS	Key Agreement Scheme
KAT	Known Answer Test
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
KW	Key Wrap
KWP	Key Wrap with Padding
MAC	Message Authentication Code
NDRNG	NonDeterministic Random Number Generator

Acronym	Definition
NIST	National Institute of Standards and Technology
OCB	Offset CodeBook mode
OCSP	Online Certificate Status Protocol
OFB	Output FeedBack mode
OID	Object IDentifer
PBKDF	Password Based Key Derivation Function
PGP	Pretty Good Privacy
PKCS	Public Key Cryptography Standards
RNG	Random Number Generator
SEC	Standards for Efficient Cryptography
SPI	Service Provider Interface
SSL	Secure Sockets Layer
TLS	Transport Layer Security
XOF	eXpandable Output Function

Bibliography and Further Reading

[1] Unknown. (1994). DIGITAL SIGNATURE STANDARD (DSS). [online] NIST. Retrieved from http://www.umich.edu/~x509/ssleay/fip186/fip186.htm [Accessed 25 Apr. 2018].

[2] E. Barker (NIST). (2013). DIGITAL SIGNATURE STANDARD (DSS). [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/fips/186/4/final [Accessed 25 Apr. 2018].

[3] Various. (2016). Public Comments Received on NISTIR 8105 - Draft Report on Post-Quantum Cryptography. [online] NIST. Retrieved from https://csrc.nist.gov/CSRC/media/Publications/nistir/8105/final/documents/nistir-8105-public-comments-mar2016.pdf [Accessed 10 Jan. 2018].

[4] M. Dworkin (NIST). (2001). SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-38a/final [Accessed 10 Jan. 2018].

[5] M. Dworkin (NIST). (2010). SP 800-38A Addendum: Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-38a/addendum/final [Accessed 10 Jan. 2018].

[6] M. Dworkin (NIST). (2004). SP 800-38C: Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-38c/final [Accessed 10 Jan. 2018].

[7] M. Dworkin (NIST). (2007). SP 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-38d/final [Accessed 10 Jan. 2018].

[8] M. Dworkin (NIST). (2012). SP 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-38f/final [Accessed 10 Jan. 2018].

[9] E. Barker (NIST), L. Chen (NIST), A. Roginsky (NIST), A. Vassilev (NIST), R. Davis (NSA). (2018). SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-56a/rev-3/final [Accessed 06 Jun. 2018].

[10] E. Barker (NIST), L. Chen (NIST), D. Moody (NIST). (2014). SP 800-56B Rev. 1: Recommendation for Pair-Wise Key-Establishment Schemes Using Integer Factorization Cryptography. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-56b/rev-1/final [Accessed 19 Apr. 2018].

[11] E. Barker (NIST), L. Chen (NIST), R. Davis (NIST). (2020). SP 800-56C Rev. 2: Recommendation for Key-Derivation Methods in Key-Establishment Schemes. [online] NIST. Retrieved from

https://csrc.nist.gov/publications/detail/sp/800-56c/rev-1/final [Accessed 29 Jan. 2022].

[12] E. Barker (NIST). (2016). SP 800-57 Part 1 Rev. 4: Recommendation for Key Management, Part 1: General. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final [Accessed 19 Apr. 2018].

[13] E. Barker (NIST), J. Kelsey (NIST). (2015). SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final [Accessed 14 Jan. 2018].

[14] M. Turan (NIST), E. Barker (NIST), J. Kelsey (NIST), K. McKay (NIST), M. Baish (NSA), M. Boyle (NSA). (2018). SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-90b/final [Accessed 14 Jan. 2018].

[15] M. Turan (NIST), E. Barker (NIST), W. Burr (NIST), L. Chen (NIST). (2010). SP 800-132: Recommendation for Password-Based Key Derivation: Part 1: Storage Applications. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-132/final [Accessed 01 Apr. 2018].

[16] J. Kelsey (NIST), S. Chang (NIST), R. Perlner (NIST). (2016). SP 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-185/final [Accessed 29 Jan. 2022].

[17] D. Cooper (NIST), D. Apon (NIST), Q. Dang (NIST), M. Davidson (NIST), M. Dworkin (NIST), C. Miller (NIST). (2010). SP 800-208 (Draft): Recommendation for Stateful Hash-Based Signature Schemes. [online] NIST. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-208/draft [Accessed 01 Apr. 2018].

[18] Information Technology Laboratory, NIST. (2024). FIPS 203 Module-Lattice-Based Key-Encapsulation Mechanism Standard. [online] NIST. Retrieved from https://csrc.nist.gov/pubs/fips/204/final [Accessed 16 Jul. 2025].

[19] Information Technology Laboratory, NIST. (2024). FIPS 204 Module-Lattice-Based Digital Signature Standard. [online] NIST. Retrieved from https://csrc.nist.gov/pubs/fips/204/final [Accessed 16 Jul. 2025].

[20] Information Technology Laboratory, NIST. (2024). FIPS 205 Stateless Hash-Based Digital Signature Standard. [online] NIST. Retrieved from https://csrc.nist.gov/pubs/fips/205/final [Accessed 16 Jul. 2025].

[21] ISO. (2010). Information technology - Security techniques - Digital signature schemes giving message recovery - Part 2: Integer factorization based mechanisms.

[22] ANSI. (1998). Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA).

[23] ITU-T, ISO, and IEC. (2016). X.500: Information technology - Open Systems Interconnection -The Directory: Overview of concepts, models and services.

[24] ITU-T, ISO, and IEC. (2016). X.501: Information technology - Open Systems Interconnection -The

Directory: Models.

[25] ITU-T, ISO, and IEC. (2016). X.509: Information technology - Open Systems Interconnection -The Directory: Public-key and attribute certificate frameworks.

[26] ITU-T, ISO, and IEC. (2015). X.680: Information technology - Abstract Notation One (ASN.1): Specification of Basic Notation.

[27] ITU-T, ISO, and IEC. (2015). X.681: Information technology - Abstract Notation One (ASN.1): Information Object Specification.

[28] ITU-T, ISO, and IEC. (2015). X.682: Information technology - Abstract Notation One (ASN.1): Constraint Specification.

[29] ITU-T, ISO, and IEC. (2015). X.683: Information technology - Abstract Notation One (ASN.1): Parameterization of ASN.1.

[30] ITU-T, ISO, and IEC. (2015). X.690: Information technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).

[31] ITU-T, ISO, and IEC. (2015). X.691: Information technology - ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER).

[32] ITU-T, ISO, and IEC. (2015). X.693: Information technology - ASN.1 Encoding Rules: Specification of XML Encoding Rules (XER).

[33] ANSI. (2014). X.696: Information Technology - Specification of Octet Encoding Rules (OER).

[34] ANSI. (2013). Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography.

[35] ANSI. (2005). Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).

[36] ANSI. (2011). Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography.

[37] P. Gutmann. (2000). X.509 Style Guide. [online] Retrieved from https://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt [Accessed 29 Aug. 2017].

[38] Legion of the Bouncy Castle Inc.. (2023). BC-FJA (Bouncy Castle FIPS Java API User Guide). [online] Retrieved from https://downloads.bouncycastle.org/fips-java/BC-FJA-UserGuide-1.0.2.pdf [Accessed 14 Jan. 2023].

[39] Legion of the Bouncy Castle Inc.. (2023). Java (D)TLS API and JSSE Provider User Guide. [online] Retrieved from https://downloads.bouncycastle.org/fips-java/BC-FJA-(D)TLSUserGuide-1.0.15.pdf [Accessed 14 Jan. 2023].

[40] A. Shamir. (1979). How to Share a Secret. Communications of the ACM, 22 (11)

[41] Research in Motion Limited. (2014). Disclosure of Patent Information Relating to "RFC 5753 - Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)". [online]

Certicom. Retrieved from http://www.ietf.org/ietf-ftp/IPR/certicom-ipr-rfc-5753.pdf [Accessed 06 May. 2018].

[42] S. Shen (Ed.), X. Lee (Ed.). (2014). RFC DRAFT - SM2 Digital Signature Algorithm (draft-shen-sm2-ecdsa-02). [online] IETF. Retrieved from https://tools.ietf.org/html/draft-shen-sm2-ecdsa-02 [Accessed 06 May. 2018].

[43] S. Shen, X. Lee, R. Tse, W. Wong, Y. Yang. (2017). RFC DRAFT - The SM3 Cryptographic Hash Function (draft-oscca-cfrg-sm3-02). [online] IETF. Retrieved from https://tools.ietf.org/html/draft-shen-sm2-ecdsa-02 [Accessed 06 May. 2018].

[44] D. Van Geest, S. Fluhrer. (2019). RFC DRAFT - Algorithm Identifiers for HSS and XMSS for Use in the Internet X.509 Public Key Infrastructure (draft-vangeest-x509-hash-sigs-03). [online] IETF. Retrieved from https://tools.ietf.org/html/draft-vangeest-x509-hash-sigs-03 [Accessed 06 Sep. 2019].

[45] R. Baldwin, R. Rivest. (1996). RFC 2040 - The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc2040 [Accessed 06 Jan. 2018].

[46] E. Rescorla. (1999). RFC 2631 - Diffie-Hellman Key Agreement Method. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc2631 [Accessed 10 Jun. 2018].

[47] B. Kaliski. (1998). RFC 2315 - PKCS #7: Cryptographic Message Syntax Version 1.5. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc2315 [Accessed 06 Sep. 2017].

[48] H. Prafullchandra, J. Schaad. (2000). RFC 2875 - Diffie-Hellman Proof-of-Possession Algorithms. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc2875 [Accessed 21 Jun. 2018].

[49] M. Nystrom, B. Kaliski. (2001). RFC 2986 - PKCS #10: Certification Request Syntax Specification Version 1.7. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc2986 [Accessed 06 Apr. 2018].

[50] D. Pinkas, J. Ross, N. Pope. (2001). RFC 3126 - Electronic Signature Formats for long term electronic signatures. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc3126 [Accessed 11 May. 2019].

[51] C. Adams, P. Cain, D. Pinkas, R. Zuccherato. (2001). RFC 3161 - Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc3161 [Accessed 21 Feb. 2019].

[52] P. Gutmann. (2001). RFC 3211 - Password-based Encryption for CMS. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc3211 [Accessed 06 Apr. 2018].

[53] R. Housley. (2001). RFC 3217 - Triple-DES and RC2 Key Wrapping. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc3217 [Accessed 06 Apr. 2018].

[54] R. Housley. (2002). RFC 3370 - Cryptographic Message Syntax (CMS) Algorithms. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc3370 [Accessed 06 Apr. 2018].

[55] D. Pinkas, N. Pope, J. Ross. (2003). RFC 3628 - Policy Requirements for Time-Stamping Authorities

(TSAs). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc3628 [Accessed 01 May. 2019].

[56] R. Housley. (2004). RFC 3686 - Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc3686 [Accessed 06 Sep. 2017].

[57] D. Eastlake, 3rd, J. Schiller, S. Crocker. (2005). RFC 3686 - Randomness Requirements for Security. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4086 [Accessed 06 Sep. 2017].

[58] C. Adams, S. Farrell, T. Kause, T. Mononen. (2005). RFC 4210 - Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4210 [Accessed 25 Jan. 2022].

[59] J. Schaad. (2005). RFC 4211 - Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4211 [Accessed 19 Apr. 2018].

[60] P. Eronen, H. Tschofenig. (2005). RFC 4279 - Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4279 [Accessed 11 Jun. 2022].

[61] M. Friedl, N. Provos, W. Simpson. (2006). RFC 4419 - Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4419 [Accessed 11 Jun. 2018].

[62] S. Leontiev (Ed.), D. Shefanovski (Ed.). (2006). RFC 4491 - Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4491 [Accessed 06 May. 2018].

[63] A. Sciberras. (2006). RFC 4519 - Lightweight Directory Access Protocol (LDAP): Schema for User Applications. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4519 [Accessed 06 Feb. 2018].

[64] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, R. Thayer. (2007). RFC 4880 - OpenPGP Message Format. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc4880 [Accessed 14 Sep. 2019].

[65] D. Taylor, T. Wu, N. Mavrogiannopolous, and T. Perrin. (2007). RFC 5054 - Using the Secure Remote Password (SRP) Protocol for TLS Authentication. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5054 [Accessed 10 Jan. 2022].

[66] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. (2007). RFC 5084 - Using AES-CCM and AES-GCM Authenticated Encryption in the Cryptographic Message Syntax (CMS). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5084 [Accessed 10 Apr. 2018].

[67] M. Lepinski, S. Kent. (2008). RFC 5114 - Additional Diffie-Hellman Groups for Use with IETF Standards. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5114 [Accessed 10 Jun. 2018].

[68] D. Pinkas, N. Pope, J. Ross. (2008). RFC 5126 - CMS Advanced Electronic Signatures (CAdES).

[online] IETF. Retrieved from https://tools.ietf.org/html/rfc5126 [Accessed 11 Jun. 2019].

[69] J. Schaad and M. Myers. (2008). RFC 5272 - Certificate Management over CMS (CMC). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5272 [Accessed 29 Aug. 2020].

[70] J. Schaad and M. Myers. (2008). RFC 5273 - Certificate Management over CMS (CMC): Transport Protocols. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5273 [Accessed 29 Aug. 2020].

[71] J. Schaad and M. Myers. (2008). RFC 5274 - Certificate Management Messages over CMS (CMC): Compliance Requirements. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5274 [Accessed 29 Aug. 2020].

[72] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. (2008). RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5280 [Accessed 29 Jul. 2017].

[73] A. Santoni. (2010). RFC 5544 - Syntax for Binding Documents with Time-Stamps. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5544 [Accessed 12 May. 2019].

[74] R. Houseley. (2009). RFC 5652 - Cryptographic Message Syntax (CMS). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5652 [Accessed 10 Jun. 2018].

[75] B. Ramsdell, and S. Turner. (2010). RFC 5751 - Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5751 [Accessed 24 Feb. 2019].

[76] S. Turner, D. Brown. (2010). RFC 5753 - Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5753 [Accessed 06 Jun. 2018].

[77] S. Farrell, R. Houseley, and S. Turner. (2010). RFC 5755 - An Internet Attribute Certificate Profile for Authorization. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5755 [Accessed 20 Sep. 2017].

[78] S. Santesson, N. Pope. (2010). RFC 5816 - ESSCertIDv2 Update for RFC 3161. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5816 [Accessed 01 May. 2019].

[79] V. Dolmatov (Ed.). (2010). RFC 5832 - GOST R 34.10-2001: Digital Signature Algorithmn. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5832 [Accessed 06 May. 2018].

[80] J. Altman, N. Williams, and L. Zhu. (2010). RFC 5929 - Channel Bindings for TLS. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5929 [Accessed 29 Jan. 2022].

[81] S. Turner. (2001). RFC 5967 - The application/pkcs10 Media Type. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5967 [Accessed 06 Apr. 2018].

[82] J. Randall, B. Kaliski, J. Brainard, S. Turner. (2010). RFC 5990 - Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc5990 [Accessed 26 May. 2018].

[83] J. Schaad. (2011). RFC 6211 - Cryptographic Message Syntax (CMS) Algorithm Identifier Protection

Attribute. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6211 [Accessed 10 Nov. 2018].

[84] J. Schaad. (2012). RFC 6402 - Certificate Management over CMS (CMC) Updates. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6402 [Accessed 29 Aug. 2020].

[85] A. Jivsov. (2012). RFC 6637 - Elliptic Curve Cryptography (ECC) in OpenPGP. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6637 [Accessed 10 Feb. 2020].

[86] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams. (2013). RFC 6960 - X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6960 [Accessed 22 Jan. 2018].

[87] Y. Pettersen. (2013). RFC 6961- The Transport Layer Security (TLS) Multiple Certificate Status Request Extension. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6961 [Accessed 22 Jan. 2018].

[88] b. Laurie, A. Langley, E. Kasper. (2013). RFC 6962 - Certificate Transparency. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6962 [Accessed 28 Apr. 2019].

[89] T. Pornin. (2013). RFC 6979 - Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6979 [Accessed 11 Apr. 2018].

[90] V. Dolmatov (Ed.), A. Degtyarev. (2013). RFC 6986 - GOST R 34.11-2012: Hash Function. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc6986 [Accessed 06 May. 2018].

[91] M. Pritikin (Ed.), P. Yee (Ed.), D. Harkins, (Ed.). (2013). RFC 7030 - Enrollment over Secure Transport. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7030 [Accessed 11 Apr. 2018].

[92] V. Dolmatov (Ed.), A. Degtyarev. (2013). RFC 7091 - GOST R 34.10-2012: Digital Signature Algorithm. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7091 [Accessed 11 Apr. 2018].

[93] T. Krovetz, P. Rogaway. (2014). RFC 7253 - The OCB Authenticated-Encryption Algorithm. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7253 [Accessed 11 Apr. 2018].

[94] T. Krovetz, P. Rogaway. (2014). RFC 7292 - PKCS #12: Personal Information Exchange Syntax v1.1. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7292 [Accessed 30 Aug. 2018].

[95] Y. Nir, A. Langley. (2015). RFC 7539 - ChaCha20 and Poly1305 for IETF Protocols. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7539 [Accessed 22 Jan. 2018].

[96] A. Bakker, R. Petrocco, V. Grishchenko. (2015). RFC 7574 - Peer-to-Peer Streaming Peer Protocol (PPSPP). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7574 [Accessed 28 Apr. 2019].

[97] C. Percival, S. Josefsson. (2016). RFC 7914 - The scrypt Password-Based Key Derivation Function. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7914 [Accessed 06 Sep. 2017].

[98] D. Gillmor. (2016). RFC 7919 - Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc7919 [Accessed 10 Jun. 2018].

[99] K. Moriarty (Ed.), B. Kaliski, J. Jonsson, A. Rusch. (2016). RFC 8017 - PKCS #1: RSA Cryptography

Specifications Version 2.2. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc8017 [Accessed 18 May. 2018].

[100] K. Moriarty (Ed.), B. Kaliski, A. Rusch. (2017). RFC 8018 - PKCS #5: Password-Based Cryptography Specification Version 2.1. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc8018 [Accessed 06 Sep. 2017].

[101] S. Josefsson, I. Liusvaara. (2017). RFC 8032 - Edwards-Curve Digital Signature Algorithm (EdDSA). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc8032 [Accessed 06 Sep. 2017].

[102] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, A. Mohaisen. (2017). RFC 8391 - XMSS: eXtended Merkle Signature Scheme. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc8391 [Accessed 12 Jun. 2018].

[103] D. McGrew, M. Curcio, S. Fluhrer. (2019). RFC 8554 - Leighton-Micali Hash-Based Signatures. [online] IETF. Retrieved from https://tools.ietf.org/html/rfc8554 [Accessed 06 Aug. 2020].

[104] R. Housley. (2020). RFC 8708 - Use of the HSS/LMS Hash-Based Signature Algorithm in the Cryptographic Message Syntax (CMS). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc8708 [Accessed 06 Aug. 2020].

[105] R. Housley. (2020). RFC 8778 - Use of the HSS/LMS Hash-Based Signature Algorithm with CBOR Object Signing and Encryption (COSE). [online] IETF. Retrieved from https://tools.ietf.org/html/rfc8778 [Accessed 06 Aug. 2020].

[106] Jean-Philippe Aumasson, et al.. (2015). Password Hashing Competition. [online] Retrieved from https://password-hashing.net/ [Accessed 29 Jul. 2017].

[107] Percival, C.. (2009). STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS. [online] BSDCan'09. Retrieved from http://www.tarsnap.com/scrypt/scrypt.pdf [Accessed 29 Jul. 2017].

[108] BSI. (2018). BSI TR-03111 Elliptic Curve Cryptography, Version 2.10. [online] Retrieved from https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_pdf.html [Accessed 08 Jun. 2018].

[109] M. Bellare, P. Rogaway, D. Wagner. (2004). The EAX Mode of Operation (A Two-Pass Authenticated-Encryption Scheme Optimized for Simplicity and Efficiency). [online] Retrieved from http://web.cs.ucdavis.edu/∼rogaway/papers/eax.pdf [Accessed 06 Apr. 2018].

[110] The HQC Team (see website for full list). (2025). HQC. [online] Retrieved from https://pqc-hqc.org/ [Accessed 16 Jul. 2025].

[111] M. Bellare, C. Namprempre. (2007). Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. [online] Retrieved from https://cseweb.ucsd.edu/∼mihir/papers/oem.pdf [Accessed 10 Apr. 2018].

[112] WebCrypto GOST team. (2018). WebCrypto GOST Library - **expired**, use Internet Archive only. [online] Retrieved from

http://gostcrypto.com/ [Accessed 07 May. 2018].

[113] Guan Zhi (Maintainer). (2018). The GmSSL Project. [online] Retrieved from http://gmssl.org [Accessed 07 May. 2018].

[114] N. Ferguson. (2005). Authentication weaknesses in GCM. [online] Retrieved from https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf [Accessed 10 Apr. 2018].

[115] State Service for Special Communications and Information Protection of Ukraine. (2002). DSTU 4145-2002 - Information Technologies. Cryptographic protection of information. A digital signature that is based on elliptic curves. Formation and verification..

[116] M. Bellare , P. Rogaway. (1995). Optimal Asymmetric Encryption - How to Encrypt with RSA. [online] Retrieved from https://cseweb.ucsd.edu/~mihir/papers/oae.pdf [Accessed 20 May. 2018].

[117] M. Bellare , P. Rogaway. (1998). PSS: Provably Secure Encoding Method for Digital Signatures. [online] Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.8704 [Accessed 20 May. 2018].

[118] J-S Coron. (2002). Optimal security proofs for PSS and other signature schemes. [online] Retrieved from https://eprint.iacr.org/2001/062.pdf [Accessed 20 May. 2018].

[119] N. Smart (Ed.), V. Rijmen, B. Gierlichs, K. Paterson, M. Stam, B. Warinschi, G. Watson. (2014). Algorithms, key size and parameters report - 2014. [online] ENISA. Retrieved from https://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/algorithms-key-size-and-parameters-report-2014/at_download/fullReport [Accessed 20 May. 2018].

[120] RSA Laboratories. (1993). PKCS #3: Diffie-Hellman Key-Agreement Standard. [online] RSA Laboratories. Retrieved from https://www.teletrust.de/fileadmin/files/oid/oid_pkcs-3v1-4.pdf [Accessed 10 Jun. 2018].

[121] Ned Freed and Martin Dürst. (2018). Character Sets. [online] IANA. Retrieved from https://www.iana.org/assignments/character-sets/character-sets.xhtml [Accessed 20 Feb. 2019].

[122] A. Menezes, P. van Oorschot, S. Vanstone. (1997). Handbook of Applied Cryptography.

[123] Wikipedia. (2018). Shamir's Secret Sharing. [online] Wikimedia Foundation, Inc.. Retrieved from https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing [Accessed 21 Jan. 2018].

[124] Wikipedia. (2019). Merkle tree. [online] Wikimedia Foundation, Inc.. Retrieved from https://en.wikipedia.org/wiki/Merkle_tree [Accessed 21 Apr. 2019].

[125] A. Menezes, M. Qu, S. Vanstone. (1995). Some New Key Agreement Protocols Providing Mutual Implicit Authentication.

[126] O. Goldreich. (2009). Foundations of Cryptography: Volume 2, Basic Applications.

[127] Wikipedia. (2019). NSA Suite B Cryptography. [online] Wikimedia Foundation, Inc.. Retrieved from https://en.wikipedia.org/wiki/NSA_Suite_B_Cryptography [Accessed 12 Feb. 2020].

[128] GnuPG. (2020). The keybox data format. [online] gnupg.org. Retrieved from https://git.gnupg.org/

cgi-bin/gitweb.cgi?p=gnupg.git;a=blob;f=kbx/keybox-blob.c [Accessed 04 Aug. 2020].

[129] Olivier Dubuisson (translated by Philippe Fouquart). (2000). ASN.1 - Communication between Heterogeneous Systems. [online] Elsevier-Morgan Kaufmann.. Retrieved from https://www.oss.com/asn1/resources/books-whitepapers-pubs/asn1-books.html [Accessed 16 Dec. 2020].

[130] Ronald Rivest. (1997). SEXP—(S-expressions). [online] csail.mit.edu. Retrieved from https://people.csail.mit.edu/rivest/sexp.html [Accessed 04 Aug. 2020].

[131] GnuPG. (2020). The Libgcrypt Reference Manual. [online] gnupg.org. Retrieved from https://www.gnupg.org/documentation/manuals/gcrypt/ [Accessed 04 Aug. 2020].

[132] GnuPG. (2019). The Libgcrypt Reference Manual. [online] gnupg.org. Retrieved from http://git.gnupg.org/cgi-bin/gitweb.cgi?p=gnupg.git;a=blob;f=agent/keyformat.txt;h=42c4b1f06faf1bbe71ffadc2fee0fad6bec91a97;hb=refs/heads/master [Accessed 04 Aug. 2020].

[133] Leslie Lamport. (1979). Constructing Digital Signatures from a One Way Function. [online] IEEE. Retrieved from https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/ [Accessed 29 Aug. 2020].

[134] Ralph Merkle. (1979). A Certified Digital Signature. [online] Crypto 89. Retrieved from http://www.merkle.com/papers/Certified1979.pdf [Accessed 29 Aug. 2020].

[135] D. Bernstein, A. Hülsing, S. Kölbl, R. Niedrhagen, J. Rijneveld, P. Schwabe. (2019). The SPHINCS+ Signature Framework. [online] SPHINCS+ Team. Retrieved from https://sphincs.org/data/sphincs+-paper.pdf [Accessed 29 Jan. 2022].

[136] Federal Office for Information Security. (2021). BSI TR-02102-1: “Cryptographic Mechanisms: Recommendations and Key Lengths” Version: 2021-1. [online] BSI. Retrieved from https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf [Accessed 29 Jan. 2022].

[137] Classic McEliece Submission Team. (2020). Classic McEliece: conservative code-based cryptography. [online] classic.mceliece.org. Retrieved from https://classic.mceliece.org/nist/mceliece-20201010.pdf [Accessed 29 Jan. 2022].

[138] FrodoKEM Submission Team. (2021). FrodoKEM: Learning With Errors Key Encapsulation. [online] frodokem.org. Retrieved from https://frodokem.org/files/FrodoKEM-specification-20210604.pdf [Accessed 29 Jan. 2022].

[139] Falcon Submission Team. (2020). Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. [online] falcon-sign.info. Retrieved from https://falcon-sign.info/falcon.pdf [Accessed 26 Dec. 2022].

[140] CRYSTALS Submission Team. (2021). CRYSTALS-Dilithium - Algorithm Specifications and Supporting Documentation (Version 3.1). [online] pq-crystals.org. Retrieved from https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf [Accessed 26 Dec. 2022].

[141] CRYSTALS Submission Team. (2021). CRYSTALS-Kyber (version 3.02) - Submission to round

3 of the NIST post-quantum project. [online] pq-crystals.org. Retrieved from https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf [Accessed 26 Dec. 2022].