Security and Trust Services APIs for Java™ 2 Platform, Micro Edition Developer's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A. 650-960-1300

December 2004

Contents

Preface

This book describes how to use the Security and Trust Services APIs (SATSA) for the Java™ 2 Platform, Micro Edition (J2ME™) 1.0 Reference Implementation (RI) to create applications that can communicate with a smart card and perform cryptography.

Who Should Read This Book

This book is designed for Mobile Information Device Profile (MIDP) developers who wish to learn how to use the SATSA APIs. Its purpose is to provide a practical introduction to SATSA without being an exhaustive reference. For comprehensive coverage of SATSA, refer to the Security and Trust Services API specification, available at `http://jcp.org/en/jsr/detail?id=177`.

Before You Read This Book

You should already understand how to create MIDP applications before you read this book. You also need to understand the Java Card platform. The following books are good resources:

- *Java Card Platform, Version 2.2.1 Development Kit, User's Guide*, available from `http://java.sun.com/products/javacard/dev_kit.html`
- *MIDP Reference Implementation, Version 2.0, Using MIDP*, available from `http://java.sun.com/products/midp/`
- *MIDP Reference Implementation, Version 2.0, Creating MIDlets Suites*, available from `http://java.sun.com/products/midp/`

How This Book Is Organized

Chapter 1 is an overview of SATSA and describes the tools that are available in the SATSA RI.

Chapter 2 gives background information on the capabilities of the SATSA APIs and how they can be used in MIDP applications.

Chapter 3 describes how to use SATSA-APDU to communicate with smart card applications.

Chapter 4 describes using SATSA-JCRMI as an alternate method for communicating with smart card applications.

Chapter 5 details how to use the SATSA-PKI API.

Chapter 6 includes information on programming with SATSA-CRYPTO.

Online References

The following web page is a collection of resources for learning MIDP 2.0 application programming:

Learning Path: Getting Started with MIDP 2.0

`http://developers.sun.com/techtopics/mobility/learn/midp/midp20/`

Sun’s Mobility page is also a helpful starting point for wireless and Java Card topics:

`http://developers.sun.com/techtopics/mobility/`

The J2ME Wireless Toolkit is an excellent tool for creating MIDP applications:

`http://java.sun.com/products/j2mewtoolkit/`

The Source for Java Developers web site enables you to access Java™ platform technical documentation online:

`http://java.sun.com/reference/docs/`

We Welcome Your Comments

We are constantly improving our products and documentation and welcome your comments and suggestions. Provide feedback to Sun at `http://java.sun.com/docs/forms/sendusmail.html`.

CHAPTER 1

Introduction

The Security and Trust Services APIs (SATSA) provide smart card access and cryptographic capabilities to applications running on small devices. The SATSA specification defines four distinct APIs:

- SATSA-APDU allows applications to communicate with smart card applications using a low-level protocol.
- SATSA-JCRMI provides an alternate method for communicating with smart card applications using a remote object protocol.
- SATSA-PKI allows applications to use a smart card to digitally sign data and manage user certificates.
- SATSA-CRYPTO is a general-purpose cryptographic API that supports message digests, digital signatures, and ciphers.

This book describes the use of SATSA in Mobile Information Device Profile (MIDP) applications, although its use in other J2ME platform environments is also possible. The SATSA Reference Implementation (RI) used in this book is based on a MIDP 2.0 implementation.

Understanding the SATSA RI

The four SATSA APIs are defined by JSR 177, the Security and Trust Services APIs specification. The specification is a document that defines the APIs. The SATSA RI is a piece of software that implements the specification. This book describes how to use the SATSA RI to develop applications.

The SATSA RI is available at `http://java.sun.com/products/satsa/`. It is based on MIDP 2.0 and includes complete implementations of the SATSA APIs.

Note – The SATSA specification doesn't mandate the use of a smart card. Instead, it uses a more general term *security element*, which could be a removeable smart card, an embedded SIM card, or even special hardware inside a device. This book uses the term *smart card*, but bear in mind that it's ultimately the SATSA implementation that determines what security element is used for secure operations.

Installing Development Tools

To develop and test MIDP applications that use SATSA, you will need the following tools:

- **Java 2 Standard Edition Software Development Kit (J2SE SDK) version 1.4.2**. This includes the Java technology compiler, which is necessary for both MIDP and Java Card™ technology development. The J2SE SDK is freely available at `http://java.sun.com/j2se/`. You should install before other tools.
- **Security and Trust Services APIs Reference Implementation (SATSA RI) version 1.0**. This includes an implementation of the SATSA specification and a device emulator for testing applications. Get the SATSA RI from `http://java.sun.com/products/satsa/`. It is packaged as a zip archive, which you can unpack wherever you'd like.
- **Java Card Development Kit version 2.2.1**. This includes a Java Card technology simulator that you can use to test the interaction between a SATSA application and a Java Card applet. It also includes everything you need to create your own applets. The Java Card Development Kit is available from `http://java.sun.com/products/javacard/`.

At this writing (December 2004), no MIDP development tools support SATSA. The only way to build SATSA applications is using the SATSA RI. This is likely to change in the future. For example, if the J2ME Wireless Toolkit supports SATSA in the future, you can use it to handle most of the details of building and packaging applications. In the meantime, the following section describes the manual steps to follow with the SATSA RI.

Building Applications with the SATSA RI

Building SATSA applications is exactly like building other MIDP applications at the command line. The basic steps are described here:

1. Edit source files.

2. Compile source files using `javac`. You need to tell the compiler where to find the MIDP and SATSA APIs. They are located in *{satsa}*`\classes`, where *{satsa}* is the installation directory of the SATSA RI. Here is an example that compiles source files in the `src` directory and places the compiled class files in the `classes` directory. The example is split across multiple lines for clarity.

```
javac -classpath \satsa1.0\classes
      -target 1.1
      -d classes
      src\*.java
```

3. Preverify the compiled class files, using *{satsa}*`\bin\preverify`. This example takes class files in the `classes` directory and places the preverified versions in a directory named `preverified`. The MIDP Reference Implementation documentation describes compiling classes into a directory named `tmpclasses`, then placing preverified classes in `classes`. This document uses the more descriptive directory names `classes` and `preverified`.

```
\satsa1.0\bin\preverify -classpath \satsa1.0\classes
                        -d preverified
                        classes
```

4. Package the application using the J2SE platform's `jar` tool. The following example takes classes from the `preverifed` directory and places them in a Java Archive (JAR) file called `bin\Mohair.jar`, adding information to the manifest from the file `bin\MANIFEST.MF`.

```
jar cvfm bin\Mohair.jar bin\MANIFEST.MF -C preverified .
```

5. Test on the SATSA RI device emulator, using *{satsa}*`\bin\midp`.

To avoid tedious typing, use a tool to automate this process. Ant (`http://ant.apache.org/`) is an excellent choice. For more information on using Ant for MIDP development, try this article:

Managing Wireless Builds with Ant

`http://developers.sun.com/techtopics/mobility/midp/articles/ant/`

CHAPTER 2

Application Design

This chapter describes the strengths of various components of a typcial SATSA application. The remainder of the chapters in this book describe specific parts of SATSA, but this chapter bridges the gap between a low-level understanding of APIs and a high-level understanding of application design.

Architectural Components

A typical SATSA application consists of three types of hardware:

- A *server* contains persistent data for the application and performs most of the actual work.
- A MIDP *device*, usually a mobile phone, provides the user with a way to interact with the server. The device interacts with the server using the network.
- A *smart card*, connected temporarily or permanently to the MIDP device, can perform cryptography or additional processing.

There are many ways to structure applications, and cases where one or more components are unnecessary. This simple architecture indicates components that are most often present in SATSA applications.

Servers are Big

Usually, a server is a relatively powerful computer with a fast processor and lots of memory. Do the bulk of work in your application on the server, if at all possible.

Aside from the available power, the other reason it’s nice to have application code running on a server is that it’s easy to change. You can optimize, fix bugs, or make updates in server code without changing the client code.

Devices are Portable and Online

The power of a MIDP device is that it’s a highly portable networked computer. Many mobile phones fit easily in a pocket. Network connectivity means that a MIDP device can perform serious work using a server without having heavy processing power on the device itself.

MIDP applications can present a sophisticated user interface but defer the heavy lifting to the server.

Smart Cards are Hard to Break

A smart card is a tiny computer, complete with a processor and memory. It is capable of storing information and performing cryptography. That by itself doesn’t distinguish a smart card. After all, a MIDP device could perform the same tasks easily.

Smart card hardware is tamper-resistant, which means that if someone steals your smart card, it will be harder to extract information than from a more traditional device like a MIDP mobile phone.

Smart cards are most valuable as a physical representation of a private information. If you store secret information on a smart card, it’s very easy to understand that the card itself must be kept physically secure. A smart card closely resembles a credit card, which means it’s easy for users to comprehend that the smart card needs to be treated similarly.

In the case of private cryptographic keys, smart cards are especially useful. A smart card can cryptographically sign information, proving possession of a private key, without the key itself ever leaving the smart card.

Understanding SATSA’s Place in the World

You’ve already read about the four APIs defined by the SATSA specification. How do these APIs relate to the architectural elements described previously?

SATSA-APDU and SATSA-JCRMI both provide a way for MIDP applications to communicate with a smart card. Using either API, the basic function is the same: the MIDP application asks the smart card to do something and the smart card provides a response.

SATSA-PKI allows a MIDP application to request cryptographic signature or authentication from a smart card.

SATSA-CRYPTO provides MIDP applications with the ability to compute message digests, verify digital signatures, and encrypt and decrypt data. Of the four SATSA APIs, SATSA-CRYPTO is the only one that does not involve interaction with a smart card.

Choosing APDU or JCRMI

If you want to communicate with a smart card from a MIDP application, which API should you use, SATSA-APDU or SATSA-JCRMI?

Functionally, there is no difference between the two. Both APIs provide a way for a MIDP application to request action from the smart card and receive a response. The distinction between the two APIs is how the programmer communicates with a smart card.

SATSA-APDU uses a protocol based on the exchange of byte arrays between the MIDP application and the smart card. It is a low-level protocol. By contrast, SATSA-JCRMI exposes a more object-oriented protocol to the developer. Using SATSA-JCRMI, the developer can request work from the smart card using a remote object protocol, a high-level protocol.

Generally speaking, programmers are less likely to make mistakes using SATSA-JCRMI than using SATSA-APDU. Because of this, SATSA-JCRMI is the preferred method of communication. The only situations where you might choose SATSA-APDU are to port legacy applications or to use card hardware that doesn't support JCRMI.

Application Distribution

How are SATSA applications distributed? How do users get MIDP devices and smart cards? Obviously, the answer varies from application to application. Here are two possibilities:

- Existing GSM phones already contain a smart card, called a SIM card. SATSA could provide MIDP applications with access to the SIM card. Wireless carriers might sell phones bundled with SIM cards to users. On these phones, MIDP applications could use SATSA to enable online purchasing and authentication using the SIM card.
- MIDP phones that implement SATSA and include a smart card slot might be created in the future. Users would buy these phones from carriers or manufacturers, then obtain a smart card from another source. The user could use the phone to browse an online catalog, then insert a smart card to make a purchase. SATSA would enable the interaction between the catalog appliction and the smart card.

Remember Threading

If your application is going to do anything that takes longer than a few hundred milliseconds, you need to put that work in a separate thread. Remember, most of the code you write in a MIDlet is in threads that belong to the system, not to your application. If you perform lengthy work in a system thread, you'll lock up the user interface on the device, which is a very unpleasant experience for your user.

Some operations in SATSA are potentially time-consuming. Whenever you communicate with a smart card via SATSA-APDU or SATSA-JCRMI, consider both the communication time as well as the time it will take the card to process the request and generate a response. Also bear in mind that the system might need to display prompts to ask for permission or access codes.

Cryptographic operations are typically slow, especially when they are performed on small hardware like MIDP devices or tiny hardware like smart cards. You can expect some methods in SATSA-PKI and SATSA-CRYPTO to take a significant amount of time. Make sure your application handles threading correctly and provides a useful, lively interface even while slow operations are taking place.

For a detailed look at threading in MIDlets related to network access, try this article:

> *Networking, User Experience, and Threads*
> `http://developers.sun.com/techtopics/mobility/midp/articles/threading/`

CHAPTER **3**

The SATSA-APDU Optional Package

The SATSA-APDU Optional Package enables MIDP applications to communicate with a smart card using a protocol based on Application Protocol Data Units (APDUs). This protocol is defined by ISO 7816-4 and described in the Java Card Development Kit documentation. See Chapter 1 for a pointer to the Java Card Development Kit.

An APDU is a short message represented by bytes. SATSA-APDU enables your application to exchange APDU messages with a card application. The SATSA-JCRMI Optional Package, which is described in the next chapter, allows applications to interact with card applications at a higher level. SATSA-JCRMI allows your application to use objects on a smart card directly. Which API you use depends on the requirements of your application.

In SATSA-APDU, messages are either *commands* or *responses*. Your application can use SATSA-APDU to send commands to a card application and receive responses. An application could send a command message to query an account balance and receive the answer in the response message.

SATSA-APDU is a single interface, `javax.microedition.apdu.APDUConnection`, which contains seven methods.

This chapter describes basic use of SATSA-APDU and includes small pieces of example code that illustrates the concepts. Appendix A includes complete example MIDlets that you can compile and run.

Opening an APDU Connection

SATSA-APDU is implemented as an extension of the Generic Connection Framework (GCF). The GCF provides a uniform API for many different types of data connections. In typical use of the GCF, your application requests a connection from a factory class, `javax.microedition.io.Connector`. If the connection is established, `Connector` hands your application an object that is capable of exchanging data on the connection. To make an HTTP connection to a server, for example, your application would call `Connector.open("http://myserver.com/")`. The GCF returns a connection object that makes it possible for your application to send an HTTP request and receive the response.

In SATSA-APDU, your application passes a locator string representing a card application to `Connector`'s `open()` method, which returns an `APDUConnection` that can be used to communicate with the card application.

The locator string begins with `apdu` and specifies the slot number and the card application identifier. In following example, the application attempts to open a connection with a card application with identifier `a0.00.00.00.62.03.01.0c.02.01` running on a smart card in slot 0.

```
String purseURL = "apdu:0;target=a0.00.00.00.62.03.01.0c.02.01";
APDUConnection purseConnection;
purseConnection = (APDUConnection)Connector.open(purseURL);
```

`Connector`'s `open()` method throws a `ConnectionNotFoundException` if there is no such slot, no card in the given slot or if the specified card application is not present. Other types of exceptions are thrown if other types of failures occur. Check the documentation for `APDUConnection` for full details.

You can discover a device's available card slots at runtime by retrieving the value of the `microedition.smartcardslots` system property. Use the `getProperty()` method in the `java.lang.System` class.

The property string contains a comma-separated list of slot numbers. Each slot number is accompanied by a character that indicates if the slot is cold-swappable (C) or hot-swappable (H). See the package description for `javax.microedition.io` in the Security and Trust Services APIs Specification for more details.

Exchanging APDU Messages

Use the `exchangeAPDU()` method to send a command to a card application and receive a response. Pass a byte array containing a command APDU to `exchangeAPDU()`. The command is sent to the card. When the card sends its response APDU, this method returns the response as another byte array. A variety of exceptions might be thrown if communications failures or other disasters occur.

In the following example, the APDU commands to be sent to the card are contained in a separate class file called `APDUList`. The array `firstSessionApdus[]` are specified in that file and called in the following example.

```
byte[] apdu = {
  (byte)0x00, (byte)0x20, (byte)0x00, (byte)0x82, (byte)0x04,
  (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x00
};
byte[] response = purseConnection.exchangeAPDU(apdu);
```

The `exchangeAPDU()` method blocks until a response is received from the card application.

Closing an APDU Connection

To close an `APDUConnection`, simply call its `close()` method as shown in the following example.

```
purseConnection.close();
```

If you close a connection that is being used by other threads to exchange APDUs, the connection is closed immediately and the `exchangeAPDU()` methods in other threads throw `InterruptedException`.

Supporting the (U)SIM Application Toolkit

If your application is running in a SATSA implementation that supports the (U)SIM Application Toolkit (U)SAT, you can communicate with SAT applications by using `APDUConnection`.

There are various constraints on this type of usage, which are definitively detailed in the *Security and Trust Services API Specification*.

Opening a connection using (U)SAT is straightforward. The following example attempts a (U)SAT connection on slot 0.

```
APDUConnection sat;
sat = (APDUConnection)Connector.open("apdu:0;target=SAT");
```

Once the connection is established, the application can send ENVELOPE APDU commands to the smart card using the `exchangeAPDU()` method. An *envelope* contains an array of APDU bytes in hexidecimal format corresponding to an APDU status word.

```
byte[] envelope = {
   (byte)0xA0, (byte)0xC2, (byte)0x00, (byte)0x00, (byte)0x2f,
   (byte)0xD1, (byte)0x2d, (byte)0x82, (byte)0x02, (byte)0x83,
   (byte)0x81, (byte)0x06, (byte)0x05, (byte)0x80, (byte)0x11,
   (byte)0x22, (byte)0x33, (byte)0x44, (byte)0x8B, (byte)0x20,
   (byte)0x10, (byte)0x02, (byte)0x81, (byte)0x55, (byte)0x7F,
   (byte)0xF6, (byte)0x00, (byte)0x11, (byte)0x29, (byte)0x12,
   (byte)0x00, (byte)0x00, (byte)0x04, (byte)0x14, (byte)0x10,
   (byte)0x11, (byte)0x12, (byte)0x13, (byte)0x14, (byte)0x15,
   (byte)0x16, (byte)0x17, (byte)0x18, (byte)0x19, (byte)0x1a,
   (byte)0x1b, (byte)0x1c, (byte)0x1d, (byte)0x41, (byte)0xAB,
   (byte)0xCD, (byte)0xEF
};
byte[] response = sat.exchangeAPDU(envelope);
```

To close a (U)SAT connection, use the `close()` method.

```
sat.close();
```

For a complete example, see Appendix A.

CHAPTER 4

The SATSA-JCRMI Optional Package

The SATSA-JCRMI Optional Package enables you to communicate with a smart card using the Java Card Remote Method Invocation (JCRMI) protocol.

JCRMI is a distributed computing protocol, which means it allows applications that are not located on the card to use objects that exist on the card. Applications use a *remote interface* to work with an object on the card. When the application calls a method on the remote interface, the method call is transferred to the card using the JCRMI protocol. The method is run on the card and the results are returned to the application.

JCRMI is a subset of the J2SE platform's Remote Method Invocation (RMI) that is tailored for the constraints of a Java Card environment. JCRMI places restrictions on the allowed parameter types and return types for remote objects. See the SATSA specification for more details.

SATSA-JCRMI comprises classes and interfaces in several different packages:

- The `Remote` interface and `RemoteException` class from the J2SE platform RMI implementation are both included in the `java.rmi` package.
- `JavaCardRMIConnection`, in the `javax.microedition.jcrmi` package, serves as a bridge between the Generic Connection Framework (GCF) and SATSA-JCRMI.
- Exception classes from the Java Card API are also included because these exceptions might be thrown when calling methods on a card object. These exceptions are defined in `javacard.framework`, `javacard.framework.service`, and `javacard.security`.

This chapter describes basic use of SATSA-JCRMI and includes example code that illustrates the concepts. A complete example is included in Appendix B.

A Brief Introduction to SATSA-JCRMI

The fundamental concept of SATSA-JCRMI is that your application has the ability to call methods on an object that lives on the smart card. The card object is a *remote object*, and your application, the *client*, interacts with the remote object using a *remote interface*. The actual communication is accomplished using a *stub* object that you must generate at development time. The stub implements the remote interface and contains the actual details of communicating with the remote object.

At development time, your responsibilities are as follows:

1. Create a remote interface that includes the methods you wish to call on the remote object.
2. Create a stub class that implements the remote interface. You can do this using the JCRMI stub compiler.
3. Create client code in your application that opens a connection and calls methods on the remote object.

Creating a Remote Interface

Creating a remote interface is relatively straightforward. The remote interface contains all the methods you wish to call on the remote object. It must extend `java.rmi.Remote`, and every method must throw `RemoteException`. `RemoteException` is thrown if a communication failure occurs between your application and the remote object.

The following example shows a remote interface for a `Purse` class that includes definitions for five methods.

```
public interface Purse extends Remote {
  public short getBalance() throws RemoteException;
  public void debit(short m) throws RemoteException, UserException;
  public void credit(short m) throws RemoteException, UserException;

  public void setAccountNumber(byte[] number)
      throws RemoteException, UserException;
  public byte[] getAccountNumber() throws RemoteException;
}
```

Creating a Stub Class

A stub class implements the remote interface and provides the code that actually calls methods on the remote object. The stub class lives in the same package as the remote interface. Its name is the remote interface name with `_Stub` appended. For example, if the remote interface in the preceding example is `Purse`, the stub class name is `Purse_Stub`.

The SATSA reference implementation includes a stub compiler utility, `jcrmic`, that generates a stub class for a given remote interface. All you really need to do is tell `jcrmic` where to find your remote interface. The command line syntax is as follows:

```
jcrmic <options> <interface names>
```

The `jcrmic` utility recognizes these options:

- `-classpath <path>` - Specifies the location of the input class files.
- `-d <directory>` - Specifies where the compiler will place the generated files.
- `-nocompile` - Specifies that generated source files will not be compiled.
- `-keep` - Specifies that generated source files will not be deleted after compilation.

By default, `jcrmic` calls the compiler specified by the `JAVAC_PATH` environment variable. The generated source files are deleted after they are compiled and the stub classes are stored in the directory specified by the `-d` option.

For example, you could create a stub class for the `Purse` remote interface like this:

```
jcrmic -d . Purse
```

Connecting to a Remote Object

SATSA-JCRMI is implemented through the Generic Connection Framework (GCF). The basic tenet of the GCF is that you supply a connection string and receive an object that allows you to communicate. In the case of SATSA-JCRMI, you supply a connection string and receive an interface that allows you to call methods on a remote object, an object that resides on a smart card.

The connection string specifies a slot number and a card application identifier (AID). The following example connects to an application with identifier `a0.0.0.0.62.3.1.c.8.1` on a card in slot 0.

```
String url = "jcrmi:0;AID=a0.0.0.0.62.3.1.c.8.1";
JavaCardRMIConnection connection = null;
connection = (JavaCardRMIConnection)Connector.open(url);
```

Various kinds of exceptions might be thrown from `Connector`'s `open()` method if any kind of communication failure occurs. Be sure your application catches and handles exceptions properly. A complete example is included in Appendix B.

To retrieve a reference to the remote object, use the `getInitialReference()` method in `JavaCardRMIConnection`. This method returns an instance of `java.rmi.Remote` that you cast to the appropriate remote interface type, as shown here:

```
Purse purse = (Purse)connection.getInitialReference();
```

Calling Methods on a Remote Object

Once you establish a reference to a remote object, you can treat it almost like a local object. When you call methods on your instance of the remote interface, the same methods are called on the remote object, and return values are transferred from the remote object to your application.

The only tricky part is that if a communication failure occurs, you must be prepared to handle `RemoteException` properly. Remember, every method in the remote interface can throw `RemoteException`, because every remote method call has the possibility of communication failure.

Closing a Connection to a Remote Object

When you are finished calling methods on a JCRMI remote object, be sure to close the corresponding `JavaCardRMIConnection`, as shown in the following example.

```
connection.close();
```

CHAPTER 5

The SATSA-PKI Optional Package

SATSA-PKI allows applications to request digital signatures from a smart card. It also includes basic certificate management methods.

SATSA-PKI contains two classes, `javax.microedition.securityservice.CMSMessageSignatureService` and `javax.microedition.pki.UserCredentialManager`.

A Brief Introduction to PKI

Public Key Infrastructure (PKI) is one way of associating cryptographic keys with people, businesses, or other objects. It is based on cryptographic *signatures* and *certificates*.

A signature is a value computed from input data and a private key, in a process that is called *signing* the data. A signature can be *verified* using the same input data and the public key that corresponds to the signing key. Signatures are useful for ensuring data integrity. If Bob signs some data with his private key and sends it to Fiona, she can use his public key to verify the signature, assuring herself that the data has not changed in transit.

A certificate is a special package that consists of information about a person, a copy of the person’s public key, and a signature by another person. In essence, a certificate says “I, the signer, certify that this person has this public key.” For example, Fiona could create a certificate that links Bob to his public key.

Why is a certificate useful? It allows you to trust someone you might not know very well. For example, suppose that Orrin trusts Fiona but doesn’t know Bob. If Orrin gets a signed message from Bob, how can he verify the signature? Bob can supply his certificate from Fiona that links him to his public key. Orrin can verify the certificate using Fiona’s public key, which means he can now use Bob’s public key to verify the message he just received from Bob.

Certificate can be chained together in this way until they end in a *root* certificate, issued by a Certificate Authority (CA). The term *root* is a little misleading, implying strength and fundamental structure. In truth, a CA root certificate is a *self-signed* certificate, which means the certificate was signed using the private key whose matching public key is contained in the certificate. Self-signed certificates are easy to make, and you can claim to be anyone you want in a self-signed certificate. The reason it's hard to forge a CA certificate is that there are so many authentic copies already distributed throughout the computing world.

In practice, devices (or browsers) usually have a database of CA root certificates so that when certificate chains are received, they can be verified back to a CA root.

This section provides only a brief introduction to signatures and certificates. For deeper coverage, try Bruce Schneier's excellent *Applied Cryptography* (Wiley, 1995).

Generating Signatures

The `CMSMessageSignatureService` class allows MIDP applications to request a cryptographic signature. A signature is useful for two reasons. First, it proves the identity of the user, which is called *authentication*. Second, it certifies the *integrity* of data. This use is also referred to as *non-repudiation*, which means after data is signed, the signer can't "take it back" or deny signing the data. A user's keys can be marked specifically for one use or the other.

`CMSMessageSignatureService` supports both uses of digital signatures by selecting an appropriate key to generate a signature. The type of key selected depends on whether you call the `sign()` method or one of the `authenticate()` methods.

Although it is likely that the signature is performed on a smart card, if one is available, the cryptographic signing can take place anywhere, depending on the implementation. At the application level, simply call one of the methods in `CMSMessageSignatureService` and the implementation handles the details.

A call to `authenticate()` or `sign()` returns a *formatted digital signature*, which is essentially the signature itself plus information about the signer and the data that was signed. `CMSMessageSignatureService` produces formatted digital signatures that conform to the Cryptographic Message Syntax (CMS), which is defined in RFC 2630 and extended in RFC 2634. For more information, refer to the RFCs directly:

`http://www.ietf.org/rfc/rfc2630.txt`
`http://www.ietf.org/rfc/rfc2634.txt`

Typically, a MIDP application uses `CMSMessageSignatureService` to have a smart card create a signature, then passes the signature to a server where it is verified and used to perform some action on behalf of the user.

Finding Keys

When your application calls `sign()` or `authenticate()`, the SATSA implementation tries to locate an appropriate signing key. You can pass additional parameters to `sign()` or `authenticate()` to affect how the implementation searches for keys.

You can pass a list of acceptable Certificate Authorities (CAs). The implementation only considers keys that are certified by the CAs in the list you supply. The list is a string array, where each string is the distinguished name of a CA, formatted according to RFC 2253.

If the implementation cannot find any appropriate keys for signing, it displays a prompt to the user. You can specify the text of the prompt that is shown to the user, which is usually something like "Please insert such-and-so card now."

Conversely, if the implementation finds more than one appropriate signing key, it is obliged to offer the user a choice of keys.

The use of private keys on a smart card or other security element might be restricted. For example, a smart card could require the user to enter a short code (a PIN) to access a private key, or a more sophisticated card might need to verify a fingerprint. As an application developer, bear in mind that a call to `sign()` or `authenticate()` might result in several user prompts outside your application's control.

Signature Options

The form of the signature returned by `sign()` or `authenticate()` can be controlled using two options that are defined as constants in the `CMSMessageSignatureService` class.

`SIG_INCLUDE_CONTENT` indicates that the formatted signature should include the content that was signed (an *opaque* signature). The absence of this option tells the implementation to create a *detached* signature.

The other option, `SIG_INCLUDE_CERTIFICATE`, controls whether the signer's certificate is included in the formatted signature.

To combine the two options, use the bitwise OR operator (`|`).

Signing Data

To generate a signature to verify data integrity, use the `sign()` method:

```
public static final byte[] sign(String stringToSign,
    int options, String[] caNames, String securityElementPrompt)
    throws CMSMessageSignatureServiceException,
           UserCredentialManagerException
```

The stringToSign is the data you would like signed. The options are described previously, as is the list of CAs in caNames. Finally, securityElementPrompt is the string that is shown to the user if an appropriate key cannot be found.

Before proceeding with the signing, the implementation must display stringToSign as well as the certificate name corresponding to the key that will be used for signing. The user must explicitly confirm everything before the implementation can create the signature.

Authentication

Authentication proceeds almost exactly like signing, except the implementation searches for keys that are marked for authentication. In addition, the authenticate() method has two overloaded forms. The first form is just like the sign() method:

```
public static final byte[] authenticate(String stringToAuthenticate,
    int options, String[] caNames, String securityElementPrompt)
    throws CMSMessageSignatureServiceException,
           UserCredentialManagerException
```

The second form signs a byte array instead of a string.

```
public static final byte[] authenticate(
    byte[] byteArrayToAuthenticate,
    int options, String[] caNames, String securityElementPrompt)
    throws CMSMessageSignatureServiceException,
           UserCredentialManagerException
```

In this case, the byte array is not displayed to the user, although the implementation is still supposed to show the name of the certificate whose key will be used for signing.

Exceptions

The sign() and authenticate() methods can throw CMSMessageSignatureServiceException, a special type of exception class that includes constant values representing the cause of the exception. The getReason() method returns a value that indicates the specific problem with the cryptographic signature or the security element that was supposed to generate the signature. See the documentation of the CMSMessageSignatureServiceException class for the reason values and their meanings.

Certificate Management

The other class in SATSA-PKI, UserCredentialManager, provides methods for managing a user's keys. These are the same keys that CMSMessageSignatureService uses to generate signatures.

UserCredentialManager provides methods for adding keys, removing keys, and generating a request for new keys. Keys are represented by certificates that link keys to specific people.

Adding Certificates

Keys are represented by a *certificate path*. The path is a trail of certification that begins with a CA and ends with the user's keys. For example, the certificate path might consist of three certificates:

1. A self-signed root certificate from the Acme Certificate Authority.
2. A certificate containing the Trustworthy Software company's public key, signed by Acme.
3. A certificate representing the user's key pair, signed by Trustworthy Software.

The certificate path is represented as a byte array, and normally is the response you receive after submitting a Certificate Signing Request (CSR) to a certificate issuer. Later in this chapter, "Requesting a New Certificate" describes how to create a CSR with UserCredentialManager.

To add a user's certificate path, use this method in UserCredentialManager:

```
public static final boolean addCredential(String certDisplayName,
    byte[] pkiPath, String uri)
    throws UserCredentialManagerException
```

You need to supply a human-readable name for the key pair in certDisplayName. The actual certificate path is pkiPath.

The uri parameter is optional. If it is not null, uri represents the user certificate, the last one in the path. This is useful for devices whose keys are stored off the device. On some WAP networks, for example, certificate stores are maintained on a gateway server. In this case, a device can use uri to reference the certificate on the gateway server rather than storing certificates locally.

The user is prompted to confirm the addition.

Where Do Certificates Go?

When you add certificates with UserCredentialManager, where do they go? The specification is deliberately flexible about this question and dictates only that they are added to a *certificate store*. This means that a device is assumed to contain some database of certificates, but it could be stored on the device or on a smart card or other security element. At the application level, you don't have to worry where the certificates are physically stored. You can add certificates with UserCredentialManager and obtain signatures with CMSMessageSignatureService.

Removing Certificates

You can remove certificates from the certificate store with the `removeCredential()` method:

```
public static final boolean removeCredential(String certDisplayName,
    byte[] issuerAndSerialNumber, String securityElementID,
    String securityElementPrompt)
    throws UserCredentialManagerException
```

You need to supply the displayable name in `certDisplayName` (the same name that was used to add the certificate). You also need to supply the certificate's issuer and the certificate serial number, formatted as a byte array conforming to RFC 3369.

The `securityElementID` parameter identifies the smart card or other security element where the given certificate is expected to be found. If this paramter is `null`, the implementation attempts to find the given certificate wherever it can. Finally, `securityElementPrompt` is text that is shown to the user to suggest what needs to be done for the implementation to find the certificate it needs to remove. Usually, this is a suggestion to insert a smart card into the device.

Requesting a New Certificate

The last method in `UserCredentialManager` creates a Certificate Signing Request (CSR) that you can use to request a certificate for a key pair.

```
public static final byte[] generateCSR(String nameInfo,
    String algorithm, int keyLen, int keyUsage,
    String securityElementID, String securityElementPrompt,
    boolean forceKeyGen)
    throws UserCredentialManagerException,
           CMSMessageSignatureServiceException
```

In general, new keys are generated on a smart card (or other security element) and the CSR is also generated on the smart card. You can specify which security element to use with `securityElementID`, and `securityElementPrompt` is shown to the user if the security element your application requests is not immediately available. If you pass `null` for `securityElementID`, the implementation selects an appropriate security element if one is available.

The `nameInfo` parameter contains the name you want to associate with a key pair. It is a distinguished name which is formatted according to RFC 2253. You can pass `null` for `nameInfo`, in which case the implementation chooses a name for you.

The actual keys that are used may already exist on the smart card, or they are created if no existing appropriate keys are found. The `forceKeyGen` parameter controls whether existing keys can be used (`false`) or new keys must be generated (`true`).

The type of keys are controlled by `algorithm`, `keyLen`, and `keyUsage`.

The key algorithm is an *object identifier* for a cryptographic signature algorithm, as described in RFC 1778. `UserCredentialManager` provides two constants, `ALGORITHM_DSA` and `ALGORITHM_RSA`, that can be used for the `algorithm` parameter.

The desired key length, in bits, is passed in the `keyLen` parameter.

As you already learned in the `CMSMessageSignatureService` class, keys can be used for authentication or general-purpose signing. The certificate that contains a key indicates its intended use. When you generate a CSR with `UserCredentialManager`, the `keyUsage` parameter specifies the purpose of the key. Use a constant value, either `KEY_USAGE_AUTHENTICATION` or `KEY_USAGE_NON_REPUDIATION`.

CHAPTER **6**

The SATSA-CRYPTO Optional Package

The SATSA-CRYPTO Optional Package allows applications to use cryptographic tools like message digests, digital signatures, and ciphers. These tools give applications the ability to support commerce, safeguard information, and protect privacy.

SATSA-CRYPTO is a subset of the J2SE platform Java Cryptography Extension (JCE). For more information on JCE, see `http://java.sun.com/products/jce/`.

This chapter describes how to use SATSA-CRYPTO from your own applications. It is not a comprehensive tutorial on cryptographic concepts, although it does include some introductory explanations. For a more detailed introduction to the concepts, read this article:

MIDP Application Security 1: Design Concerns and Cryptography
`http://developers.sun.com/techtopics/mobility/midp/articles/security1/`

For deeper coverage, try Bruce Schneier's excellent *Applied Cryptography* (Wiley, 1995).

SATSA-CRYPTO provides an API for three cryptographic tools:

- *Message digests* can create fingerprints of data. They are handy for checking the integrity of data that has been transmitted over a network.
- *Digital signatures*, like message digests, are useful for verifying data integrity, but are generated using a private key. They are much harder to forge.
- *Ciphers* are used to encrypt and decrypt data.

The classes and interfaces of SATSA-CRYPTO are located in the same packages as the J2SE platform JCE: `java.security`, `java.security.spec`, `javax.crypto`, and `javax.crypto.spec`.

Using Ciphers

A cipher is useful for keeping data secret. The cipher takes regular data, the *plaintext*, and converts it into an unreadable form, called *ciphertext*. This process is *encryption*. The cipher can also *decrypt* ciphertext to restore the original plaintext.

A cipher is valuable because you can encrypt sensitive information, like credit card numbers or your Aunt May's medical records. As ciphertext, the data can be safely transmitted over an insecure medium like the Internet. At the receiving end, you can use the cipher to decrypt the ciphertext and retrieve the original information.

Cryptographic ciphers use *keys* to make unique ciphertext. Only the same key, or a matching key, will decrypt the ciphertext and restore the plaintext. A *symmetric* cipher uses the same key for encryption and decryption. An *asymmetric* cipher uses one key for encryption and a matched key for decryption. The matched set of keys is called a *key pair*. One key in a key pair, the *public key*, can be freely distributed and used by other parties for verification. The other part of the key pair, the *private key*, must be kept secret.

Ciphers use a mathematical algorithm to encrypt and decrypt data. Some of the widely known cipher algorithms are AES, DES, and RC4.

Streams, Blocks, Padding, and Modes

A *stream cipher* encrypts one byte at a time. A *block cipher* encrypts a block of bytes in a single operation.

Because plaintext is not always an even multiple of the block size, *padding* may be added to the plaintext. Padding is extra bytes of data that fill up the empty space in a block. Several standard padding schemes describe what values are placed in the padding bytes.

Block ciphers can operate in different *modes*. A mode describes exactly how plaintext is transformed into ciphertext and back again. In the simplest mode, each block of plaintext is encrypted to one block of ciphertext. More complicated modes use some combination of the current plaintext block and previous ciphertext as the input to the cipher. Some modes require an *initialization vector*, which is combined with the very first block of plaintext as input to the cipher.

Working with Keys

Cryptography is based on mathematics. In essence, keys are numbers that are used in cryptographic algorithms. Different algorithms use different kinds of keys.

SATSA-CRYPTO provides three ways to think about a key:

- A key object is a high-level conceptual representation. In SATSA-CRYPTO, key objects implement the `java.security.Key` interface.

- The numbers that make up a key can be exposed through some subinterface of `java.security.spec.KeySpec`.
- An *encoded* key is a representation of a key as a byte stream.

Keys are typically stored in encoded form on some form of persistent storage. At runtime, the application loads a key as a byte array, then converts it into a key object that can be used with the rest of the API.

The way you convert byte stream representations of keys to key objects depends on the type of key.

For symmetric cipher keys, you can create an implementation of `KeySpec` using a byte array. If the `KeySpec` class implements the `Key` interface, the object can be used directly as a key. For example, the `SecretKeySpec` class can be used to create DES keys from byte arrays.

For asymmetric keys, a public key can be extracted by constructing a `KeySpec` implementation from a byte array. Then you can use a `KeyFactory` object to extract the public key from the `KeySpec`. An example later in this chapter illustrates the technique.

Using Ciphers in SATSA-CRYPTO

SATSA-CRYPTO encapsulates cryptographic ciphers in the `javax.crypto.Cipher` class. `Cipher` is surprisingly easy to use:

1. Create a `Cipher` instance for the algorithm you want to use. You can also specify a mode and a padding scheme in this step.
2. Initialize the `Cipher` with a key and a flag that indicates whether the `Cipher` will be encrypting or decrypting.
3. Feed data to the `Cipher` using the `update()` method.
4. Send the last data to the `Cipher` with the `doFinal()` method.

Rather than instantiate a `Cipher` directly, use a factory method, `getInstance()`, to request an instance that corresponds to a particular algorithm. Pass `getInstance()` an algorithm name or a combination of an algorithm name, mode, and padding. If the underlying implementation has a matching implementation, a new `Cipher` object is created and returned. Otherwise, a `NoSuchAlgorithmException` is thrown.

The following example demonstrates several of these techniques.

It begins by constructing a DES secret key from a byte array using a `SecretKeySpec`. It creates a `Cipher` based on the DES algorithm in Electronic Code Book (ECB) mode, with no padding. The cipher is initialized for encryption using the secret key. Finally, this example encrypts a small plaintext message. For a complete example that uses SATSA-CRYPTO, see Appendix D.

```
byte[] keyBits = {
  (byte) 0x2b, (byte) 0x7e, (byte) 0x15, (byte) 0x16,
  (byte) 0x28, (byte) 0xae, (byte) 0xd2, (byte) 0xa6
};
byte[] plaintext = {
  (byte)0x01, (byte)0x23, (byte)0x45,
  (byte)0x67, (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef
};
byte[] ciphertext = new byte[8];

SecretKeySpec key = new SecretKeySpec(keyBits, 0, keyBits.length,
    "DES");
Cipher cipher = Cipher.getInstance("DES/ECB/NoPadding");
cipher.init(Cipher.ENCRYPT_MODE, key);
int count = cipher.doFinal(plaintext, 0, plaintext.length,
    ciphertext, 0);
```

The next example is slightly more complicated. It uses a DES cipher in Cipher Block Chaining (CBC) mode, in which the previous block of ciphertext is combined with the current block of plaintext before encryption. For the first block of plaintext, there is no previous ciphertext block, so an initialization vector is needed. This example demonstrates how to use `IvParameterSpec` to pass an initialization vector to a `Cipher` using one of its `init()` methods.

```
byte[] keyBits = {
  (byte)0x2b, (byte)0x7e, (byte)0x15, (byte)0x16,
  (byte)0x28, (byte)0xae, (byte)0xd2, (byte)0xa6
};
byte[] ivBits =  {
  (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04,
  (byte)0x05, (byte)0x06, (byte)0x07, (byte)0x08
};
byte[] plaintext = {
  (byte)0x01, (byte)0x23, (byte)0x45,
  (byte)0x67, (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef
};
byte[] ciphertext = new byte[8];

SecretKeySpec key = new SecretKeySpec(keyBits, 0, keyBits.length,
    "DES");
IvParameterSpec iv = new IvParameterSpec(ivBits, 0, ivBits.length);
Cipher cipher = Cipher.getInstance("DES/CBC/NoPadding");
cipher.init(Cipher.ENCRYPT_MODE, key, iv);
int count = cipher.doFinal(plaintext, 0, plaintext.length,
    ciphertext, 0);
```

The final example shows how to encrypt data using the public key of an RSA key pair. It demonstrates the use of the `KeyFactory` class to extract a public key from an encoded representation.

```
byte[] publicKeyBits;
// Assign publicKeyBits here.
byte[] plaintext = "This is just an example".getBytes();
byte[] ciphertext = new byte[128];
X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKeyBits);
KeyFactory kf = KeyFactory.getInstance("RSA");
PublicKey publicKey = keyFactory.generatePublic(keySpec);
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, publicKey);
int count = cipher.doFinal(plaintext, 0, plaintext.length,
    ciphertext, 0);
```

Message Digests and Signatures

A message digest is a short value that is constructed by processing an arbitrarily large set of data. It serves as a fingerprint of the data. If you change the data set, even a single bit, then compute the message digest value again, it will be different. Message digests are helpful for ensuring that data has not changed in transit from one place to another. You could use a message digest to check for communications errors, but if you’re worried about attackers, use a digital signature to verify data integrity.

Conceptually, a digital signature is a message digest value that has been encrypted using the private key of a key pair. Why is this useful?

Suppose you want to send this message to your friend:

> Meet me at the theater at 10:30.

If you send this message via the Internet, your enemy could intercept the message in a dozen different ways and modify it at will, like this:

> Meet me at the restaurant at 7:30.

You could try using a message digest to prevent this kind of tampering. Create a message digest value from the original message, then send the message and the digest value together. Unfortunately, your enemy is pretty sharp and could simply change the message, compute a new digest value from the new message, and send both to your friend. Your friend would see the message and its matching digest and assume the message is correct.

If you digitally sign the message, your enemy cannot change the message undetected. When you create a signature of the message, you encrypt the message digest value with your private key, which is a secret. When your friend receives the message, she uses your public key to decrypt the message, then compares the decrypted digest value with a freshly computed one. If they match, your friend is sure that the message is unaltered.

Your enemy could create a new message and compute a new digest value for it, but he cannot forge your signature because he does not know your private key.

Using Message Digests in SATSA-CRYPTO

Message digests are represented by the `java.security.MessageDigest` class in SATSA-CRYPTO. The class is easy to use:

1. Instantiate a `MessageDigest` object by passing an algorithm name to the `getInstance()` factory method. (This follows the pattern for `Cipher` and, as you'll see later, `Signature`.)
2. Process the data that is to be digested by passing it to the `update()` method.
3. Retrieve the message digest value with the `digest()` method.

The following example shows how to calculate a SHA digest value. For a complete example, see Appendix D. Different digest algorithms generate differently sized digest values. The SHA algorithm produces a 20 byte value.

```
byte[] data = "abc".getBytes();
byte[] digest = new byte[20];
MessageDigest md = MessageDigest.getInstance("SHA-1");
md.update(data, 0, data.length);
md.digest(digest, 0, 20);
```

Verifying Signatures in SATSA-CRYPTO

Digital signatures are represented by the `java.security.Signature` class. In SATSA-CRYPTO, `Signature` can be used only to verify signatures, not to generate them. `Signature` resembles `MessageDigest` except that it is initialized like `Cipher`. Follow these steps to use a `Signature`:

1. Create a `Signature` object by passing an algorithm name to the `getInstance()` factory method.
2. Initialize the `Signature` by passing a public key to `initVerify()`.
3. Process the data that is to be digested by passing it to the `update()` method.
4. Pass a previously calculated signature value to `verify()`. The `Signature` calculates its own signature value and compares it to the one you supplied. If the two signatures match, the method returns `true`.

The following example shows how to verify an RSA signature. For a complete example, see Appendix D.

```
byte[] publicKeyBits;
// Assign publicKeyBits here.
X509EncodedKeySpec pks = new X509EncodedKeySpec(publicKeyBits);
KeyFactory kf = KeyFactory.getInstance("RSA");
PublicKey publicKey = kf.generatePublic(pks);

Signature signature = Signature.getInstance("SHA1withRSA");
signature.initVerify(publicKey);
signature.update(kData, 0, kData.length);

boolean pass = signature.verify(kSignature);
```

APPENDIX **A**

SATSA-APDU Examples

This Appendix contains complete examples of the SATSA-APDU API. APDUMIDlet makes several APDU connections to the Java Card platform simulator (cref). SATMIDlet connects to the card and sends a SAT envelope.

The APDUMIDlet Example

To run the APDUMIDlet example, first run the Java Card platform simulator. When APDUMIDlet attempts to connect to a smart card, it does so by making local socket connections. In addition, APDUMIDlet expects to connect to specification card applets that must be running on the simulator.

Preparing the Java Card Platform Simulator

APDUMIDlet connects to example card applets that are distributed with the Java Card Development Kit. The first step is to create a card EEPROM image that contains the required applets.

To do this, run the card simulator, specifying that its EEPROM image should be saved. In this example, the file name is demo2.eeprom:

```
start cref -o demo2.eeprom
```

In DOS, start launches cref in a separate window. The next step is to load some of the Java Card Development Kit example applets onto the card. This example assumes that the Java Card Development Kit is installed in \java_card_kit-2_2_1.

```
apdutool /java_card_kit-2_2_1/samples/src/demo/demo2.scr
```

After this step is complete, cref exits, saving its EEPROM image in the file you specified. You only need to perform this step once.

When you have the EEPROM image file, run two instances of `cref` as follows:

```
start cref -p 9025 -i demo2.eeprom
start cref -p 9026 -i demo2.eeprom
```

When you run `APDUMIDlet`, the SATSA RI is able to find two simulated cards that are running the `demo2` card applets.

Source Code

```
/*
 * Copyright © 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

import java.io.*;
import javax.microedition.io.Connector;
import javax.microedition.apdu.APDUConnection;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class APDUMIDlet
    extends MIDlet
    implements CommandListener, Runnable {
  private final String kPurseSlot0 =
   "apdu:0;target=a0.00.00.00.62.03.01.0c.02.01";
  private final String kPurseSlot1 =
   "apdu:1;target=a0.00.00.00.62.03.01.0c.02.01";
  private final String kWalletSlot0 =
   "apdu:0;target=a0.00.00.00.62.03.01.0c.06.01";

  private APDUConnection mPurseConnection0;
  private APDUConnection mPurseConnection1;
  private APDUConnection mWalletConnection;

  private Display mDisplay;
  private Form mMainForm;
  private Command mExitCommand, mGoCommand, mBackCommand;
  private Form mProgressForm;

  public APDUMIDlet() {
    mExitCommand = new Command("Exit", Command.EXIT, 0);
    mGoCommand = new Command("Go", Command.SCREEN, 0);
    mBackCommand = new Command("Back", Command.BACK, 0);

    mMainForm = new Form("APDU Example");
    mMainForm.append("Press Go to use the SATSA-APDU API " +
        "to connect to card applications.");
    mMainForm.addCommand(mExitCommand);
    mMainForm.addCommand(mGoCommand);
    mMainForm.setCommandListener(this);
```

```
}

public void startApp() {
  mDisplay = Display.getDisplay(this);

  mDisplay.setCurrent(mMainForm);
}

public void pauseApp() {}

public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable s) {
  if (c == mExitCommand) {
    notifyDestroyed();
  }
  else if (c == mGoCommand) {
    mProgressForm = new Form("Working...");
    mDisplay.setCurrent(mProgressForm);

    Thread t = new Thread(this);
    t.start();
  }
  else if (c == mBackCommand) {
    mDisplay.setCurrent(mMainForm);
  }
}

public void run() {
  try {
    setProgress("Opening purse slot 0");
    mPurseConnection0 =
 (APDUConnection)Connector.open(kPurseSlot0);
    setProgress("Opening purse slot 1");
    mPurseConnection1 =
 (APDUConnection)Connector.open(kPurseSlot1);
    setProgress("Opening wallet slot 0");
    mWalletConnection =
 (APDUConnection)Connector.open(kWalletSlot0);

    setProgress("Exchanging APDUs with purse slot 0");
    for (int i = 0; i < kPurseAPDUs.length; i++) {
      byte[] apdu = kPurseAPDUs[i];
      byte[] response = mPurseConnection0.exchangeAPDU(apdu);
      // Process response.
    }

    setProgress("Closing purse slot 0");
    mPurseConnection0.close();

    setProgress("Exchanging APDUs with wallet");
    for (int i = 0; i < kWalletAPDUs.length; i++) {
        byte[] apdu = kWalletAPDUs[i];
        byte[] response = mWalletConnection.exchangeAPDU(apdu);
        // Process response.
      }

      setProgress("Closing wallet");
      mWalletConnection.close();

      setProgress("Exchanging APDUs with purse slot 1");
      for (int i = 0; i < kPurseAPDUs.length; i++) {
        byte[] apdu = kPurseAPDUs[i];
        byte[] response = mPurseConnection1.exchangeAPDU(apdu);
        // Process response.
      }

      setProgress("Closing purse slot 1");
      mPurseConnection1.close();

      mProgressForm.setTitle("Working...done.");
      mProgressForm.addCommand(mBackCommand);
      mProgressForm.setCommandListener(this);
    }
    catch (Exception e) {
      try { mPurseConnection0.close(); } catch (Throwable t) {}
      try { mPurseConnection1.close(); } catch (Throwable t) {}
      try { mWalletConnection.close(); } catch (Throwable t) {}

      Form f = new Form("Exception");
      f.append(e.toString());
      f.addCommand(mBackCommand);
      f.setCommandListener(this);
      mDisplay.setCurrent(f);
    }
  }

  private void setProgress(String s) {
    StringItem si = new StringItem(null, s);
    si.setLayout(Item.LAYOUT_2 | Item.LAYOUT_NEWLINE_AFTER);
    mProgressForm.append(si);
  }

  private final byte[][] kPurseAPDUs = {
    // Verify PIN (User PIN 01020304)
    {(byte)0x00, (byte)0x20, (byte)0x00, (byte)0x82, (byte)0x04,
     (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x00},
    // should return 90 00

    // Initialize Transaction: Credit $250.00
    {(byte)0x80, (byte)0x20, (byte)0x01, (byte)0x00, (byte)0x0a,
     (byte)0x61, (byte)0xa8, (byte)0x22, (byte)0x44, (byte)0x66,
     (byte)0x88, (byte)0x00, (byte)0x00, (byte)0x00,
     (byte)0x00, (byte)0x7F},
      // 05 05 05 05 0c 1f 62 00 00 00 07 00 00 00 00 00 00 00 00 90 00
      // = Purse ID : 0x05050505; ExpDate 12/31/98; TN=7

      // Complete Transaction: Date 10/27/97; Time 15:33
      {(byte)0x80, (byte)0x22, (byte)0x00, (byte)0x00, (byte)0x0d,
       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x0a,
       (byte)0x1b, (byte)0x61, (byte)0x0f, (byte)0x21, (byte)0x7F},
      // 61 a8 00 00 00 00 00 00 00 00 90 00= Purse Balance $250.00;

      // Initialize Transaction: Debit $25.00;
      {(byte)0x80, (byte)0x20, (byte)0x02, (byte)0x00, (byte)0x0a,
       (byte)0x09, (byte)0xc4, (byte)0x22, (byte)0x44, (byte)0x66,
       (byte)0x88, (byte)0x00, (byte)0x00, (byte)0x00,
       (byte)0x00, (byte)0x7F},
      // 05 05 05 05 0c 1f 62 61 a8 00 08 00 00 00 00 00 00 00 00 90 00;
      // = Purse ID : 0x05050505; ExpDate 12/31/98; TN=8

      // Complete Transaction: Date 10/27/97; Time 15:35
      {(byte)0x80, (byte)0x22, (byte)0x00, (byte)0x00, (byte)0x0d,
       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
       (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x0a,
       (byte)0x1b, (byte)0x61, (byte)0x0f, (byte)0x23, (byte)0x7F}
      // 57 e4 00 00 00 00 00 00 00 00 90 00= Purse Balance $225.00;
    };

    public final byte[][] kWalletAPDUs = {
      // verify PIN expects 90 00
      {(byte)0x80, (byte)0x20, (byte)0x00, (byte)0x00, (byte)0x05,
       (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x05,
       (byte)0x7F},

      // get wallet balance
      {(byte)0x80, (byte)0x50, (byte)0x00, (byte)0x00, (byte)0x00,
       (byte)0x02},

      // credit 128 to the wallet
      {(byte)0x80, (byte)0x30, (byte)0x00, (byte)0x00, (byte)0x01,
       (byte)0x80, (byte)0x7F},

      // get wallet balance
      {(byte)0x80, (byte)0x50, (byte)0x00, (byte)0x00, (byte)0x00,
       (byte)0x02}
    };
  }
```

The `SATMIDlet` Example

`SATMIDlet` connects to a card applet and sends a SAT envelope.

Setting Up the Java Card Platform Simulator

The first step (as with `APDUMIDlet`) is to create a card EEPROM image that contains the required applet. Run the card simulator, specifying that its EEPROM image should be saved. In this example, the file name is `sat.eeprom`:

```
start cref -o sat.eeprom
```

Next, load the necessary applet onto the card. This applet is distributed with the SATSA RI in the file `javacard_classes\jc_script\satsimulator.scr`. Assuming the SATSA RI is installed in `\satsa1.0`, you can install the applet with this command:

```
apdutool /satsa1.0/javacard_classes/jc_script/satsimulator.scr
```

After this step is complete, `cref` exits, saving its EEPROM image in the file you specified. You only need to perform this step once.

When you have the EEPROM image file, run an instance of `cref`:

```
start cref -i sat.eeprom
```

When you run `SATMIDlet`, the SATSA RI is able to find the simulated card that is running the `satsimulator` card applet.

Source Code

```
/*
 * Copyright © 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

import java.io.*;
import javax.microedition.io.Connector;
import javax.microedition.apdu.APDUConnection;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SATMIDlet
    extends MIDlet
    implements CommandListener, Runnable {
  private APDUConnection mSATConnection1;
  private APDUConnection mSATConnection2;

  private Display mDisplay;
private Form mMainForm;
private Command mExitCommand, mGoCommand, mBackCommand;
private Form mProgressForm;

public SATMIDlet() {
  mExitCommand = new Command("Exit", Command.EXIT, 0);
  mGoCommand = new Command("Go", Command.SCREEN, 0);
  mBackCommand = new Command("Back", Command.BACK, 0);

  mMainForm = new Form("SAT Example");
  mMainForm.append("Press Go to use the SATSA-APDU API " +
      "to connect to a SAT application.");
  mMainForm.addCommand(mExitCommand);
  mMainForm.addCommand(mGoCommand);
  mMainForm.setCommandListener(this);
}

public void startApp() {
  mDisplay = Display.getDisplay(this);

  mDisplay.setCurrent(mMainForm);
}

public void pauseApp() {}

public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable s) {
  if (c == mExitCommand) {
    notifyDestroyed();
  }
  else if (c == mGoCommand) {
    mProgressForm = new Form("Working...");
    mDisplay.setCurrent(mProgressForm);

    Thread t = new Thread(this);
    t.start();
  }
  else if (c == mBackCommand) {
    mDisplay.setCurrent(mMainForm);
  }
}

public void run() {
  try {
    String url = "apdu:0;target=SAT";

    setProgress("Opening first SAT connection");
    mSATConnection1 = (APDUConnection)Connector.open(url);
    // The second connection demonstrates that two SAT
    // connections can be open simultaneously.
    setProgress("Opening second SAT connection");
```

```
        mSATConnection2 = (APDUConnection)Connector.open(url);

        setProgress("Sending envelopes");
        byte[] response1 = mSATConnection1.exchangeAPDU(kEnvelope);
        byte[] response2 = mSATConnection2.exchangeAPDU(kEnvelope);

        setProgress("Closing first SAT connection");
        mSATConnection1.close();

        setProgress("Closing second SAT connection");
        mSATConnection2.close();

        mProgressForm.setTitle("Working...done.");
        mProgressForm.addCommand(mBackCommand);
        mProgressForm.setCommandListener(this);
      }
      catch (Exception e) {
        try { mSATConnection1.close(); } catch (Throwable t) {}
        try { mSATConnection2.close(); } catch (Throwable t) {}

        Form f = new Form("Exception");
        f.append(e.toString());
        f.addCommand(mBackCommand);
        f.setCommandListener(this);
        mDisplay.setCurrent(f);
      }
    }

    private void setProgress(String s) {
      StringItem si = new StringItem(null, s);
      si.setLayout(Item.LAYOUT_2 | Item.LAYOUT_NEWLINE_AFTER);
      mProgressForm.append(si);
    }

    // Update the binary file EFpuct (Price per unit and currency table)
    private final byte[] kEnvelope = {
      (byte)0xA0, (byte)0xC2, (byte)0x00, (byte)0x00, (byte)0x2f,
      (byte)0xD1, (byte)0x2d, (byte)0x82, (byte)0x02, (byte)0x83,
      (byte)0x81, (byte)0x06, (byte)0x05, (byte)0x80, (byte)0x11,
      (byte)0x22, (byte)0x33, (byte)0x44, (byte)0x8B, (byte)0x20,
      (byte)0x10, (byte)0x02, (byte)0x81, (byte)0x55, (byte)0x7F,
      (byte)0xF6, (byte)0x00, (byte)0x11, (byte)0x29, (byte)0x12,
      (byte)0x00, (byte)0x00, (byte)0x04, (byte)0x14, (byte)0x10,
      (byte)0x11, (byte)0x12, (byte)0x13, (byte)0x14, (byte)0x15,
      (byte)0x16, (byte)0x17, (byte)0x18, (byte)0x19, (byte)0x1a,
      (byte)0x1b, (byte)0x1c, (byte)0x1d, (byte)0x41, (byte)0xAB,
      (byte)0xCD, (byte)0xEF
    };
  }
```

APPENDIX **B**

SATSA-JCRMI Example

This Appendix contains an example that demonstrates the use of the SATSA-JCRMI API.

Preparing the Java Card Platform Simulator

`JCRMIMIDlet` connects to an example card applet that is distributed with the Java Card Development Kit. If you've already worked through the `APDUMIDlet` example, note that `JCRMIMIDlet` uses the same `demo2` card applets as `APDUMIDlet`. The first step is to create a card EEPROM image that contains the required applets.

To do this, run the card simulator, specifying that its EEPROM image should be saved. In this example, the file name is `demo2.eeprom`:

```
start cref -o demo2.eeprom
```

In DOS, `start` launches `cref` in a separate window. The next step is to load some of the Java Card Development Kit example applets onto the card. This example assumes that the Java Card Development Kit is installed in `\java_card_kit-2_2_1`.

```
apdutool /java_card_kit-2_2_1/samples/src/demo/demo2.scr
```

After this step is complete, `cref` exits, saving its EEPROM image in the file you specified. You only need to perform this step once.

When you have the EEPROM image file, run `cref` as follows:

```
start cref -p 9025 -i demo2.eeprom
```

When you run `JCRMIMIDlet`, the SATSA RI is able to find the simulated card that is running the `demo2` card applets.

Source Code

The source code for this example is contained in two files, `APDUMIDlet.java` and `com\sun\javacard\samples\RMIDemo\Purse.java`. Use the `jcrmic` tool, described in Chapter 4, to generate a stub for the remote interface `Purse`.

`JCRMIMIDlet` Source Code

```
/*
 * Copyright © 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

import java.io.*;
import javax.microedition.io.Connector;
import javax.microedition.jcrmi.JavaCardRMIConnection;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

import com.sun.javacard.samples.RMIDemo.Purse;

public class JCRMIMIDlet
    extends MIDlet
    implements CommandListener, Runnable {
  private final String kRMIURL =
   "jcrmi:0;AID=a0.0.0.0.62.3.1.c.8.1";

  private JavaCardRMIConnection mConnection;

  private Display mDisplay;
  private Form mMainForm;
  private Command mExitCommand, mGoCommand, mBackCommand;
  private Form mProgressForm;

  public JCRMIMIDlet() {
    mExitCommand = new Command("Exit", Command.EXIT, 0);
    mGoCommand = new Command("Go", Command.SCREEN, 0);
    mBackCommand = new Command("Back", Command.BACK, 0);

    mMainForm = new Form("JCRMI Example");
    mMainForm.append("Press Go to use the SATSA-JCRMI API " +
        "to connect to a card application.");
    mMainForm.addCommand(mExitCommand);
    mMainForm.addCommand(mGoCommand);
    mMainForm.setCommandListener(this);
  }

  public void startApp() {
    mDisplay = Display.getDisplay(this);

    mDisplay.setCurrent(mMainForm);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}

  public void commandAction(Command c, Displayable s) {
    if (c == mExitCommand) {
      notifyDestroyed();
    }
    else if (c == mGoCommand) {
      mProgressForm = new Form("Working...");
      mDisplay.setCurrent(mProgressForm);

      Thread t = new Thread(this);
      t.start();
    }
    else if (c == mBackCommand) {
      mDisplay.setCurrent(mMainForm);
    }
  }

  public void run() {
    try {
      setProgress("Opening a connection");
      mConnection = (JavaCardRMIConnection)Connector.open(kRMIURL);

      setProgress("Getting an initial reference");
      Purse purse = (Purse)mConnection.getInitialReference();

      short balance = purse.getBalance();
      setProgress("Balance = " + balance);

      setProgress("Crediting 20");
      purse.credit((short)20);
      setProgress("Debiting 15");
      purse.debit((short)15);

      balance = purse.getBalance();
      setProgress("Balance = " + balance);

      setProgress("Setting account number to 54321");
      purse.setAccountNumber(new byte[] {5, 4, 3, 2, 1});

      setProgress("Closing connection");
      mConnection.close();

      mProgressForm.setTitle("Working...done.");
      mProgressForm.addCommand(mBackCommand);
      mProgressForm.setCommandListener(this);
    }
    catch (Exception e) {
      try { mConnection.close(); } catch (Throwable t) {}

      Form f = new Form("Exception");
      f.append(e.toString());
      f.addCommand(mBackCommand);
      f.setCommandListener(this);
      mDisplay.setCurrent(f);
    }
  }

  private void setProgress(String s) {
    StringItem si = new StringItem(null, s);
    si.setLayout(Item.LAYOUT_2 | Item.LAYOUT_NEWLINE_AFTER);
    mProgressForm.append(si);
  }
}
```

`Purse` Source Code

```
/*
 * Copyright © 2002 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

package com.sun.javacard.samples.RMIDemo;

import java.rmi.*;
import javacard.framework.*;

public interface Purse extends Remote {
  public static final short UNDERFLOW = (short)0x6000;
  public static final short OVERFLOW  = (short)0x6001;

  public static final short BAD_ARGUMENT  = (short)0x6002;

  public static final short MAX_AMOUNT = (short)400; // for whatever
    reason

  public short getBalance() throws RemoteException;
  public void debit(short m) throws RemoteException, UserException;
  public void credit(short m) throws RemoteException, UserException;

  public void setAccountNumber(byte[] number)
      throws RemoteException, UserException;
  public byte[] getAccountNumber() throws RemoteException;
}
```

APPENDIX C

SATSA-PKI Examples

The following examples are copied from the JSR 177 specification.

This example shows how to use `CMSMessageSignatureService` to generate signatures:

```
String caName =
    new String("cn=ca_name,ou=ou_name,o=org_name,c=ie");
String[] caNames = new String[1];
String stringToSign = new String("JSR 177 Approved");
String userPrompt = new String("Please insert the security element "
                               + "issued by bank ABC"
                               + "for the application XYZ.");
byte[] byteArrayToSign = new byte[8];
byte[] authSignature;
byte[] signSignature;

caNames[0] = caName;

try {
    // Generate a formatted authentication signature that includes
  the
    // content that was signed in addition to the certificate.
    // Selection of the key is implicit in selection of the
  certificate,
    // which is selected through the caNames parameter.
    // If the appropriate key is not found in any of the security
    // elements present in the device, the implementation may guide
    // the user to insert an alternative security element using
    // the securityElementPrompt parameter.
    authSignature = CMSMessageSignatureService.authenticate(
                 byteArrayToSign,
                CMSMessageSignatureService.SIG_INCLUDE_CERTIFICATE
                 |CMSMessageSignatureService.SIG_INCLUDE_CONTENT,
                 caNames, userPrompt);

    // Generate a formatted signature that includes the
    // content that was signed in addition to the certificate.
    // Selection of the key is implicit in selection of the
  certificate,
    // which is selected through the caNames parameter.
    // If the appropriate key is not found in any of the
    // security elements present in the device, the implementation
    // may guide the user to insert an alternative
    // security element using the securityElementPrompt parameter.
    signSignature = CMSMessageSignatureService.sign(
                 stringToSign,
                CMSMessageSignatureService.SIG_INCLUDE_CERTIFICATE
                 |CMSMessageSignatureService.SIG_INCLUDE_CONTENT,
                 caNames, userPrompt);
    } catch (IllegalArgumentException iae) {
        // Perform error handling
        iae.printStackTrace();
    } catch (CMSMessageSignatureServiceException ce) {
        if (ce.getReason() == ce.CRYPTO_FORMAT_ERROR) {
            System.out.println("Error formatting signature.");
        } else {
            System.out.println(ce.getMessage());
        }
    }
```

The following example demonstrates the use of `UserCredentialManager`.

```
// Parameters for certificate request message.
String nameInfo = new String("CN=User Name");
byte[] enrollmentRequest = null;
int keyLength = 1024;

// User friendly names and prompts.
String securityElementID = new String("Bank XYZ");
String securityElementPrompt = new String
  ("Please insert bank XYZ security element before proceeding");
String friendlyName = new String("My Credential");

// Certificate chain and URI from registration response.
byte[] pkiPath;
String uri;


// Obtain a certificate enrollment request message.
try {
    enrollmentRequest = UserCredentialManager.generateCSR
     (nameInfo, UserCredentialManager.ALGORITHM_RSA, keyLength,
        UserCredentialManager.KEY_USAGE_NON_REPUDIATION,
        securityElementID, securityElementPrompt, false);

    // Send it to a registration server.
     ...
    // Assign values for pkipath and certificate uri
    // from the registration response.
     ...

    // Store the certificate on the security element.
    UserCredentialManager.addCredential(friendlyName,
                                    pkiPath, uri);
} catch (IllegalArgumentException iae) {
    iae.printStackTrace();
} catch (NullPointerException npe) {
    npe.printStackTrace();
} catch (CMSMessageSignatureServiceException cmse) {
    cmse.printStackTrace();
} catch (UserCredentialManagerException pkie) {
    pkie.printStackTrace();
}
```

APPENDIX **D**

SATSA-CRYPTO Example

```
/*
 * Copyright © 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class CryptoMIDlet
    extends MIDlet
    implements CommandListener, Runnable {
  private Display mDisplay;
  private Form mMainForm;
  private Command mExitCommand, mGoCommand, mBackCommand;
  private Form mProgressForm;

  public CryptoMIDlet() {
    mExitCommand = new Command("Exit", Command.EXIT, 0);
    mGoCommand = new Command("Go", Command.SCREEN, 0);
    mBackCommand = new Command("Back", Command.BACK, 0);

    mMainForm = new Form("Crypto Example");
    mMainForm.append("Press Go to use the SATSA-CRYPTO API " +
        "to perform cryptography.");
    mMainForm.addCommand(mExitCommand);
    mMainForm.addCommand(mGoCommand);
    mMainForm.setCommandListener(this);
  }

  public void startApp() {
    mDisplay = Display.getDisplay(this);

    mDisplay.setCurrent(mMainForm);
  }
```

```
public void pauseApp() {}

public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable s) {
  if (c == mExitCommand) {
    notifyDestroyed();
  }
  else if (c == mGoCommand) {
    mProgressForm = new Form("Working...");
    mDisplay.setCurrent(mProgressForm);

    Thread t = new Thread(this);
    t.start();
  }
  else if (c == mBackCommand) {
    mDisplay.setCurrent(mMainForm);
  }
}

public void run() {
  try {
    runDigest();

    runSignature();

    runCipher();

    byte[] nonBlockPlaintext =
        "This is not a block length.".getBytes();

    runCipherSymmetric("AES/ECB/NoPadding", kAESKey, "AES",
        null, kBlockPlaintext);
    runCipherSymmetric("AES/ECB/PKCS5Padding", kAESKey, "AES",
        null, nonBlockPlaintext);
    runCipherSymmetric("AES/CBC/NoPadding", kAESKey, "AES",
        kAESiv, kBlockPlaintext);
    runCipherSymmetric("AES/CBC/PKCS5Padding", kAESKey, "AES",
        kAESiv, nonBlockPlaintext);

    runCipherSymmetric("DES/ECB/NoPadding", kDESKey, "DES",
        null, kBlockPlaintext);
    runCipherSymmetric("DES/ECB/PKCS5Padding", kDESKey, "DES",
        null, nonBlockPlaintext);
    runCipherSymmetric("DES/CBC/NoPadding", kDESKey, "DES",
        kDESiv, kBlockPlaintext);
    runCipherSymmetric("DES/CBC/PKCS5Padding", kDESKey, "DES",
        kDESiv, nonBlockPlaintext);

    runCipherSymmetric("DESede/ECB/NoPadding", kDESEDEKey,
        "DESede", null, kBlockPlaintext);
    runCipherSymmetric("DESede/ECB/PKCS5Padding", kDESEDEKey,
        "DESede", null, nonBlockPlaintext);
    runCipherSymmetric("DESede/CBC/NoPadding", kDESEDEKey,
        "DESede", kDESiv, kBlockPlaintext);
    runCipherSymmetric("DESede/CBC/PKCS5Padding", kDESEDEKey,
        "DESede", kDESiv, nonBlockPlaintext);

    mProgressForm.setTitle("Working...done.");
    mProgressForm.addCommand(mBackCommand);
    mProgressForm.setCommandListener(this);
  }
  catch (Exception e) {
    Form f = new Form("Exception");
    f.append(e.toString());
    f.addCommand(mBackCommand);
    f.setCommandListener(this);
    mDisplay.setCurrent(f);
  }
}

private void runDigest()
    throws NoSuchAlgorithmException, DigestException {
  setProgress("Generating SHA-1 digest");

  byte[] data = "abc".getBytes();
  byte[] digest = new byte[20];
  MessageDigest md = MessageDigest.getInstance("SHA-1");
  md.update(data, 0, data.length);
  md.digest(digest, 0, 20);

  boolean pass = true;
  for (int i = 0; i < 20; i++) {
    if (digest[i] != kDigest[i])
      pass = false;
  }
  setProgress("SHA1 digest..." + (pass ? "pass" : "fail"));
}

private void runSignature()
    throws NoSuchAlgorithmException, InvalidKeySpecException,
           InvalidKeyException, SignatureException {
  X509EncodedKeySpec pks = new X509EncodedKeySpec(kRSAPublicKey);
  KeyFactory kf = KeyFactory.getInstance("RSA");
  PublicKey publicKey = kf.generatePublic(pks);

  Signature signature = Signature.getInstance("SHA1withRSA");
  signature.initVerify(publicKey);
  signature.update(kData, 0, kData.length);

  boolean pass = signature.verify(kSignature);
  setProgress("RSA signature..." + (pass ? "" : "not ") +
 "verified");
```

```
  }

  private void runCipher()
      throws NoSuchAlgorithmException, InvalidKeySpecException,
             NoSuchPaddingException, InvalidKeyException,
             IllegalStateException, ShortBufferException,
             IllegalBlockSizeException, BadPaddingException {
    X509EncodedKeySpec pks = new X509EncodedKeySpec(kRSAPublicKey);
    KeyFactory kf = KeyFactory.getInstance("RSA");
    PublicKey publicKey = kf.generatePublic(pks);

    byte[] ciphertext = new byte[512];
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, publicKey);
    cipher.doFinal(kData, 0, kData.length, ciphertext, 0);

    setProgress("RSA encryption...done");
  }

  private void runCipherSymmetric(String algorithm,
      byte[] keyBits, String keyAlgorithm,
      byte[] ivBits, byte[] plaintext)
      throws NoSuchAlgorithmException, InvalidKeySpecException,
             NoSuchPaddingException, InvalidKeyException,
             IllegalStateException, ShortBufferException,
             IllegalBlockSizeException, BadPaddingException,
             InvalidAlgorithmParameterException {
    Cipher cipher = Cipher.getInstance(algorithm);
    Key key = new SecretKeySpec(keyBits,
        0, keyBits.length, keyAlgorithm);
    IvParameterSpec iv = null;
    if (ivBits != null)
      iv = new IvParameterSpec(ivBits, 0, ivBits.length);

    // Calculate ciphertext size.
    int blocksize = 16;
    int ciphertextLength = 0;
    int remainder = plaintext.length % blocksize;
    if (remainder == 0)
      ciphertextLength = plaintext.length;
    else
      ciphertextLength = plaintext.length - remainder + blocksize;

    if (iv == null)
      cipher.init(Cipher.ENCRYPT_MODE, key);
    else
      cipher.init(Cipher.ENCRYPT_MODE, key, iv);
    byte[] ciphertext = new byte[ciphertextLength];
    cipher.doFinal(plaintext, 0, plaintext.length, ciphertext, 0);

    if (iv == null)
      cipher.init(Cipher.DECRYPT_MODE, key);
  else
    cipher.init(Cipher.DECRYPT_MODE, key, iv);
  byte[] decrypted = new byte[plaintext.length];
  cipher.doFinal(ciphertext, 0, ciphertext.length, decrypted, 0);

  boolean pass = compareArrays(plaintext, decrypted);
  setProgress(algorithm + "..." + (pass ? "passed" : "failed"));
}

private boolean compareArrays(byte[] array1, byte[] array2) {
  if (array1.length != array2.length) return false;

  for (int i = 0; i < array1.length; i++) {
    if (array1[i] != array2[i]) {
      return false;
    }
  }

  return true;
}

private void setProgress(String s) {
  StringItem si = new StringItem(null, s);
  si.setLayout(Item.LAYOUT_2 | Item.LAYOUT_NEWLINE_AFTER);
  mProgressForm.append(si);
}

private static final byte[] kDigest = {
  (byte) 0xA9, (byte) 0x99, (byte) 0x3E, (byte) 0x36, (byte) 0x47,
  (byte) 0x06, (byte) 0x81, (byte) 0x6A, (byte) 0xBA, (byte) 0x3E,
  (byte) 0x25, (byte) 0x71, (byte) 0x78, (byte) 0x50, (byte) 0xC2,
  (byte) 0x6C, (byte) 0x9C, (byte) 0xD0, (byte) 0xD8, (byte) 0x9D
};

private static final byte[] kRSAPublicKey = {
  (byte) 0x30, (byte) 0x82, (byte) 0x1,  (byte) 0x22,
  (byte) 0x30, (byte) 0xd,  (byte) 0x6,  (byte) 0x9,
  (byte) 0x2a, (byte) 0x86, (byte) 0x48, (byte) 0x86,
  (byte) 0xf7, (byte) 0xd,  (byte) 0x1,  (byte) 0x1,
  (byte) 0x1,  (byte) 0x5,  (byte) 0x0,  (byte) 0x3,
  (byte) 0x82, (byte) 0x1,  (byte) 0xf,  (byte) 0x0,
  (byte) 0x30, (byte) 0x82, (byte) 0x1,  (byte) 0xa,
  (byte) 0x2,  (byte) 0x82, (byte) 0x1,  (byte) 0x1,
  (byte) 0x0,  (byte) 0xe0, (byte) 0xe2, (byte) 0x9f,
  (byte) 0xc2, (byte) 0x75, (byte) 0x4c, (byte) 0x10,
  (byte) 0x53, (byte) 0xbb, (byte) 0x48, (byte) 0xcb,
  (byte) 0x54, (byte) 0x23, (byte) 0xe4, (byte) 0x91,
  (byte) 0x17, (byte) 0xa2, (byte) 0xec, (byte) 0x59,
  (byte) 0x9f, (byte) 0x6f, (byte) 0x57, (byte) 0x7f,
  (byte) 0x9b, (byte) 0x6a, (byte) 0x1f, (byte) 0x93,
  (byte) 0x5e, (byte) 0x69, (byte) 0xf1, (byte) 0xd4,
  (byte) 0x56, (byte) 0xb9, (byte) 0x65, (byte) 0x9e,
(byte) 0x14, (byte) 0x27, (byte) 0xb8, (byte) 0xb1,
(byte) 0xb5, (byte) 0x9d, (byte) 0xea, (byte) 0xd6,
(byte) 0xef, (byte) 0xc2, (byte) 0x3,  (byte) 0x4e,
(byte) 0x9b, (byte) 0x28, (byte) 0x1e, (byte) 0x1b,
(byte) 0x8,  (byte) 0x1a, (byte) 0x5,  (byte) 0x4d,
(byte) 0xf7, (byte) 0xb5, (byte) 0xe7, (byte) 0x92,
(byte) 0xcd, (byte) 0x3a, (byte) 0x59, (byte) 0xd8,
(byte) 0xb6, (byte) 0xb6, (byte) 0x20, (byte) 0xf3,
(byte) 0xc8, (byte) 0x2b, (byte) 0xf8, (byte) 0x1e,
(byte) 0x38, (byte) 0xd9, (byte) 0xb4, (byte) 0xf4,
(byte) 0x23, (byte) 0xc0, (byte) 0x3,  (byte) 0xc9,
(byte) 0x2,  (byte) 0x71, (byte) 0x7a, (byte) 0xac,
(byte) 0x40, (byte) 0x25, (byte) 0x67, (byte) 0xfe,
(byte) 0xc2, (byte) 0x6a, (byte) 0xd2, (byte) 0x3b,
(byte) 0x25, (byte) 0x14, (byte) 0x29, (byte) 0xf5,
(byte) 0x99, (byte) 0x8c, (byte) 0xef, (byte) 0x51,
(byte) 0x25, (byte) 0xa4, (byte) 0x37, (byte) 0xda,
(byte) 0xb1, (byte) 0x65, (byte) 0xb6, (byte) 0x49,
(byte) 0xf7, (byte) 0x9d, (byte) 0x1e, (byte) 0x5a,
(byte) 0x34, (byte) 0xe,  (byte) 0x17, (byte) 0xf2,
(byte) 0x50, (byte) 0x92, (byte) 0x85, (byte) 0xbb,
(byte) 0x1c, (byte) 0x6c, (byte) 0xae, (byte) 0x6a,
(byte) 0xe4, (byte) 0xe0, (byte) 0x29, (byte) 0xe5,
(byte) 0xfd, (byte) 0xcd, (byte) 0x10, (byte) 0x1a,
(byte) 0xab, (byte) 0x7,  (byte) 0xc7, (byte) 0xa4,
(byte) 0x32, (byte) 0xd7, (byte) 0xbd, (byte) 0x70,
(byte) 0x24, (byte) 0xc6, (byte) 0x53, (byte) 0x73,
(byte) 0x33, (byte) 0x95, (byte) 0x62, (byte) 0x84,
(byte) 0x99, (byte) 0xb5, (byte) 0x3b, (byte) 0x83,
(byte) 0x90, (byte) 0xe,  (byte) 0xbc, (byte) 0x91,
(byte) 0x58, (byte) 0xf0, (byte) 0x95, (byte) 0x96,
(byte) 0x15, (byte) 0xf,  (byte) 0xed, (byte) 0x68,
(byte) 0xba, (byte) 0x46, (byte) 0x5,  (byte) 0x22,
(byte) 0x99, (byte) 0x55, (byte) 0x1e, (byte) 0x39,
(byte) 0xbe, (byte) 0xf5, (byte) 0x34, (byte) 0xcd,
(byte) 0xb9, (byte) 0x43, (byte) 0xde, (byte) 0x1c,
(byte) 0xeb, (byte) 0xf0, (byte) 0x79, (byte) 0xee,
(byte) 0x9d, (byte) 0x60, (byte) 0xa5, (byte) 0x50,
(byte) 0x78, (byte) 0xe0, (byte) 0x38, (byte) 0xf9,
(byte) 0x28, (byte) 0x96, (byte) 0xaf, (byte) 0x7,
(byte) 0x99, (byte) 0xd6, (byte) 0xce, (byte) 0x7c,
(byte) 0xbc, (byte) 0x3b, (byte) 0x4,  (byte) 0xfd,
(byte) 0xd,  (byte) 0x9,  (byte) 0x70, (byte) 0xb1,
(byte) 0xad, (byte) 0xcf, (byte) 0xa5, (byte) 0x46,
(byte) 0xc8, (byte) 0x41, (byte) 0x5c, (byte) 0x7,
(byte) 0xd8, (byte) 0x9b, (byte) 0xcb, (byte) 0xd7,
(byte) 0xcb, (byte) 0x5c, (byte) 0xc4, (byte) 0x96,
(byte) 0xe,  (byte) 0x41, (byte) 0x84, (byte) 0x3b,
(byte) 0x28, (byte) 0x91, (byte) 0x7,  (byte) 0xc5,
(byte) 0xdc, (byte) 0x9e, (byte) 0x71, (byte) 0x78,
(byte) 0x10, (byte) 0x41, (byte) 0x8d, (byte) 0x5,
(byte) 0x3d, (byte) 0x36, (byte) 0x3f, (byte) 0x78,
    (byte) 0xa1, (byte) 0x9c, (byte) 0xb3, (byte) 0x37,
    (byte) 0x81, (byte) 0x2a, (byte) 0xa5, (byte) 0xd0,
    (byte) 0x25, (byte) 0xad, (byte) 0xfe, (byte) 0x71,
    (byte) 0x7,  (byte) 0x2,  (byte) 0x3,  (byte) 0x1,
    (byte) 0x0,  (byte) 0x1
};

private static final byte[] kData = {0, 1, 2, 3, 4};

private static final byte[] kSignature = {
    (byte) 0xb0, (byte) 0x29, (byte) 0xb7, (byte) 0x74,
    (byte) 0xfc, (byte) 0xdc, (byte) 0xf7, (byte) 0xae,
    (byte) 0xaf, (byte) 0x11, (byte) 0x60, (byte) 0x16,
    (byte) 0xcb, (byte) 0x72, (byte) 0x20, (byte) 0xb0,
    (byte) 0x98, (byte) 0xeb, (byte) 0x68, (byte) 0x5b,
    (byte) 0xa0, (byte) 0x37, (byte) 0xe1, (byte) 0x20,
    (byte) 0xf,  (byte) 0x1a, (byte) 0x4a, (byte) 0xb7,
    (byte) 0x4b, (byte) 0xf9, (byte) 0xa2, (byte) 0x50,
    (byte) 0x6,  (byte) 0x8c, (byte) 0x6d, (byte) 0x6,
    (byte) 0xc2, (byte) 0x7a, (byte) 0xfd, (byte) 0x22,
    (byte) 0xd0, (byte) 0xf,  (byte) 0xaa, (byte) 0xbd,
    (byte) 0x62, (byte) 0x73, (byte) 0x69, (byte) 0x30,
    (byte) 0x8e, (byte) 0xea, (byte) 0xfa, (byte) 0x73,
    (byte) 0x7d, (byte) 0x50, (byte) 0x25, (byte) 0x34,
    (byte) 0xaa, (byte) 0x54, (byte) 0x7c, (byte) 0xac,
    (byte) 0xc3, (byte) 0xcb, (byte) 0xe3, (byte) 0xf0,
    (byte) 0x85, (byte) 0xf7, (byte) 0x38, (byte) 0xd0,
    (byte) 0xa8, (byte) 0xd9, (byte) 0x89, (byte) 0xd4,
    (byte) 0x6b, (byte) 0x3,  (byte) 0x60, (byte) 0x54,
    (byte) 0xf4, (byte) 0xcc, (byte) 0xc7, (byte) 0x2e,
    (byte) 0x28, (byte) 0x25, (byte) 0x59, (byte) 0x1d,
    (byte) 0xec, (byte) 0x67, (byte) 0x7d, (byte) 0xd1,
    (byte) 0x24, (byte) 0x77, (byte) 0xd0, (byte) 0x80,
    (byte) 0x12, (byte) 0x23, (byte) 0xeb, (byte) 0x57,
    (byte) 0xdb, (byte) 0x12, (byte) 0x48, (byte) 0x9a,
    (byte) 0x5d, (byte) 0xeb, (byte) 0xef, (byte) 0x28,
    (byte) 0x34, (byte) 0x5d, (byte) 0x2b, (byte) 0x23,
    (byte) 0x7e, (byte) 0x52, (byte) 0xdd, (byte) 0xe,
    (byte) 0xbf, (byte) 0x5d, (byte) 0xed, (byte) 0xf,
    (byte) 0x36, (byte) 0x73, (byte) 0x77, (byte) 0xe3,
    (byte) 0x15, (byte) 0xf6, (byte) 0xa0, (byte) 0xf2,
    (byte) 0x50, (byte) 0x5c, (byte) 0x7d, (byte) 0x7e,
    (byte) 0x90, (byte) 0x50, (byte) 0xa4, (byte) 0xea,
    (byte) 0x6a, (byte) 0x2,  (byte) 0x5b, (byte) 0x5b,
    (byte) 0xdf, (byte) 0x51, (byte) 0xe9, (byte) 0x1f,
    (byte) 0x60, (byte) 0x46, (byte) 0x16, (byte) 0xb3,
    (byte) 0xb6, (byte) 0x72, (byte) 0xc1, (byte) 0x70,
    (byte) 0x8c, (byte) 0x59, (byte) 0xb2, (byte) 0xae,
    (byte) 0x83, (byte) 0xb5, (byte) 0x19, (byte) 0x30,
    (byte) 0x5b, (byte) 0x4,  (byte) 0x22, (byte) 0x39,
    (byte) 0x2,  (byte) 0x4f, (byte) 0x13, (byte) 0x17,
    (byte) 0xe1, (byte) 0xed, (byte) 0x7b, (byte) 0x41,
    (byte) 0x34, (byte) 0x97, (byte) 0x8c, (byte) 0x54,
    (byte) 0x44, (byte) 0xb4, (byte) 0xa2, (byte) 0xf3,
    (byte) 0xe,  (byte) 0x51, (byte) 0x59, (byte) 0x0,
    (byte) 0x54, (byte) 0xc5, (byte) 0xd2, (byte) 0xae,
    (byte) 0x9f, (byte) 0xe3, (byte) 0x38, (byte) 0xc6,
    (byte) 0x2,  (byte) 0x7a, (byte) 0x9b, (byte) 0xb0,
    (byte) 0xda, (byte) 0xfb, (byte) 0x58, (byte) 0x9c,
    (byte) 0x4e, (byte) 0x15, (byte) 0xb7, (byte) 0x75,
    (byte) 0xe9, (byte) 0xe3, (byte) 0x93, (byte) 0xee,
    (byte) 0x2,  (byte) 0xa,  (byte) 0xef, (byte) 0xe6,
    (byte) 0xc2, (byte) 0xa0, (byte) 0xde, (byte) 0x3d,
    (byte) 0x81, (byte) 0x36, (byte) 0xa0, (byte) 0xe,
    (byte) 0x78, (byte) 0xd5, (byte) 0x81, (byte) 0x3a,
    (byte) 0xa3, (byte) 0xb7, (byte) 0xdb, (byte) 0x4e,
    (byte) 0x72, (byte) 0xa1, (byte) 0x6a, (byte) 0xa8,
    (byte) 0xd9, (byte) 0x58, (byte) 0xc9, (byte) 0x99,
    (byte) 0xd9, (byte) 0x39, (byte) 0x94, (byte) 0xf9,
    (byte) 0x1a, (byte) 0xd1, (byte) 0x84, (byte) 0x26,
    (byte) 0xc7, (byte) 0xfd, (byte) 0x54, (byte) 0x51,
    (byte) 0xba, (byte) 0xd7, (byte) 0xff, (byte) 0xc2,
    (byte) 0x72, (byte) 0xe5, (byte) 0x7d, (byte) 0x2,
    (byte) 0x91, (byte) 0xb0, (byte) 0xe,  (byte) 0x3e
  };

  private static final byte[] kAESKey = {
    (byte) 0x2b, (byte) 0x7e, (byte) 0x15, (byte) 0x16,
    (byte) 0x28, (byte) 0xae, (byte) 0xd2, (byte) 0xa6,
    (byte) 0xab, (byte) 0xf7, (byte) 0x15, (byte) 0x88,
    (byte) 0x09, (byte) 0xcf, (byte) 0x4f, (byte) 0x3c
  };

  private static final byte[] kDESKey = {
    (byte) 0x2b, (byte) 0x7e, (byte) 0x15, (byte) 0x16,
    (byte) 0x28, (byte) 0xae, (byte) 0xd2, (byte) 0xa6
  };

  private static final byte[] kDESEDEKey = {
    (byte) 0x2b, (byte) 0x7e, (byte) 0x15, (byte) 0x16,
    (byte) 0x28, (byte) 0xae, (byte) 0xd2, (byte) 0xa6,
    (byte) 0x2b, (byte) 0x7e, (byte) 0x15, (byte) 0x16,
    (byte) 0x28, (byte) 0xae, (byte) 0xd2, (byte) 0xa6,
    (byte) 0x2b, (byte) 0x7e, (byte) 0x15, (byte) 0x16,
    (byte) 0x28, (byte) 0xae, (byte) 0xd2, (byte) 0xa6,
  };

  private static final byte[] kBlockPlaintext = {
    (byte) 0x32, (byte) 0x43, (byte) 0xf6, (byte) 0xa8,
    (byte) 0x88, (byte) 0x5a, (byte) 0x30, (byte) 0x8d,
    (byte) 0x31, (byte) 0x31, (byte) 0x98, (byte) 0xa2,
    (byte) 0xe0, (byte) 0x37, (byte) 0x07, (byte) 0x34
  };
  private static final byte[] kAESiv = {
    (byte) 0x01, (byte) 0x02, (byte) 0x03, (byte) 0x04,
    (byte) 0x05, (byte) 0x06, (byte) 0x07, (byte) 0x08,
    (byte) 0x09, (byte) 0x0a, (byte) 0x0b, (byte) 0x0c,
    (byte) 0x0d, (byte) 0x0e, (byte) 0x0f, (byte) 0x10
  };

  private static final byte[] kDESiv = {
    (byte) 0x01, (byte) 0x02, (byte) 0x03, (byte) 0x04,
    (byte) 0x05, (byte) 0x06, (byte) 0x07, (byte) 0x08,
  };
}
```

Index