PROVIDER INDEPENDENT CRYPTOGRAPHIC TOOLS

(ALATAN KRIPTOGRAFI BEBAS PENYEDIA)

SUBARIAH BT. IBRAHIM
MAZLEENA SALLEH

Jabatan Sistem & Komunikasi Komputer
Fakulti Sains Komputer & Sistem Maklumat
UNIVERSITI TEKNOLOGI MALAYSIA

2003

PROVIDER INDEPENDENT CRYPTOGRAPHIC TOOLS

(ALATAN KRIPTOGRAFI BEBAS PENYEDIA)

SUBARIAH BT. IBRAHIM
MAZLEENA SALLEH

RESEARCH VOTE NO.
71858

Jabatan Sistem & Komunikasi Komputer
Fakulti Sains Komputer & Sistem Maklumat
Universiti Teknologi Malaysia

2003

ACKNOWLEDGEMENT

Alhamdullilah, the research development project for electronic voting is finally completed. With that, I would like to say thank you to those who has contributed to the success of this project directly or indirectly.

Firstly, I would like to express my gratitude to RMC for its financial support which has made it possible for the completion of this project. Special thanks to RMC staff, Puan Zarina, for her help in various clerical work regarding the project. Thank you too, to FSKSM staffs who has somehow contributed indirectly to the success of this project.

Special thanks to my co-researcher, Assoc. Prof. Mazleena Salleh, for fruitful discussion in the development of the project.

I also would like to thank my research assistant, Shah Rizan Abdul Aziz, who has spent an enormous amount of time in designing and development the cryptographic library. He also spent a considerable amount of time in preparing this report, thank you again.

Last but not least, I wish to thank my husband, Assoc. Prof. Dr Mohd Salihin Ngadiman for his support in my research participation.

Puan Subariah Ibrahim,

Project Leader.

ABSTRACT

PROVIDER INDEPENDENT CRYPTOGRAPHIC TOOLS

(*Keyword : Cryptography, Cryptographic ServiceProvider*)

This aim of this research is to provide a library of cryptographic tools which can be used for teaching and further research. The library consists of cryptographic algorithms that include symmetric and asymmetric encryptions, key exchange, hashing algorithms and digital signature algorithms. The library is implemented by using Java cryptographic service provider framework that conforms to Java Cryptographic Architecture (JCA) and Java Cryptographic Extension (JCE). The library is developed using Software Development Life Cycle (SDLC), supported by Unified Modeling Language (UML) for the design. The programming language used is Java JDK1.4.1.

Key Researchers:

Puan Subariah Ibrahim (Head)

Assoc. Prof. Mazleena Salleh

Shah Rizan Abdul Aziz

E-mail:	subariah@fsksm.utm.my
Tel. No.:	07-553 2386
Vote No.:	71858

ABSTRAK

ALATAN KRIPTOGRAFI BEBAS PENYEDIA

(*Kata Kunci : Kriptografi, Penyedia Perkhidmatan Kriptografi*)

Matlamat utama penyelidikan ini adalah bagi menyediakan pustakaan alatan kriptografi yang boleh digunakan untuk pengajaran dan juga melakukan penyelidikan terhadap algoritma kriptogradi selanjutnya. Pustakaan ini terdiri daripada algoritma-algoritma kriptografi termasuk algoritma penyulitan simetri dan tak simetri, penukaran kekunci, algoritma cincang serta tandatangan digital. Pustakaan ini dilaksanakan menggunakan rangka kerja penyedia perkhidmatan kriptografi Java yang selaras dengan *Java Cryptographic Architecture* (JCA) dan *Java Cryptographic Extension* (JCE). Pustakaan ini dibangunkan dengan menggunakan kaedah Kitaran Hayat Pembangunan Perisian dan disokong dengan rekabentuk *Unified Modeling Language* (UML). Bahasa pengaturcaraan yang digunakan untuk pembangunan perisian ialah Java JDK1.4.1.

Key Researchers:

Puan Subariah Ibrahim (Head)

Assoc. Prof. Mazleena Salleh

E-mail: subariah@fsksm.utm.my

Tel. No.: 07-553 2386

Vote No.: 71858

TABLE OF CONTENT

TABLE LIST

DIAGRAM LIST

APPENDIX LIST

CHAPTER 1

INTRODUCTION

Nowadays, security is always a concern for IT developers. It is not strange to see that any IT conference that discusses this topic will always be packed out. Today the number of people using the Internet continues to increase. This trend has increased the public awareness of the need of secure application and the problems that are caused by failure in this area. For an e-commerce company, breaches in computer security can reduce the public confidence in doing online transaction with the company.

1.1 Problem Statement

Security is a fundamental issue in the development of information and communication technology applications. Cryptography is the most established mechanism that can provide confidentiality, integrity and authentication security services in these applications. Cryptographic tools provide encryption, key-exchange and hashing functions. Encryption algorithms consist of symmetric and asymmetric algorithms. Symmetric algorithms use substitution and transposition of message symbols, while asymmetric algorithms employ mathematics in providing the strength of the algorithms. Key-exchange provides means of exchanging secret or session keys, while hashing provides means of checking integrity of messages.

A cryptographic provider is a set of cryptographic tools provided by an organization or company. There are several cryptographic libraries available, some of which are freely available. One of them is SunJCE provider that is provided by JavaSoft. However, the provider only provides implementation for a small number of algorithms.

In addition, there are no available cryptographic library tools implemented locally in Malaysia. By using other providers, we are unsure if there exists any trapdoor or backdoor in the coding. By developing our own code for the cryptographic tools, we are certain of the security of the tools, and there is no restriction for using strong encryption. This project has developed cryptographic tools, which can work with any other providers. This allows users to incorporate our tools in existing applications, which may use tools from other providers.

1.2 Aim

The expected output of this research is a library of cryptographic tools. This library can be used for teaching undergraduate and graduate cryptography classes in enhancing students understanding of cryptographic algorithms. It also can provide ground for further research in the field.

1.3 Objective

The objectives of this project are:

a) To provide locally developed cryptographic tools that are provider independent.

b) To provide cryptographic tools, which students and researchers can do further research on.

c) To obtain comprehensive understanding of mathematic involved in cryptographic tools.

d) To identify and develop classes those are needed in developing cryptographic tools.

1.4 Scope

The scopes of this project are:

a) The provider implements encryption, key-exchange and hashing tool only.

b) Symmetric algorithms that are implemented are triple-DES, IDEA, Rijndael, RC5 and DES.

c) For asymmetric algorithms, RSA (encryption and digital signature), DSS and ECDSA are developed.

d) Hashing tools like MD5 and SHA algorithms are developed.

CHAPTER 2

LITERATURE STUDY

This chapter discusses Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE). Besides, we also discuss the requirements that are needed to implement a provider. In addition, a brief explanation about several algorithms is also discussed.

2.1 Java Cryptography Architecture

An object-oriented framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact [1]. JCA [2] is an object-oriented framework for accessing and developing cryptographic functionality for the Java platform. It was first brought in the Java Development Kit (JDK) 1.1 to accommodate message digest and digital signature services.

In consequent releases, the Java 2 SDK notably extended the Java Cryptography Architecture. It also improved the certificate management infrastructure to support X.509 v3 certificates, and introduced a new Java Security

Architecture for fine-grain, highly configurable, flexible, and extensible access control.

The Java Cryptography Extension (JCE) expands the JCA API to include APIs for encryption, key exchange, and Message Authentication Code (MAC). Together, the JCE and the cryptography aspects of the SDK provide a complete, platform-independent cryptography API. JCE was previously a not obligatory package (extension) to the Java 2 SDK, Standard Edition, versions 1.2.x and 1.3.x. JCE has now been integrated into the Java 2 SDK, v 1.4.

A good cryptography framework has two important criteria, which are algorithm independent and implementation independent. For a framework to be implementation independent it should conceal the details of a provider from an application and the application should not call directly any packages of the provider. That is to say, a framework should part an application and providers by sitting between them. In this fashion, an application can only see an implementation independent interface, called upward interface, of the framework. Conversely, a framework should provide a downward API that can be implemented in many ways. To be algorithm independent the upward interface of a framework should be abstract and only show a relationship with generic cryptographic concepts, such as Message Digest, rather than concrete algorithms such as SHA-1 and MD5.

The diagram in Figure 2.1 shows that the JCA design follows the above paradigm and its structure. The JCA architecture contains three pieces: the JCA-based applications, the JCA framework and the JCA-compliant providers. The JCA-based applications are on the top. The JCA framework provides an upward interface, called engine classes, and standard names, which are a set of security algorithms names such as RSA and MD5. Since a JCA-based application only knows the engine classes and the standard names, only engine class names and standard names will appear in the application's sources code and its configuration files. The framework also defines a downward uniform API, called "Service Provider Interface" (SPI), to underlying cryptographic providers. This uniform interface allows a provider to be replaced at run-time.

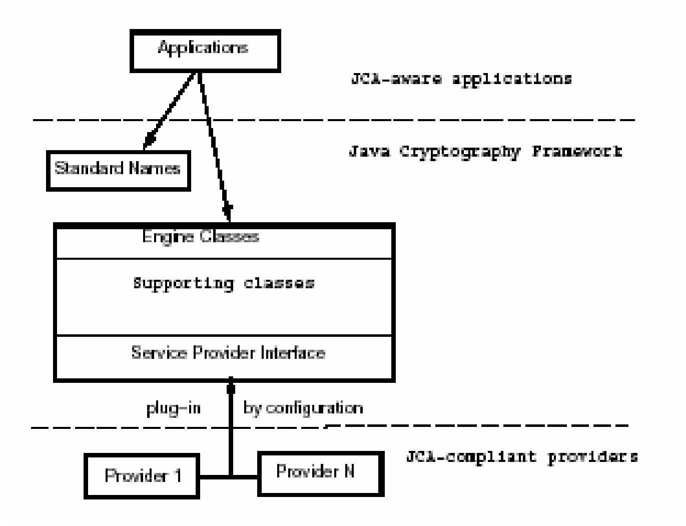


Figure 2.1: The JCA Architecture [3]

2.2 Engine Classes

The JCA includes the classes of the Java 2 SDK Security package that is related to cryptography, this includes the engine classes. Users of the API ask for and use instances of the engine classes to carry out corresponding operations. The following engine classes are defined in the JCE:

a) `MessageDigest`: used to calculate the message digest (hash) of specified data.

b) `Signature`: used to sign data and verify digital signatures.

c) `KeyPairGenerator`: used to generate a pair of public and private keys suitable for a specified algorithm.

d) `KeyFactory`: used to convert opaque cryptographic keys of type `Key` into *key specifications* (transparent representations of the underlying key material), and vice versa.

e) `CertificateFactory`: used to create public key certificates and Certificate Revocation Lists (CRLs).

f) `KeyStore`: used to create and manage a *keystore*. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.

g) `AlgorithmParameters`: used to manage the parameters for a particular algorithm, including parameter encoding and decoding.

h) `AlgorithmParameterGenerator`: used to generate a set of parameters suitable for a specified algorithm.

i) `SecureRandom`: used to generate random or pseudo-random numbers.

j) `CertPathBuilder`: used to build certificate chains (also known as certification paths).

k) `CertPathValidator`: used to validate certificate chains.

l) `CertStore`: used to retrieve `Certificates` and `CRLs` from a repository.

In the JCE, the following engine classes are defined:

a) `Cipher`: used to provide the functionality of a cryptographic cipher used for encryption and decryption. It forms the core of the JCE framework.

b) `KeyGenerator`: used to generate secret keys for symmetric algorithms

c) `KeyAgreement`: used to provide the functionality of a key agreement protocol. The keys involved in establishing a shared secret are created by one of the key generators (`KeyPairGenerator` or `KeyGenerator`), a `KeyFactory`, or as a result from an intermediate phase of the key agreement protocol.

d) `Mac`: used to provide the functionality of a Message Authentication Code (MAC).

e) `SecretKeyFactory`: used to convert keys (opaque cryptographic keys type `java.security.Key`) into key specifications (transparent representations of the underlying key material in a suitable format) and vice versa.

The application interfaces supplied by an engine class are implemented in terms of a Service Provider Interface (SPI). That is, for each engine class, there is a corresponding abstract SPI class, which defines the SPI methods that cryptographic service providers must implement.

An instance of an engine class, the API object, puts in a nutshell (as a private field) an instance of the corresponding SPI class, the SPI object. All API methods of an API object are declared final and their implementations invoke the corresponding SPI methods of the encapsulated SPI object. An instance of an engine class (and of its corresponding SPI class) is created by a call to the `getInstance` factory method of the engine class.

The name of each SPI class is the same as that of the corresponding engine class, followed by `Spi`. For example, the SPI class corresponding to the `Signature` engine class is the `SignatureSpi` class.

Each SPI class is abstract. To supply the implementation of a particular type of service, for a specific algorithm, a provider must subclass the corresponding SPI class and provides implementations for all the abstract methods. An engine class is the `MessageDigest` class, which provides access to a message digest algorithm. Its implementations, in `MessageDigestSpi` subclasses, may be those of various message digest algorithms such as SHA-1, MD5, or MD2. Table 2.1 shows the SPI classes, which are defined in the JCA and the JCE.

Table 2.1: Engine and SPI classes

Engine Class	**SPI Class**
MessageDigest	MessageDigestSpi
Signature	SignatureSpi
KeyPairGenerator	KeyPairGeneratorSpi
KeyFactory	KeyFactorySpi
CertificateFactory	CertificateFactorySpi
KeyStore	KeyStoreSpi
Cipher	CipherSpi
KeyGenerator	KeyGeneratorSpi
KeyAgreement	KeyAgreementSpi
Mac	MacSpi
SecretKeyFactory	SecretKeyFactorySpi

Another requirement is that the provider code needs to be signed by a trusted entity. The trusted entity can Sun itself. Sun added this additional requirement to install a provider because the United States imposed an export control restriction upon JCE. Therefore, a provider must request a code-signing certificate from Sun, and the certificate is then used to sign the provider code.

2.3 How JCA Works

`Java.security` and `javax.crypto` packages and their sub packages consist of important classes for the programmers to use in a development of a secure application. Both frameworks allow many different possible implementations of the algorithms from different providers. For example, although the implementation is written in other language, programmers only have to know the usage of the standard classes in the JCA and JCE.

Programmers do not have to be aware of the existence of any of the providers installed in JDK. They only need to know the usage of the classes that are defined in the JCA and JCE. When user code used the classes (engine classes), JCE will hand over all requests for cryptographic functions to those provider classes. To get the implementation of the provider, the provider must be added to the system. This can be done either statically or dynamically. Figure 2.2 shows the process.

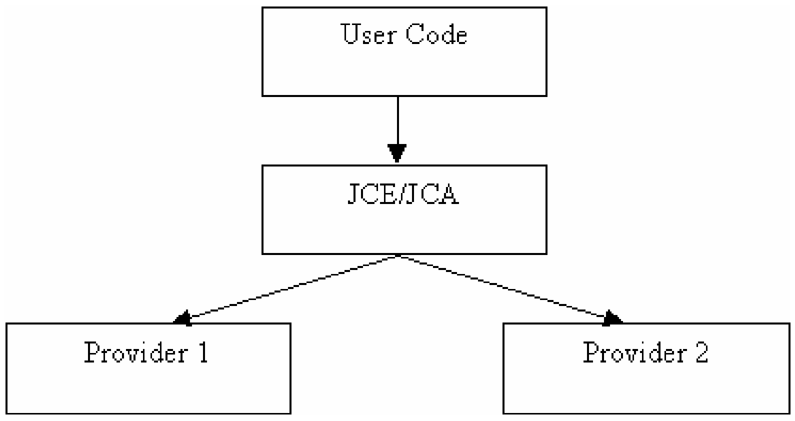


Figure 2.2: How JCA or JCE Works

The classes in the JCA and the JCE provide a framework for cryptographic functions. In order to create the implementations, we must extend the abstract classes called Service Provider Interface (SPI). Each method in SPI classes must be implemented. Figure 2.3 shows the relation of the Signature and SignatureSpi classes to a provider's implementation of a digital signature algorithm.

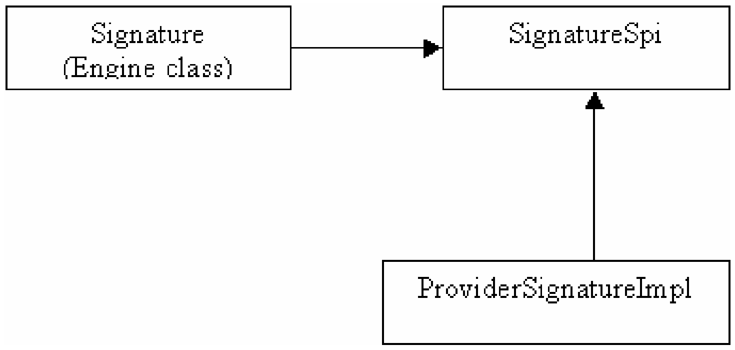


Figure 2.3: Relation of a SPI class, an engine class and the provider

2.4 Algorithms

The followings are the algorithms that are implemented in this project:

a) The RSA [4] cryptosystem is a public-key cryptosystem that offers both encryption and digital signatures (authentication). Ronald Rivest, Adi Shamir, and Leonard Adleman developed the RSA system in 1977.RSA stands for the first letter in each of its inventors' last names

b) The Data Encryption Standard (DES) [5] was developed by The National Bureau of Standards with the help of the National Security Agency in the 1970s. Its purpose is to provide a standard method for protecting sensitive commercial and unclassified data. IBM created the first draft of the algorithm, calling it LUCIFER. DES officially became a federal standard in November of 1976.

c) Triple DES [4] encryption is the encryption standard used by many VPN solutions including several IPSec implementations. It uses a 192 bit key to encrypt\/decrypt data.

d) IDEA (International Data Encryption Algorithm) [4] is the second version of a block cipher designed and presented by Lai and Massey. It is a 64-bit iterative block cipher with a 128-bit key and eight rounds. Decryption is carried out in the same manner as encryption once the decryption subkeys have been calculated from the encryption subkeys. The cipher structure was designed to be easily implemented in both software and hardware, and the security of IDEA relies on the use of three incompatible types of arithmetic operations on 16-bit words. The speed of IDEA in software is similar to that of DES.

e) MD2 [6], MD4 [7], and MD5 [8] are message-digest algorithms developed by Rivest. They are meant for digital signature applications where a large message has to be "compressed" in a secure manner before being signed with the private key. All three algorithms take a message of arbitrary length and produce a 128-bit message digest. While the structures of these algorithms are somewhat similar, the design of MD2 is quite different from that of MD4 and MD5. MD2 was optimized for 8-bit machines, whereas MD4 and MD5 were aimed at 32-bit machines.

f) SHA-1, SHA-256, SHA-384, SHA-512 [9] are versions of Secure Hash Algorithm, which generate respectively, hashes of 160, 256, 384, or 512 bits.

g) Tiger [10] is a fast new **hash** function, designed to be very fast on modern computers, and in particular on the state-of-the-art 64-bit computers (like DEC-Alpha). It is optimized for 64-bit processors and produces 128, 160, or 192-bit hashes

h) RIPEMD-160, RIPEMD-128 [11] are cryptographic hash functions, designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. It is used as a secure replacement for the 128-bit hash functions MD4, MD5, and RIPEMD. RIPEMD was developed in the framework of the EU project RIPE (RACE Integrity Primitives Evaluation, 1988-1992).

i) Diffie-Hellman [12] key agreement protocol (also called exponential key agreement) was developed by Diffie and Hellman in 1976 and published in the groundbreaking paper "New Directions in Cryptography." The protocol allows two users to exchange a secret key over an insecure medium without any prior secrets

j) DSA [4] was published by The National Institute of Standards and in the Digital Signature Standard (DSS), which is a part of the U.S. government's Capstone project

k) The Elliptic Curve Digital Signature Algorithm (ECDSA) [13] is the elliptic curve analogue of the Digital Signature Algorithm (DSA). It was accepted in 1999 as an ANSI standard, and was accepted in 2000 as IEEE and NIST standards. It was also accepted in 1998 as an ISO standard, and is under consideration for inclusion in some other ISO standards. Unlike the ordinary discrete logarithm problem and the integer factorization problem, no sub exponential-time algorithm is known for the elliptic curve discrete logarithm problem. For this reason, the strength-per-key-bit is substantially greater in an algorithm that uses elliptic curves.

l) ECDH [14] a key exchange scheme based on the elliptic curve discrete logarithm problem.

CHAPTER 3

METHODOLOGY

This chapter describes how this cryptographic library was developed based on the scopes that were stipulated in Chapter 1 and discusses the steps that were used. The first section explains the methodology in general. The section explains the implementation steps in detail.

3.1 Methodology

In general, this project followed Software Development Life Cycle model. The project started with the study of the algorithms that was going to be implemented. The algorithms fell into three main categories, which were encryption/signature tools, key exchange tools and hashing tools. The study included how the algorithm work, the key size needed, types of padding and modes of operation.

After that, a design of the whole library was made. The design conformed to the framework defined in Java Cryptographic Architecture. In the design, a number of SPI classes were sub classed. The sub classes implemented all the abstract methods in the corresponding SPI class. The methods contained the implementation of the algorithms.

The implementation process followed the guidelines stated in the official Java documentation. The guidelines ensured that library conformed to the standard imposed by Sun and also guaranteed that the library will work seamlessly in Java. One of the important requirements in the guidelines was the signing requirement from authorized entity. Such signing would ensure the authenticity of the library.

After completing the implementation, the correctness of the library was determined by testing. Due to time constraint, the fastest and accurate way to determine it was by comparing the output of the algorithms with the other library. In the testing, the inputs to the algorithms were determined in advance and must be equal to each other.

3.2 Implementation Details

Once the design of the library was completed, the coding process began. Java classes that correspond to UML classes in the design were created. The implementation of the algorithm was put into the sub classes of SPI class. All the abstract methods in the SPI classes were overridden.

Having completed the coding process, the Master class was created. The function of the class was to register the algorithms and services implemented in the library. The Master class was a sub class of Provider class in `java.security` package. The coding for registering the algorithms and services was included in the constructor function. Besides the algorithms, the class also held the name of the library. For this library, we named it as “FSKSM”. After coding, the code was compiled as normal.

For testing, several steps were taken. Because we also developed algorithms that require classes in `javax.crypto` package (JCE), we signed the library using a signing certificate. The certificate was issued by an authorized entity, which was SUN. Therefore, the next step was to obtain the certificate. Next, after compressing the library files into a JAR file, the certificate was used to sign the library. Finally, the library was ready to use.

3.3 Hardware And Software Specification

Hardware and software play an important role to determine the development efficiency. To develop this provider, several hardware and software criteria were determined.

3.3.1 Hardware

The following are hardware specifications:

a) 166 MHz (recommended minimum)
b) 64 MB memory (recommended)
c) 50 MB hard disk space (depending on features installed)
d) Microsoft® Windows 98 and above

3.3.2 Software

The followings are two pieces of software that were needed in this project:

a) JBuilder is a Java development environment that is used to speed up the programming process. Open, scalable, and standards-based, JBuilder provides a set of visual development tools for creating applications for the Java 2 platform. It also provides customers the freedom of choice to develop and deploy their Java applications to any of the leading operating systems, including Mac OS X, Linux, Windows, and Solaris.

b) Another tool that is used is Rational Rose. Rational Rose® software is the award-winning model-driven development tool that is part of Rational Software's comprehensive and fully integrated solution designed to meet today's software development challenges. Rational Rose assists developer to design better software using the UML method.

CHAPTER 4

DESIGN AND IMPLEMENTATION

This chapter discusses the design and implementation of the provider. UML notation is used to design the classes that are needed for the library. Besides, it also discusses the files created, the installation process and brief instructions on how to use the provider.

4.1 SPI Concrete Classes

This provider has the following SPI concrete classes:

a) `JCEDESCipher`, `JCEIDEACipher`, `JCEDESedeCipher`, `JCERC564Cipher`, `JCERSACipher`, `JCERijndaelCipher` extend `CipherSpi`. The classes are used for encryption and decryption.

b) `JCERC5KeyGenerator`, `JCERijndaelKeyGenerator`, `JCEDESKeyGenerator`, `JCEDESedeKeyGenerator`, `JCEIDEAKeyGenerator` extend `KeyGeneratorSpi`. The classes are used for generating symmetric key.

c) `DSAKeyPairGenerator, RSAKeyPairGenerator and DHKeyPairGenerator` extend `KeyGeneratorSpi`. The classes are used to generate asymmetric keys.

d) SHA1, SHA256, SHA384, SHA512, MD2, MD4, MD5, RIPEMD128, RIPEMD160 extend MessageDigestSpi. The classes are used for hashing.

e) `DHKeyAgreement` extends `KeyAgreementSpi`. This class is used to generate secret key using Diffie-Hellman algorithm.

f) `DHAlgorithmParameters` extends `AlgorithmParameterGeneratorSpi` `AlgorithmParametersSpi`, and `DHAAlgorithmParameterGenerator` extends.

Besides the classes above, there are several additional classes that is used to support the functionality of the provider.

4.2 Provider File

The final product of the file is a JAR file, which contains the provider class. The JAR file has been signed by a certificate-signing certificate, which was received from SUN. The name of the JAR file is `FsksmProvider.jar`. The name, however, can be changed into any other names.

The files in the JAR file must not be altered or removed. This is because JCA framework can detect such modifications and throw exception. As a result, the provider becomes unusable. This is an example of security features of this provider that can prevent application from attacks such as modification and alteration.

4.3 Installation

The installation procedure described in this section and the following are adapted from Java Documentation by JavaSoft. The procedure enables Java Security to find the algorithm implementations in the provider when clients (applications) request them. The installation consists of two steps: installing the provider package classes and configuring the provider.

The JAR file containing the provider classes can be installed as an "installed" or "bundled" extension or by placing it in the application CLASSPATH. In this section, we will only discuss the first and third methods.

To install the provider as installed extension, place `FsksmProvider.jar` file in the standard place for the JAR files of an installed extension:

```
<java-home>/lib/ext        [Solaris]
<java-home>\lib\ext        [Windows]
```

Here `<java-home>` refers to the directory where the runtime software is installed, which is the top-level directory of the Java™ 2 Runtime Environment (JRE) or the `jre` directory in the Java™ 2 SDK (Java 2 SDK) software. For example, if you have the Java 2 SDK, v 1.4 installed on Solaris in a directory named

`/home/user1/J2SDK1.4.0`, or on Microsoft Windows in a directory named `C:\J2SDK1.4.0`, then you need to install the JAR file in the following directory:

`/home/user1/J2SDK1.4.0/jre/lib/ext`	[Solaris]
`C:\J2SDK1.4.0\jre\lib\ext`	[Windows]

Similarly, if you have the JRE, v 1.4 installed on Solaris in a directory named `/home/user1/j2re1.4.0`, or on Microsoft Windows in a directory named `C:\j2re1.4.0`, you need to install the JAR file in the following directory:

`/home/user1/j2re1.4.0/lib/ext`	[Solaris]
`C:\j2re1.4.0\lib\ext`	[Windows]

Another way to install the provider is by placing the JAR file in the CLASSPATH. Take, for example, we want to run an application named program.class on Microsoft Windows and the program uses the provider. We run the program by typing the following command:

```
java –cp FsksmProvider.jar program
```

4.3.1 Configuring the Provider

Having installed the provider, the next step is to add the provider to a list of approved providers. This can be done either statically or dynamically.

This is done statically by editing the security properties file

`<java-home>/lib/security/java.security`	[Solaris]
`<java-home>\lib\security\java.security`	[Windows]

Here `<java-home>` refers to the directory where the JRE was installed. For example, if you have the Java 2 SDK v 1.4 installed on Solaris in a directory named `/home/user1/J2SDK1.4.0`, or on Microsoft Windows in a directory named `C:\J2SDK1.4.0`, then you need to edit the following file:

```
/home/user1/J2SDK1.4.0/jre/lib/security/java.security  [Solaris]
C:\J2SDK1.4.0\jre\lib\security\java.security            [Windows]
```

Similarly, if you have the Java 2 Runtime Environment, v 1.4 installed on Solaris in a directory named `/home/user1/j2re1.4.0`, or on Windows in a directory named `C:\j2re1.4.0`, then you need to edit this file:

```
/home/user1/j2re1.4.0/lib/security/java.security     [Solaris]
C:\j2re1.4.0\lib\security\java.security               [Windows]
```

For this provider, this file should have a statement of the following form:

```
security.provider.n=my.utm.fsksm.provider.FsksmProvider
```

This declares the provider, and specifies its preference order *n*. The preference order is the order in which providers are searched for requested algorithms when no specific provider is requested. The order is 1-based; 1 is the most preferred, followed by 2, and so on.

To register the provider dynamically, a program can call either the `addProvider` or `insertProviderAt` method in the Security class. This type of registration is not persistent and can only be done by code that is granted the following permission:

```
java.security.SecurityPermission "insertProvider.FSKSM}"
```

If the JAR file is in the `myjce_provider.jar` file in the `/localWork` directory, then here is a sample policy file `grant` statement granting that permission:

```
grant codeBase "file:/localWork/FsksmProvider.jar" {
      permission
java.security.SecurityPermission."insertProvider.FSKM";
};
```

User must note that whenever JCE providers are not installed extensions, permissions must be granted for when applets or applications using JCE are run while a security manager is installed. There is typically a security manager installed whenever an applet is running, and a security manager may be installed for an application either via code in the application itself or via a command-line argument. Permissions do not need to be granted to installed extensions, since the default system policy file grants all permissions to installed extensions.

Whenever a client does not install the provider as an installed extension, the provider may need the following permissions granted to it in the client environment:

a) `java.lang.RuntimePermission` to get class protection domains. The provider may need to get its protection domain in the process of doing self-integrity checking.
b) `java.security.SecurityPermission` to set provider properties.

In addition, prior to running such application you need to grant appropriate permissions to the provider. For example, a sample statement granting permissions this provider appears below. Such a statement could appear in a policy file. In this example, the FsksmProvider`.jar` file is assumed to be in the `/localWork` directory.

```
grant codeBase "file:/localWork/FsksmProvider.jar" {
  permission java.lang.RuntimePermission
"getProtectionDomain";
  permission java.security.SecurityPermission
      "putProviderProperty.FSKSM";
};
```

4.4 How To Use The Provider

The usage of the provider can be divided into four categories: encryption, signature, hash and key exchange. Therefore, in this section, we will discuss the usage separately.

Since this provider conforms the JCA framework, user must only know the usage of classes in java.security package and its sub packages as well as classes in javax.crypto package and its sub packages. To get the implementation of the provider, user only need to type “FSKSM”, the name of this provider.

4.4.1 Encryption

This section explains the process of generating a key, creating and initializing a cipher object, encrypting a file, and then decrypting it. Throughout this example, we use the Data Encryption Standard (DES).

4.4.1.1 Generating a Key

To create a DES key, we have to instantiate a `KeyGenerator` for DES. To get this provider key generator implementation, specify the provider “FSKSM” in the statement. Since we do not initialize the `KeyGenerator`, a system-provided source of randomness will be used to create the DES key:

```
  KeyGenerator keygen = KeyGenerator.getInstance("DES",”FSKSM”);
  SecretKey desKey = keygen.generateKey();
```

After the key has been generated, the same KeyGenerator object can be re-used to create further keys.

4.4.1.2 Creating a Cipher

The next step is to create a Cipher instance. To do this, we use one of the `getInstance` factory methods of the Cipher class. We must specify the name of the requested transformation, which includes the following components, separated by slashes (/):

a) The algorithm name

b) The mode (optional)

c) The padding scheme (optional)

In this example, we create a DES (Data Encryption Standard) cipher in Electronic Codebook mode, with PKCS #5-style padding and specify FSKSM as the provider. The standard algorithm name for DES is "DES", the standard name for the Electronic Codebook mode is "ECB", and the standard name for PKCS #5-style padding is "PKCS5Padding":

```
Cipher desCipher;
// Create the cipher
desCipher =
Cipher.getInstance("DES/ECB/PKCS5Padding",”FSKSM”);
```

We use the generated `desKey` from above to initialize the Cipher object for encryption:

```
// Initialize the cipher for encryption
desCipher.init(Cipher.ENCRYPT_MODE, desKey);
```

```
    // Our cleartext
    byte[] cleartext = "This is just an example".getBytes();
    // Encrypt the cleartext
    byte[] ciphertext = desCipher.doFinal(cleartext);
    // Initialize the same cipher for decryption
    desCipher.init(Cipher.DECRYPT_MODE, desKey);
    // Decrypt the ciphertext
    byte[] cleartext1 = desCipher.doFinal(ciphertext);
cleartext and cleartext1 are identical.
```

4.5 Signature

In this section, we will generate a public-private key pair for the algorithm named "DSA" (Digital Signature Algorithm). We will generate keys with a 1024-bit modulus using the implementation from FSKSM provider.

4.5.1 Creating the Key Pair Generator

The first step is to get a key pair generator object for generating keys for the DSA algorithm:

```
KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("DSA",”FSKSM”);
```

The next step is to initialize the key pair generator. In most cases, algorithm-independent initialization is sufficient, but in some cases, algorithm-specific initialization is used.

4.5.1.1 Algorithm-Independent Initialization

All key pair generators share the concepts of a keysize and a source of randomness. A KeyPairGenerator class initialize method has these two types of arguments. Thus, to generate keys with a keysize of 1024 and a new SecureRandom object, you can use the following code:

```
SecureRandom random = SecureRandom.getInstance();
random.setSeed(userSeed);
keyGen.initialize(1024, random);
```

4.5.1.2 Algorithm-Specific Initialization

For situations where a set of algorithm-specific parameters already exists (such as "community parameters" in DSA), there are two initialize methods that have an AlgorithmParameterSpec argument. Suppose your key pair generator is for the "DSA" algorithm, and you have a set of DSA-specific parameters, p, q, and g, that you would like to use to generate your key pair. You could execute the following code to initialize your key pair generator (recall that `DSAParameterSpec is an AlgorithmParameterSpec`):

```
DSAParameterSpec dsaSpec = new DSAParameterSpec(p, q, g);
SecureRandom random = New SecureRandom();
keyGen.initialize(dsaSpec, random);
```

4.5.1.3 Generating the Pair of Keys

The final step is generating the key pair. No matter which type of initialization was used (algorithm-independent or algorithm-specific), the same code is used to generate the key pair:

```
KeyPair pair = keyGen.generateKeyPair();
```

4.5.2 Signature Generation and Verification

The following signature generation and verification examples use the key pair generated in the key pair example above.

```
Signature dsa = Signature.getInstance("SHA1withDSA",”FSKSM”);

/* Initializing the object with a private key */
PrivateKey priv = pair.getPrivate();
dsa.initSign(priv);

/* Update and sign the data */
dsa.update(data);
byte[] sig = dsa.sign();

Verifying the signature is straightforward.

/* Initializing the object with the public key */
PublicKey pub = pair.getPublic();
dsa.initVerify(pub);

/* Update and verify the data */
dsa.update(data);
boolean verifies = dsa.verify(sig);
System.out.println("signature verifies: " + verifies);
```

4.6 Hashing

First create the message digest object, as in the following example:

```
MessageDigest sha = MessageDigest.getInstance("SHA-1",”FSKSM”);
```

This call assigns a properly initialized message digest object to the `sha` variable. The implementation implements the Secure Hash Algorithm (SHA-1), as defined in the National Institute for Standards and Technology's (NIST) FIPS 180-1 document.

Next, suppose we have three byte arrays, `i1`, `i2` and `i3`, which form the total input whose message digest we want to compute. This digest (or "hash") could be calculated via the following calls:

```
sha.update(i1);
sha.update(i2);
sha.update(i3);
byte[] hash = sha.digest();
```

An equivalent alternative series of calls would be:

```
sha.update(i1);
sha.update(i2);
byte[] hash = sha.digest(i3);
```

After the message digest has been calculated, the message digest object is automatically reset and ready to receive new data and calculate its digest. All former state (i.e., the data supplied to `update` calls) is lost.

4.7 Key Exchange

Key agreement is a protocol by which 2 or more parties can establish the same cryptographic keys, without having to exchange any secret information. In this example, we discuss how to write Diffie-Hellman key exchange code.

Two parties who wish to communicate; each must have a pair of Diffie-Hellman key. First, create a DH parameter using an `AlgorithmParameterGenerator` object, and specify FSKSM as the name of the provider. FSKSM implementation of this generator uses the SKIP pre-generated value. Therefore, time taken to generate the parameters is not very long.

```
AlgorithmParameterGenerator paramGen
= AlgorithmParameterGenerator.getInstance("DH",”FSKSM”);
paramGen.init(512);
AlgorithmParameters params = paramGen.generateParameters();
dhSkipParamSpec = (DHParameterSpec)params.getParameterSpec
(DHParameterSpec.class);
```

Next, create a key pair by calling `KeyPairGenerator` generate method.

```
KeyPairGenerator KpairGen =
KeyPairGenerator.getInstance("DH",”FSKSM”);
KpairGen.initialize(dhSkipParamSpec);
KeyPair Kpair = KpairGen.generateKeyPair();
```

After that, instantiate a `KeyAgreement` object. This object will be used to do the key agreement by calling its initialize and doPhase methods.

```
KeyAgreement KeyAgree = KeyAgreement.getInstance("DH",”FSKSM”);
KeyAgree.init(Kpair.getPrivate());
KeyAgree.doPhase(OtherPubKey, true);
```

Finally, call generateSecret () method to create the secret value in the form of byte array. The byte array can be used as a key to any symmetric ciphers.

```
byte[] Secret = aliceKeyAgree.generateSecret();
```

4.8 Provider Specification

4.8.1 Symmetric Algorithms

For all symmetric algorithms, the modes of operation that are supported in this provider are:

a) ECB
b) CBC
c) OFB(n)
d) CFB(n)

Where *(n)* is a multiple of 8 that gives the block size in bits, e.g., OFB8. OFB and CFB mode can be used with plain text that is not an exact multiple of the block size if NoPadding has been specified.

Padding schemes that are supported are:

a) No padding
b) PKCS5/7
c) ISO10126/ISO10126-2
d) X9.23/X923

When placed together this gives a specification for an algorithm. The followings are some examples:

a) DES/CBC/X9.23Padding
b) DES/OFB8/NoPadding

c) IDEA/CBC/ISO10126Padding

Table 4.1 shows information about the algorithms.

Name	**KeySizes (in bits)**	**Block Size**
DES	64	64 bit
DESede	128, 192	64 bit
IDEA	128 **(128)**	64 bit
RC5-64	0 .. 256 **(256)**	128 bit
Rijndael	0 .. 256 **(192)**	128 bit

4.1: Symmetric Algorithms

4.8.2 Asymmetric Algorithm

Padding schemes that are supported are:

a) OAEP - Optimal Asymmetric Encryption Padding
b) PCKS1 - PKCS v1.5 Padding

This providers support multiple block encryption and decryption for RSA algorithm. Two modes of operation that are supported are:

a) ECB
b) CBC

When placed together with RSA this gives a specification for an algorithm. Some examples are;

- RSA/CBC/PKCS1Padding
- RSA/ECB/OAEPPadding

The key size can be any multiple of 8 bits large enough for encryption (2048).

4.8.3 Key Agreement

Diffie-Hellman key agreement is supported using the "DH" and "ECDH";

4.8.4 Digest

Table 4.2 shows digest algorithms that are supported in this provider.

Name	Output (in bits)	Notes
MD2	128	
MD4	128	
MD5	128	
RipeMD128	128	basic RipeMD
RipeMD160	160	enhanced version of RipeMD
SHA1	160	
SHA-256	256	Draft version from FIPS 180-2
SHA-384	384	Draft version from FIPS 180-2
SHA-512	512	Draft version from FIPS 180-2
Tiger	192	

Table 4.2: Message Digest Algorithms

4.8.5 Signature Algorithms

Signature schemes that are supported by this provider are as follows:

a) MD5withRSA

b) SHA1withRSA

c) SHA1withDSA

d) SHA1withECDSA

CHAPTER 5

CONCLUSION & SUGGESTION

The provider that has been developed support several important algorithms. The algorithms cover encryption, decryption, signature, hashing and key exchange.

5.1 Discussion

This provider is a Java-based provider that implements important algorithms. It is an alternative provider for those who do not want to use default provider in JDK. This provider follows the guidelines that are provided by Sun. Therefore, it is guaranteed that this provider can be used without compatibility problems with JCA.

In addition, this provider can work with other providers without any problems except that this provider does not support key or parameter encoding because it takes a lot of time to write encoding codes. I found it quite difficult to understand ASN1 format, which most standards follow. I hope that this project will be extended to support key and parameter encoding.

5.2 Suggestion

a) There are still a lot of algorithms yet to be implemented. This project can be extended to support more algorithms.

b) As mentioned in the previous section, this project should support encoding features.

References

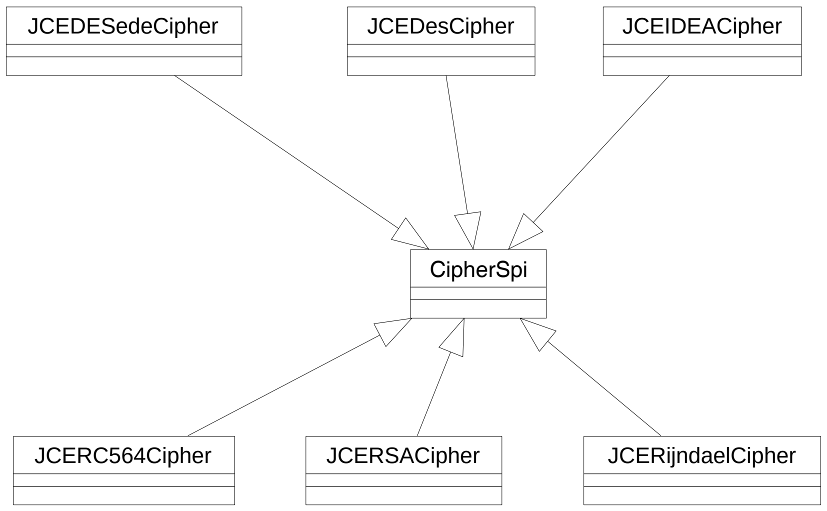
1. R. Johnson. Components, frameworks, patterns (extended abstract). In M. Harandi, editor, *Proceedings of the 1997 Symposium on Software Reusability*, pages 10 – 17, Boston, MA, 1997.
2. Sun Microsystems, Java Cryptography Architecture, API specification & reference. Available at http://java.sun.com/j2se/1.4.1/docs/guide/security/CryptoSpec.html
3. H. Yih, R. David, W. Xunhua. A JCA-based Implementation Framework for Threshold Cryptography. Available at http://www.acsac.org/2001/papers/42.pdf
4. Stallings. W. Cryptography And Network Security: Principles and Practice (Prentice Hall, Upper Saddle River, New Jersey 07458, 1999).
5. National Institute Of Standards and Technology (NIST). FIPS Publication 46-2: Data Encryption Standard (DES) (April 1994).
6. B Kaliski. RFC 1319: The MD2 Message-Digest Algorithm. Internet Activities Board, (April 1992).
7. Rivest, R. "The MD4 Message Digest Algorithm." Proceedings, Crypto '90, August 1990; published by Springer-Verlag.
8. R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. Internet Activities Board, (April 1992).
9. National Institute of Standards and Technology (NIST), FIPS Publication 180-2: Secure Hash Standard (2002 August)
10. Ross Anderson and Eli Biham, Tiger: A Fast New Hash Function, http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger/tiger.html (1996)

11. Hans Dobbertin, Antoon Bosselaers, Bart Preneel, RIPEMD-160: A Strengthened Version of RIPEMD http:// www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9601/AB-9601.pdf
12. RSA Laboratories, PKCS #3 v1.4: Diffie-Hellman Key-Agreement Standard (November 1, 1993).
13. Don Johnson, Alfred Menezes, Scott Vanstone, The Elliptic Curve Digital Signature Algorithm (ECDSA), available at http://www.certicom.com/pdfs/whitepapers/ecdsa.pdf
14. ANSI X9.63, “Public-key cryptography for the financial services industry - Elliptic Curve Key Agreement and Key Transport Algorithms,” draft, 1998.

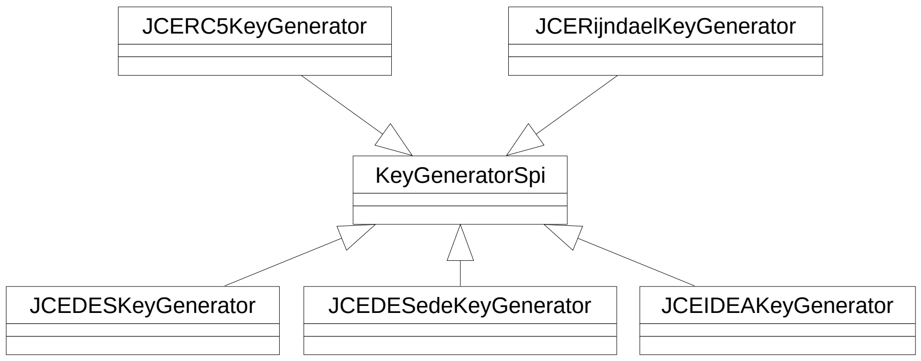
Appendix A - Design

This appendix shows overall design in UML notations.

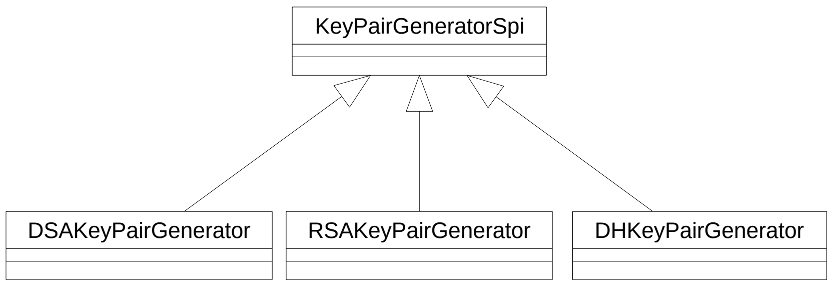
A1 `CipherSpi`



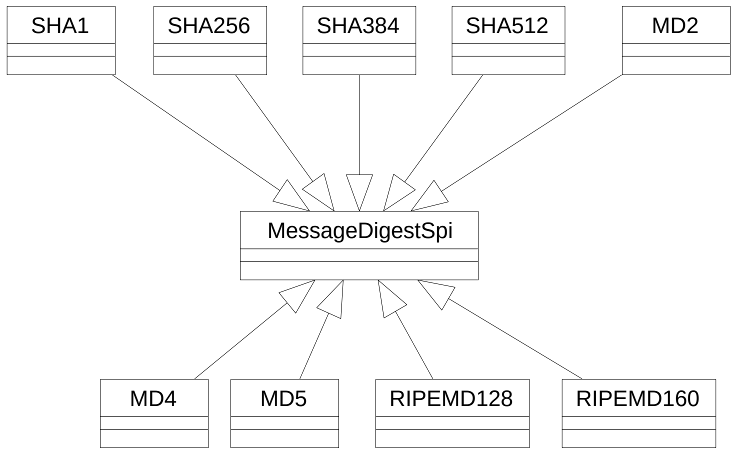
A2 `KeyGeneratorSpi`



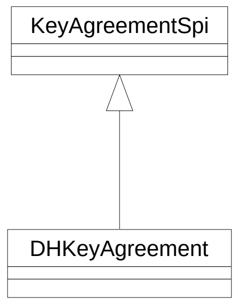
A3 `KeyPairGeneratorSpi`



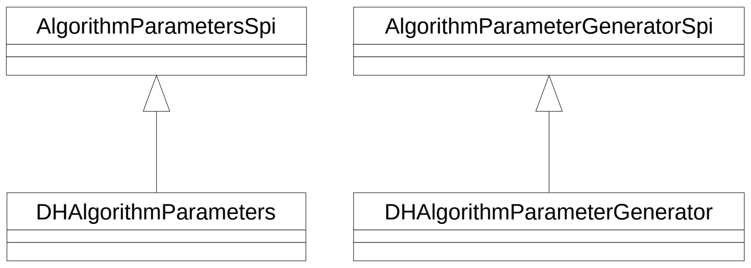
A4 MessageDigestSpi



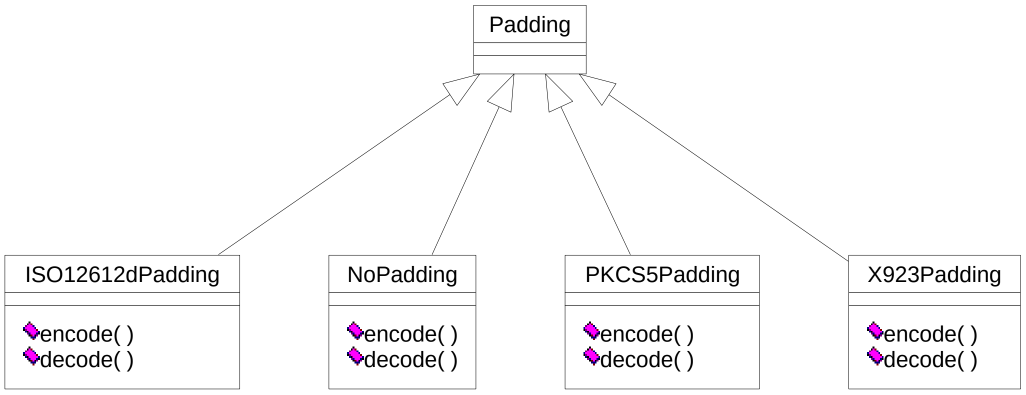
A5 `KeyAgreementSpi`



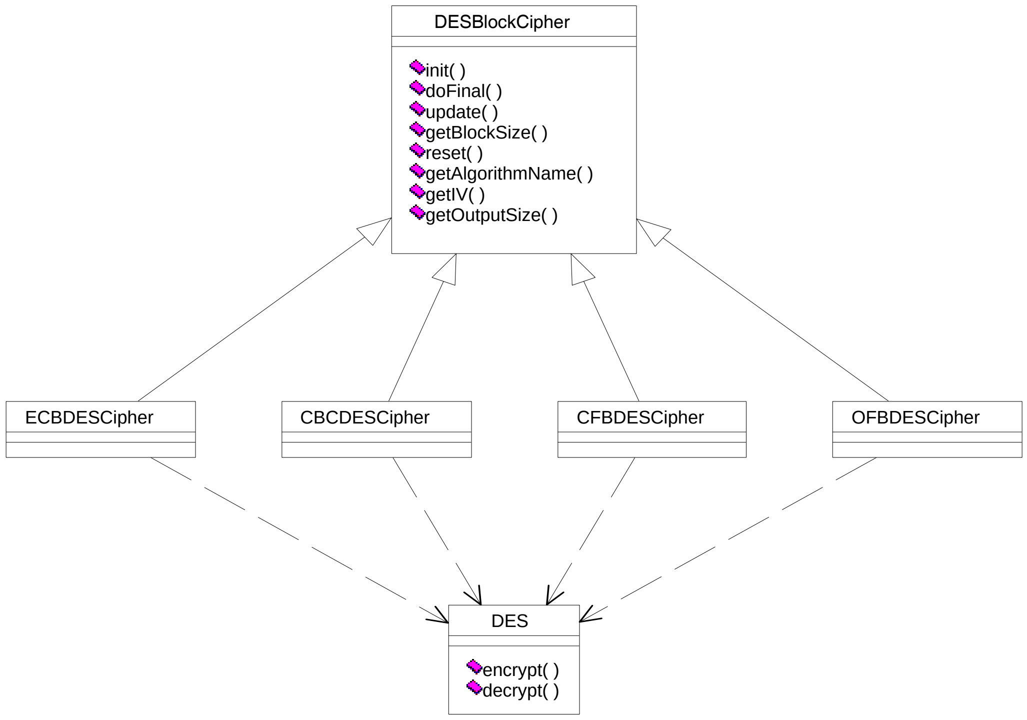
A6 `AlgorithmParametersSpi` And `AlgorithmParameterGeneratorSpi`



A7 Paddings



A8 Modes Of Operation (DES)



APPENDIX B

PAPER: Shah Rizan Abdul Aziz, Subariah Ibrahim, Mazleena Salleh, "*A Java Cryptographic Service Provider*", Proceeding in Research Seminar RMK7 & RMK8 (ASIIT), May 7, 2003.

A JAVA CRYPTOGRAPHIC SERVICE PROVIDER

Shah Rizan Bin Abdul Aziz (1), Subariah Binti Ibrahim (2), Mazleena Binti Salleh (3)

Fakulti Sains Komputer Dan Sistem Maklumat
Universiti Teknologi Malaysia
81310 Johor Bahru, Johor

(1) srk1979@lycos.com (2) subariah@fsksm.utm.my (3) mazleena@fsksm.utm.my

Abstract: *This paper describes the implementation of a Java cryptographic service provider that utilizes the Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE) to provide several security tools. The tools are mainly encryption, key exchange and hashing tools. To ensure the integrity of the security tools, the provider contains self-integrity checking code that checks the Java Archive (JAR) file, which contains the provider's code. The implementation of the provider also follows several programming techniques that ensures the implementation of the provider can only be accessed through JCA and JCE only. The provider uses the latest version of JCA and JCE that are by default built in the core Java library, Java^{TM} Standard Edition 1.4.*

Keywords: *Cryptography, Information Security, Hashing, Encryption, Key-Exchange*

1. Introduction

Security attacks has become the major concern when developing software systems. Most applications are vulnerable to security attacks such as modification, interruption, interception and fabrication [1]. To reduce the risk of being attacked, programmer employs several techniques to develop a secure system. One of the security mechanisms used during the development of a system is cryptography.

Most programmers use additional software called library to add cryptographic functionality in their system. The software or library contains application-programming interface (API) that can be used directly by programmer. Most software includes comprehensive documentation for programmers to refer. Today, there are a lot of cryptography libraries in the Internet. Some of them must be bought before it can be used and some of them are freely distributed and can be modified.

However, there are problems relating to the use of external cryptography library. For libraries that must be bought, most of them are distributed in executable form only. Therefore, programmer does not know the programming codes of the library. If problem arises because of bugs in the library, the only way to correct them is to send them to the original developer and this could cost a lot of money. In contrast, open-source library includes programming codes with the executables and the library can be modified. However, most of open-source libraries are developed on voluntarily basis and the developers are not responsible for any damages caused by the bugs in the library.

Hence, there is a need for an organization or a team of developers to have their own cryptography library. Furthermore, nowadays security modules can be one of the important parts of most software systems, therefore the cryptography library used must be trusted to function properly and reliably. Any bugs in the library should be identified and corrected immediately so that the development of the whole system will not be affected. Moreover, the cryptography library can be used in another software development.

In this project, a Java cryptographic service provider is developed. It is a cryptographic library, called provider, which utilizes the standards in Java: Java Cryptography Architecture and Java Cryptography Extension [2,3]. This provider implements encryption, key-exchange and hashing tools. The encryption tool allows a user to convert a plaintext or data into unintelligible form. The conversion is by means of a reversible translation, based on a translation table or algorithm [1]. In the encryption tools, both symmetric and asymmetric cryptographic algorithms are implemented. The key-exchange tool allows two communicating parties to exchange a session key. The purpose of the tool is to enable two users to exchange a key securely that can then be used to encrypt subsequent messages [1]. Hashing or hash function is a process of mapping a variable-length data block or message into a fixed-length value called a hash code and it is used as an authenticator to data or message [1]. Hashing can provide integrity security service and may be used in implementing digital signature protocol.

This paper discusses the implementation of the library, the algorithms supported and additional security measures taken during the development of the library.

2. Provider Requirements

JCA and JCE provide the classes needed to implement the provider [2-6]. There are several engine classes in javax.crypto (JCE) and java.security (JCA) packages. Engine class is an abstract class (without concrete implementation) that is used to provide cryptographic services [2]. Engine classes in JCA are:

- `MessageDigest`: used to calculate the message digest (hash) of specified data.
- `Signature`: used to sign data and verify digital signatures.
- `KeyPairGenerator`: used to generate a pair of public and private keys suitable for a specified algorithm.
- `KeyFactory`: used to convert opaque cryptographic keys of type `Key` into *key specifications* (transparent representations of the underlying key material), and vice versa.
- `CertificateFactory`: used to create public key certificates and Certificate Revocation Lists (CRLs).
- `KeyStore`: used to create and manage a *keystore*. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities

For JCE, the engine classes are:

- Cipher: provides the functionality of a cryptographic cipher used for encryption and decryption. It forms the core of the JCE framework.
- KeyGenerator: generate secret keys for symmetric algorithms.
- KeyAgreement: provides the functionality of a key agreement protocol.
- Mac: provides the functionality of a Message Authentication Code (MAC).
- SecretKeyFactory: used to convert *keys* (opaque cryptographic keys of type `java.security.Key`) into *key specifications* (transparent representations of the underlying key material in a suitable format), and vice versa.

Because the engine classes do not have concrete implementations, the developer must provide the implementation. The implementation is provided by sub classing Service Provider Interface (SPI) classes. For each engine class, there is a corresponding abstract SPI class, which defines the SPI methods that cryptographic service providers must implement. Table 1 shows the SPI classes.

Table 1: Engine and SPI Classes

Engine Class	SPI Class
MessageDigest	MessageDigestSpi
Signature	SignatureSpi
KeyPairGenerator	KeyPairGeneratorSpi
KeyFactory	KeyFactorySpi
CertificateFactory	CertificateFactorySpi
KeyStore	KeyStoreSpi
Cipher	CipherSpi
KeyGenerator	KeyGeneratorSpi
KeyAgreement	KeyAgreementSpi
Mac	MacSpi
SecretKeyFactory	SecretKeyFactorySpi

Another requirement is the provider code needs to be signed by a trusted entity, which is Sun itself [2]. Sun added this additional requirement for provider that implement cryptographic services found in JCE due to export control restriction imposed by the United States government [6]. A code-signing certificate was requested and it is used to sign the JAR file that contains the provider code.

3.0 How JCA and JCE Work

JCA (java.security package and its sub packages) and JCE (javax.crypto package and its sub packages) contain important classes for the programmers to use cryptographic algorithms in the application. Both frameworks allow many different possible implementations of the algorithms, from different providers. For example, the implementations are written purely in Java or in other languages. JCA and JCE hide the implementation from programmers. Figure 1 illustrates how they work.

The programmers do not have to be aware of the existence of any of the providers. They only need to know the usage of classes in java.security and javax.crypto packages and their sub packages. When the user code uses the classes (engine classes), JCA and JCE will delegate all requests for cryptographic functions to those provider classes. To get the implementation of the provider, the provider must be added to the system. This can be done either statically or dynamically.

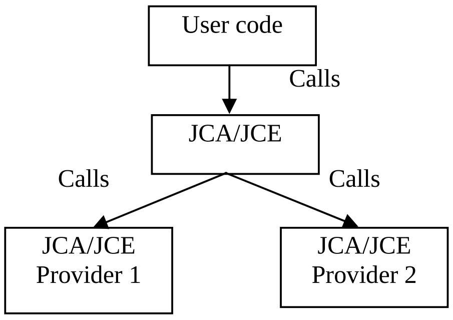


Fig. 1: How JCA or JCE Works

The classes in java.security and javax.crypto packages and their sub packages provide a framework for those cryptographic functions. In order to create the implementations of various algorithms, we must extend the abstract classes called Service Provider Interfaces (SPI). Each method in those SPI classes must be implemented. Figure 2 shows the relation of the Signature and SignatureSPI classes to a provider's implementation of a digital algorithm.

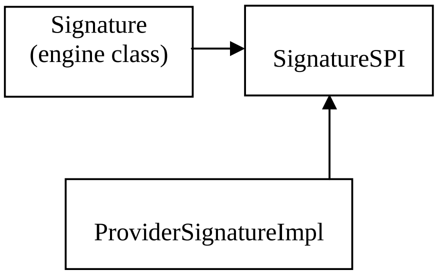


Fig. 2: Relation of a SPI class, an engine class and a provider implementation class

4.0 Implementation

Table 2 lists the algorithms that have been implemented in this provider.

Table 2: Algorithms Supported In the Provider

Cryptographic Tools	Algorithms
Encryption & Decryption	RSA [7], DES [8], Triple DES [1], IDEA [1], RC5 [9]
Signature	RSA [7]
Hashing	MD2 [10], MD4 [11], MD5 [12], SHA-1, SHA-256, SHA-384 , SHA-512 [13], Tiger [14], RIPEMD-160, RIPEMD-128 [15]
Key Exchange	Diffie-Hellman [16]

4.1 Encryption Tools

To provide implementations for encryption algorithms, classes such as CipherSpi, KeyPairGeneratorSpi and KeyGeneratorSpi are extended. KeyPairGeneratorSpi is for the generation of public key and private key in asymmetric algorithm whereas KeyGenerator is generating a key for symmetric algorithm. CipherSpi is for encryption and decryption. Table 3 shows the implementation classes.

Table 4: Implementation Classes For Encryption

Algorithms	Implementation Class
RSA	JCERSACipher JCARSAKeyPairGenerator
DES	JCEDESCipher JCEDESKeyGenerator
Triple DES	JCEDESedeCipher JCEDESedeKeyGenerator
RC564	JCERC564Cipher JCERC564KeyGenerator
IDEA	JCEIDEACipher JCEIDEAKeyGenerator

4.1.1 Modes of Operation

Four modes of operation can be used for all symmetric algorithms. The implementations of the modes are based on the description in [1,17]. The modes are:

- Electronic, Codes Book, ECB
- Chiper Block Chaining, CBC
- Cipher Feedback, CFB
- Output Feedback, OFB

This provider also have additional feature for RSA algorithm. PKCS#1 standard does not define the modes of operation for RSA [8]. Therefore most providers implement the RSA algorithm that can only encrypt a block of data at a time. An exception will be thrown if the message length is greater than a specified value. To overcome this problem, this provider has implemented an RSA cipher that can accept message, with length greater than the block length. Therefore, the cipher can function just as symmetric ciphers do except that the RSA cipher only supports CBC and ECB modes of operation.

4.1.2 Padding

Padding is required when the message length is less than the multiples of the block length. For symmetric algorithms, the provider supports the following padding schemes:

- PKCS#5 [18]
- X923 [19]
- ISO10126-2 [20]

Padding schemes supported by RSA algorithm are PKCS#1 and OAEP [8].

4.2 Signature tool

This tool is used to sign a message and verify the signature. To date, RSA signature algorithm has been implemented and the implementation is based on PKCS#1 documentation [8]. Based on the documentation, RSA signature algorithm requires hashing function such as MD5 and SHA-1. The function is taken from the hashing tools implemented in this provider.

The implementations of the algorithm are contained in two sub classes of SignatureSpi class. One class is the signature with MD5 digest function and the other class is with SHA-1 digest function. The names of the classes are MD5WithRSASignature and SHA1WithRSASignature.

4.3 Key Exchange Tool

This tool is used to enable two entities to exchange keys to produce a session key that then can be used to secure the connection between them. The session key can be used as the key for symmetric algorithms.

To date, Diffie-Hellmann key exchange has been developed. It makes use of SKIP pre-generated values for modulo and bases [6]. It is believed that the use of the SKIP pre-generated values would speed up the key pair generation, as the computer does not have to generate the modulo and bases randomly. With pre-generated values, the programmer does not have to type in the SKIP values manually.

The implementation for KeyPairGenerator, KeyAgreement, AlgorithmParameters and AlgorithmParameterGenerator engine classes for Diffie-Hellmann algorithm are implemented by sub classing the related SPI classes. KeyPairGenerator engine class is used to generate the public and private keys, while KeyAgreement engine class is used to generate the session key after it has received the public keys of the entities that want to communicate. AlgorithmParameter class is used to hold the prime and base values whereas AlgorithmParameterGenerator engine class is used to generate the global parameter values (Actually, this generator doesn't generate. It only return the pre-generated values defined by SKIP).

4.4 Hashing Tool

This hashing tool enables programmer to employ digest function easily. The function produces a unique value that will be used as an authenticator: a value to be used to authenticate a message. This lower-level function is always used as a primitive in a higher-level authentication protocol that enables a receiver to verify the authenticity of a message [1]. RSA digital signature as described in PKCS#1 document is an example of the higher-level function [8].

To implement the algorithm, the MessageDigestSpi class is extended. Each algorithm is implemented in one sub class.

5.0 Security Measures

Several security measures are taken in developing the provider. One of them is adding self-integrity checking codes as recommended by the guideline. This type of checking is included to ensure that the JAR file containing its code has not been manipulated in an attempt to invoke provider methods directly rather than through JCE and JCA [5].

The checking procedure uses the certificate that has been used to sign the JAR file that contains this provider. The certificate is embedded in the checking code as a byte array. The embedded certificate will be used to check whether or not the provider code is authentic [5]. Each constructor in each implementation class must call the code.

Another security measure that has been taken in developing the cryptographic tools are by declaring all SPI implementation classes in the provider package as final so that they cannot be sub classed.

Besides that, their implementation methods are declared protected. This kind of coding can make the provider classes become unusable if someone tries to instantiate them directly that is to bypass JCE or JCA. In order to prevent crypto-related helper classes in the provider package to be accessed from outside the provider package, the classes are declared so that they have package-private scope [5].

6.0 Conclusion

Although there are a number of providers already available, this provider is developed because of several reasons. By developing our own provider, it can be modified and distributed by us. Also, the development of the provider teaches us how to implement a provider. By implementing our own, we also can ensure the correctness and the security of the provider. The providers available from the Internet may contain malicious code that can harm the application using it. Some of the providers in the Internet might not contain some algorithms we need. Therefore, this provider is implemented with the aim to provide as many algorithms as possible. RSA algorithm implemented in this provider supports ECB and CBC modes while other providers only support ECB mode. The implementation of Diffie-Hellman key-exchange uses pre-defined SKIP values for faster key generation.

7.0 References

[1] Stallings, W, *Cryptography And Network Security: Principles and Practice* (Prentice Hall, Upper Saddle River, New Jersey 07458,1999).

[2] Sun Microsystems, Inc, *JavaTM Cryptography Architecture*, http://java.sun.com/j2se/1.4.1/docs/guide/security/CryptoSpec.html

[3] Sun Microsystems, Inc, *JavaTM Cryptography Extension*, http://java.sun.com/j2se/1.4.1/docs/guide/security/jce/JCERefGuide.html

[4] Sun Microsystems, Inc, *How to Implement a Provider for the JavaTM Cryptography Architecture*, http://java.sun.com/j2se/1.4.1/docs/guide/security/HowToImplAProvider.html

[5] Sun Microsystems, Inc, *How to Implement a Provider for the JavaTM Cryptography Extension*, http://java.sun.com/j2se/1.4.1/docs/guide/security/jce/HowToImplAJCEProvider.html

[6] Garms, J. and Somerfield, D., *Professional Java Security: JCA, JCE, JAAS, JSSE, SSL and E-Commerce* (Wrox Press Ltd, Birmingham, UK, 2001).

[7] RSA Laboratories, *PKCS #1: PKCS #1 v2.0: RSA Cryptography Standard* (October 1998).

[8] National Institute of Standards and Technology (NIST). *FIPS Publication 46-2: Data Encryption Standard (DES)* (April 1994).

[9] Rivest., R. "The RC5 Encryption Algorithm." In proceedings, Second International Workshop on Fast Software Encryption, December 1994; published by Springer-Verlag.

[10] B. Kaliski. *RFC 1319: The MD2 Message-Digest Algorithm*. Internet Activities Board, (April 1992).

[11] Rivest, R. "The MD4 Message Digest Algorithm." Proceedings, Crypto '90, August 1990; published by Springer-Verlag.

[12] R. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, (April 1992).

[13] National Institute of Standards and Technology (NIST), *FIPS Publication 180-2: Secure Hash Standard*. (2002 August)

[14] Ross Anderson and Eli Biham, *Tiger: A Fast New Hash Function,* http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger/tiger.html (1996)

[15] Hans Dobbertin, Antoon Bosselaers, Bart Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD* http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9601/AB-9601.pdf

[16] RSA Laboratories, *PKCS #3 v1.4: Diffie-Hellman Key-Agreement Standard* (November 1, 1993).

[17] National Institute of Standards and Technology (NIST). *FIPS Publication 81: DES Modes Of Operation* (April 1994).

[18] RSA Laboratories, *PKCS #5 v2.0: Password-Based Cryptography Standard* (March 25, 1999).

[19] American National Standard Institute, X9.23, *Encryption of Wholesale Financial Messages* (1995).

[20] International Organization for Standardization (ISO 10126-2:1991), *Banking – Procedures for Message Encipherment (wholesale) – Part 2: DEA Algorithm* (ISO 10126-2:1991).