Java Security Architecture and Extension in Practice

Volker Roth
Fraunhofer Institute for Computer Graphics
Rundeturmstrasse 6, 64283 Darmstadt, Germany
Phone +49 6151 155-536, Fax +49 6151 155-499
vroth@igd.fhg.de

November 8, 2001

Abstract

In this article, we report on experiences with the JCA/JCE and existing Java Security Providers (short: providers) that we drew from building provider- and algorithm-independent support for PKCS#7 based on the JCA/JCE. A number of problems came to our attention which were caused by shortcomings in the JCA and in a number of existing Java Security Providers. The repetitive pattern of these shortcomings suggests that they are likely of general interest for those building either providers or software that shall function independent of particular providers or a particular cryptographic algorithm.

1 Introduction

Sun's Java has been a success story since its inception, and is at the verge of being accepted in certain application domains as a "real" alternative to existing programming languages. Java is respected for its platform independence (due to the free availability of Java Virtual Machines for a number of popular platforms), the dynamic loading of classes and for the security features of the language, such as its bytecode verifier, strong typing and bounds checking.

With the Java Development Kit come a number of core packages that comprise the basic support of Java. This includes a reference implementation of the *Java Cryptography Architecture* (JCA), a framework that attempts to provide a common interface for accessing basic cryptographic primitives [14]. Such primitives cover digital signatures, certificates and one–way hash functions.

Provider	URL/comment
SUN	Comes with JDK1.2
SunJCE †	`http://java.sun.com/security/`
CDC	`http://www.informatik.tu-darmstadt.de/TI/`
Bouncy Castle	`http://www.bouncycastle.org/index.html`
Cryptix	`http://www.cryptix.org/`
IAIK	`http://jcewww.iaik.tu-graz.ac.at/`
J/Safe †	`http://www.rsa.com/`
Protekt	`http://www.forge.com.au/`
ABA	Cannibalized by Forge/Protekt
Wedgetail	`http://www.wedgetail.com/`
DSTC/JCSI	See Wedgetail

Table 1: Examples of *security provider* packages; Providers marked with † are subject to export control in the United States of America.

This framework is complemented with another, the *Java Cryptography Extension* (JCE), which is modeled along the same design principles, and which provides primitives such as ciphers, key exchange and keyed hash functions (MACs). The reference implementation of the JCE falls under the *Department of Commerce Bureau of Export Administration* regulations.[1] At present an unhampered version of the JCE is not made available outside the United States of America. The API documentation is available, though.[2] A number of international companies and organisations produced cleanroom implementations of this API, which are available in source code outside the United States. These are typically bundled with a number of cryptographic packages implemented along the lines of JCA and JCE. Table 1 shows a number of such *security provider* packages most of which are available at the time of writing (some are obsolete).

The goal of JCA/JCE is to provide applications independence of particular algorithms and of a particular implementation of cryptographic primitives. Ideally, multiple providers should be usable interchangeably and complementing each other, accessed through a well-defined set of API and wrapper classes that comprise the JCA/JCE. No detailed knowledge about the actual provider being used should be necessary in order to build cryptographically enhanced applications. In the best of all worlds, applications only need to query for an algorithm name and desired key length and are able to locate and use this cipher if it is supported by

[1] `http://www.bxa.doc.gov`

[2] Throughout the article, we refer to version 1.2 of the JCE.

some installed provider.

Unfortunately, we are not living in the best of all worlds and building industry-strength crypto-enabled software based on the available providers is problematic. Problems are caused on the one hand by some shortcomings of the JCA/JCE which are on the other hand exacerbated by sloppy implementation and interpretation of the API and implementation guidelines. In the remainder of this article we are pointing out some hints on how to build providers in an interoperable way and list some mistakes that should be avoided.

In sect. 2 we outline the scope of applications that we feel should be feasible to implement based on a provider-independent layer, in particular the JCA and JCE. We use these ideas in sect. 3 to motivate our discussion of certain mechanisms which are critical to locating algorithm implementations in the security provider packages. Section 3 also provides a condensed introduction to the JCA and JCE architecture and design principles. A more detailed description can be found for instance in the books of Knudsen [6] and Oaks [7]. First-hand information on the JCA is found in [14] and the API documents available from Sun. We wrap up the article in sect. 4.

2 Building on JCA/JCE

Beyond pure cryptography significant effort is required to make applications communicate securely. The secured messages as well as the keys and algorithm parameters need to be encoded in ways that allow proper decoding and the various alternative choices of algorithms and structures must be identified in an unique way. A number of standards were established with this in mind, notably the PKCS family of standards which covers but is not limited to RSA key representation and encryption [13], Diffie-Hellman key agreement [10], password-based encryption [11], the syntax of cryptographically protected messages [9], the representation of encrypted private keys [12], and the syntax of certificate requests [8]. The PKCS standards are defined in terms of ASN.1 [4] and make use of structures and attributes predefined in ITU-T Recommendations X.501 [3] and X.509 [2] which in turn refer to a vast number of further ISO and ITU standards. Encoding and decoding of the various structures defined in these standards is governed by another ITU-T Recommendation, X.690 [5]. Implementing all of these standards leaves programmers with an impressing amount of work.

Nevertheless, a variety of companies and organizations are doing exactly that and some of them even provide source code of their implementations. Some of the standards mentioned above must be dealt with on the level of JCA/JCE providers (for instance PKCS#1, PKCS#3, PKCS#5, PKCS#8, and X.509 certificates). Oth-

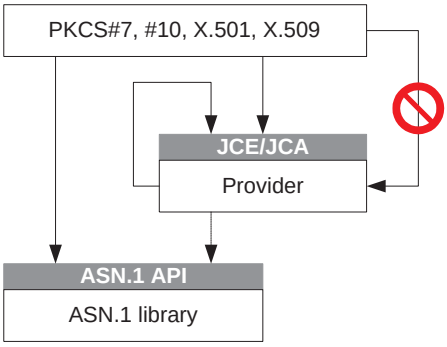

Figure 1: Layers above the JCA/JCE level should be independent of a particular provider. In order to use new algorithms with the PKCS#7 library, adding an appropriate provider should suffice.

ers are located above the layer of JCA/JCE (for instance PKCS#7). Yet others such as the X.509 *AlgorithmIdentifier* are used both by providers and by libraries built on top of the JCA/JCE. From a software designer's perspective, an architecture such as the one illustrated in fig. 1 is desirable. In other words, the PKCS#7 implementation should not depend upon a particular provider but use the abstractions provided by the JCA/JCE. Any implementation that does not live up to these expectations is either:

- hardwired to a particular provider or
- must return generic ASN.1 structures for which the application has to do the mapping to JCE/JCA structures.

The first choice is clearly not desirable since the advantages of the JCA/JCE architecture are negated and the application becomes dependent on a single provider. The second choice is inefficient. In order to use new algorithms with a PKCS#7 implementation, adding a new provider which implements said algorithms should be all there is to do. Ideally, there would be a Java Extension and API for ASN.1 on which both providers and layers atop the JCA/JCE can build. This would spare providers the efforts of re-implementing subsets of ASN.1 and the Distinguished Encoding Rules which are the default encoding rules for virtually all opaque representations of keys and algorithm parameters in the JCA/JCE.

We would like to emphasize that a truly JCA/JCE integrated implementation of PKCS#7 and related objects should use the semantics of the JCA/JCE. An implementation of *AlgorithmIdentifier* should return an instance of

java.security.AlgorithmParameters rather than an ASN.1 object; implementations of PKCS#7 *SignedData* should be able to locate a suitable *Signature* engine on their own, and implementations of PKCS#7 *EncryptedContentInfo* should be able to decrypt with little more than a *RecipientInfo* and a private key, to name some examples.

```
SignerInfo ::= SEQUENCE {
   version Version,
   issuerAndSerialNumber IssuerAndSerialNumber,
   digestAlgorithm DigestAlgorithmIdentifier,
   authenticatedAttributes [0] IMPLICIT Attributes OPTIONAL,
   digestEncryptionAlgorithm DigestEncryptionAlgorithmIdentifier,
   encryptedDigest EncryptedDigest,
   unauthenticatedAttributes [1] IMPLICIT Attributes OPTIONAL
}
```

Figure 2: The ASN.1 structure *SignerInfo* from PKCS#7 [9]

The key to these features is the way algorithms and structures are identified throughout the standards we mentioned. This is done by means of a built-in ASN.1 type which is the *Object Identifier* (OID). An OID is a sequence of integers which uniquely identifies an entity such as a company, an algorithm, or a data structure. The common notation for OIDs is to write the succession of its integers separated by dots. Each successive integer stands for a node in the tree of all registered OIDs. For instance, RSA Laboratories is registered under OID 1.2.840.113549; starting with this prefix, RSA is able to define OIDs of its own choosing. For instance, the PKCS#1 standard is denoted by 1.2.840.113549.1.1 and the RSA algorithm defined in PKCS#1 is denoted by 1.2.840.113549.1.1.1, both chosen by RSA Laboratories.

Knowing the algorithm being used is frequently not enough. Algorithms such as symmetric block ciphers in cipher block chaining mode [1] require an initialization vector (IV). Some ciphers support multiple key lengths. Key lengths and IVs are *parameters* that are required for the correct initialization and operation of such ciphers. For this reason, cryptographic algorithms are identified in X.509 and the PKCS family of standards by means of an *AlgorithmIdentifier* which contains the OID of the algorithm in question as well as the parameters, if any. The ASN.1 structure of the *AlgorithmIdentifier* type is given in fig. 3.

The PKCS#7 *SignerInfo* structure (shown in fig. 2) utilizes two *AlgorithmIdentifiers* – one for the digest algorithm which is used to reduce/hash a message of variable length to a digest value of fixed length, and another for the raw signature/encryption algorithm which is used to transform the digest. Both jointly define the signature algorithm. For instance, the *Digital Signature Standard* (DSS) defines

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY DEFINED BY algorithm OPTIONAL
}
```

Figure 3: The ASN.1 structure *AlgorithmIdentifier* from X.509 [2]

a way of producing digital signatures based on the *Secure Hash Algorithm* (SHA) and the *Digital Signature Algorithm* (DSA). The *AlgorithmIdentifiers* in a *SignerInfo* specifying a DSS signature thus contain the OIDs[3] 1.2.840.10040.4.1 (DSA) and 1.3.14.3.2.26 (SHA1). The combined signature scheme DSS = SHA1/DSA is also identified by a number of alternative OIDs of which one is for instance 1.2.840.10040.4.3.

Implementing the two structures *SignerInfo* and *AlgorithmIdentifier* given in fig. 2 and 3 (bearing in mind fig. 1) leads straight to the sort of problems we outline in sect. 3.

3 JCA/JCE Details and Interpetation

The *security provider* (or *provider* for short) is at the heart of the architecture. It registers the various cryptographic *engine classes* that may be requested by means of the JCA and JCE API. The engine types covered by the JCA/JCE are shown in Table 2. Each particular engine is located through a hash map that maps the engine type and algorithm name to the class that implements the according primitive.

In addition to engine classes, provider packages (should) contain two more types of objects: *transparent* representations of keys and algorithm parameters and *opaque* representations. Transparent representations provide a common view on implementation–specific details of keys and parameters (such as the modulus and exponent of an RSA public key) by means of a well-known interface. *KeySpecs*, with the exception of *EncodedKeySpecs*, are transparent representations of key material, and *AlgorithmParameterSpecs* are transparent representations of algorithm parameters.

Opaque representations hide the details but provide a container for the *encoded* representation of keys and parameters, which is required to exchange such data. For instance, the primary encoding of RSA keys is the one described in PKCS#1 [13] and PKCS#8 [12]. Classes that implement the *Key* interface are

[3] Multiple alternative OIDs exist; for the sake of simplicity we give only one per algorithm.

JCA	JCE
AlgorithmParameterGenerator	Cipher
AlgorithmParameters	KeyAgreement
CertificateFactory	KeyGenerator
KeyFactory	Mac
KeyPairGenerator	SecretKeyFactory
KeyStore	
MessageDigest	
SecureRandom	
Signature	

Table 2: The *engine* classes defined in the JCA and JCE.

opaque representations of key material. Likewise, classes inheriting from *AlgorithmParametersSpi* represent opaque representations of algorithm parameters.

Applications wishing to work with a transparent representation of parameters of algorithm X_i need to import a class $X_iParameterSpec$ at compile time. This is clearly a limitation since once algorithm X_i is broken the code is wasted and replacing X_i with another algorithm requires a modification, re-compilation and re-distribution of the application. An alternative approach is to work on opaque representations. For keys this is not problematic, all engine types that require keys take *Key* objects rather than *KeySpecs*, and all engine types that generate keys return *Key* objects as well. Generators of keys and parameters can be initialized by means of an abstract notion of "strength". It is up to the implementations to generate any algorithm parameters that cannot be derived from this "strength" parameter.

Algorithm parameters are treated more stepmotherly. The engines *Signature*, *Mac*, and *KeyAgreement* do not support initialization with opaque representations. There is a possible way out of this dilemma; the *AlgorithmParameters* class declares a method *getParameterSpec(Class paramSpec)* which returns an *AlgorithmParameterSpec* instance. The only problem is that the understanding of how this method should behave is generally counter to our problem at hand. Most of the implementations that we came across show one of the following behaviors:

- Ignore *paramSpec* and return an instance of a hardcoded class Y.
- If the name of *paramSpec* equals the name of a hardcoded class Y then return an instance of Y. Otherwise throw an exception.
- If a hardcoded class Y is either the same as or a superclass or superinterface of *paramSpec* then return an instance of Y. Otherwise throw an exception.

Leaving the first option aside, getting an *AlgorithmParameterSpec* from an *AlgorithmParameters* instance thus requires knowledge of the class or interface (or subclass or subinterface) of the class supported by that instance. We feel that this is counterintuitive to what the API documentation of the JDK has to say about this method: "paramSpec *identifies the specification class in which the parameters should be returned"*. Thus we suggest the following alternative behavior:

- If this instance supports an *AlgorithmParameterSpec* class Y that can be casted to *paramSpec* then return an instance of Y. Otherwise throw an exception.

We believe that this is both a more faithful and more useful interpretation of the method's documentation. In particular, this interpretation allows to pass *AlgorithmParameterSpec.class* as *paramSpec* and to get a valid default *AlgorithmParamertersSpec* in return which can be used to initialize engines that do not support opaque parameter representations.[4] An example of a correct implementation is given in fig. 4. We now turn our attention to the mechanism used to identify and locate engines, and its significance for the transparent operation of poviders.

```
public class MyAlgorithmParametersSpi extends AlgorithmParametersSpi
 {
     private MyAlgorithmParameterSpec mySpec;

     public AlgorithmParameterSpec engineGetParameterSpec(Class paramSpec)
         throws InvalidParameterSpecException
       {
           if (paramSpec.isAssignableFrom(MyAlgorithmParameterSpec.class))
               return mySpec;

           throw new InvalidParameterSpecException(
               "Unsupported parameter spec!");
       }
 }
```

Figure 4: An example implementation of a working *AlgorithmParametersSpi*. Only method *engineGetParameterSpec* is shown.

Any engine implementation is identified by three parameters: the *engine type*, the *algorithm name*, and the *provider name*. If no provider name is given then

[4]The same argument holds for implementations of *KeyFactory* and *SecretKeyFactory*; both provide methods for converting opaque key representations into transparent ones.

Key	Value
<*engine_name*>.<*stdalg*>	<*class_name*>
Alg.Alias.<*engine_name*>.<*alias*>	<*stdalg*>

Table 3: The syntax of property definitions in Provider instances.

Alias	Comment
<*oid*>	maps OIDs to <*stdalg*>
OID.<*oid*>	maps <*stdalg*> to preferred <*oid*>
<*digestalg*>/<*cipher*>	alternative name of signature engines

Table 4: The syntax of special alias forms.

the installed providers are searched for a matching engine implementation in the order of preference (the order in which they are registered). For this to work, providers declare the supported engine types and algorithms by means of *properties* – key/value pairs of type *String* which are stored in a hash table. The key is the engine type and algorithm name, and the value is the fully qualified name of the class that implements that algorithm. The syntax of these properties is given in Table 3.

The substitution pattern <*stdalg*> denotes the *standard algorithm name* as specified in the Java Cryptography Architecture and Extension API Specification & Reference [14, 15] and the corresponding document for the JCE. If no standard name has been specified for the algorithm in question then a new name must be chosen which should be modeled along the lines of the standard names. The second line in Table 3 is an *alias scheme* which can be used to define alternative names for engine implementations. The alias must be mapped to a standard name. Aliases are resolved at most once and resolving is done only in the scope of the provider that defines the alias.

Aliases are used for instance to map an OID to the standard algorithm name of the engine that implements the algorithm referred to by the OID. Likewise, a special form of aliases maps a standard algorithm name of an engine to a preferred OID for that algorithm. Although this type of mapping is not efficient since the image of the mapping is defined in the key and hence all entries in the provider hash map must be searched for a matching value and key pattern. Additionally, the JCE/JCA specification defines the alternative "slashed" name form <*digestalg*>/<*cipher*> for signature algorithms. Both <*digestalg*> and <*cipher*> stand for the standard name of the digest and raw signature/encryption algorithm name used in the com-

bined signature scheme.

Now, consider the ASN.1 structure *SignerInfo* that we introduced in sect. 2. This structure defines a signature algorithm in terms of two OIDs, one for the digest algorithm and another for the raw signature/encryption algorithm that together comprise the signature algorithm to use for verifying a signature. The signature engines designed according to the JCA already combine the digesting and encryption/verification step. Consequently, signature engines are requested by means of the standard name or an OID of the *combined* sequence. Implementations of *SignerInfo* that fall back to proprietary OID mappings might not support the complete suite of algorithms supported by the installed providers.

A solution that takes into account OIDs declared by the individual installed providers is more desireable at this point. The approach that we took exploits the slashed form of signature engine names. The OIDs extracted from the *SignerInfo* are first resolved to the standard names of the corresponding *MessageDigest* and *KeyPairGenerator* engines using the alias mechanism. This requires that OIDs are set up as aliases for said engine types. Next, the standard names are combined to the slashed form which is resolved to the standard name of the *Signature* engine. This name is then used to request the *Signature* engine implementation. The reverse process is similar. The standard signature algorithm name is resolved to its slashed form which is split into the digest algorithm and cipher name. Both are mapped to the respective OIDs using the alias form for preferred OIDs. The resolving is done over the mappings of all installed providers and resolved combinations are kept in a cache. All required mappings are "learned" from the installed providers in this way.

This scheme works only if: (1) providers support appropriate declaration of OIDs and slashed forms and (2) the naming discipline is kept by the providers.

A special name form is supported for cipher tranformations; the syntax for transformations is <*cipher*>**[**/<*mode*>/<*padding*>**]**; square brackets denote optional elements. <*cipher*> is the name of the cipher algorithm, <*mode*> the name of the operation mode [1] to use, and <*padding*> the padding scheme to use for padding the cipher text to the block size of a block cipher. This name form is parsed by the *Cipher* class in its *getInstance(String transformation [,provider])* method. The question that remains to be answered is which name to use when requesting a matching *AlgorithmParameterGenerator* instance for a given cipher transformation. The padding scheme is not significant for choosing between different algorithm parameters. Our proposition is to use <*cipher*>/<*mode*> from the returned name first and try <*cipher*> in case the first attempt failed.

4 Discussion and Further References

We outlined a piece of software (PKCS standards support) that should be feasible to implement in a provider- and algorithm-independent way based on the JCA/JCE. Some of the required features cannot be implemented in a straightforward way due to some limitations of the JCA and JCE algorithm naming scheme. Yet there is a feasible solution based on the definition of appropriate aliases which depends on the strict adherence of providers to the specifications of the JCA and JCE, in particular to the naming and aliasing schemes. Sun Microsystems Inc. put prominent notice into the specification documents that the aliasing scheme may be changed or even eliminated in the future. However, this mechanism is significant for the transparent mapping of OIDs and algorithm names and is fundamental to the transparent resolution of the signature engine identification problem described in sect. 3. Improvements of the aliasing scheme should aim at providing a more efficient reverse mapping of algorithm names to the preferred OID.

Furthermore, Java Security Provider packages must put more emphasis on parameter handling and algorithm initialization issues. In the past, we experimented with several Java Security Provider packages and particularly observed the following deficiencies:[5]

- OID mappings for symmetric ciphers are not defined.
- OID mappings for *AlgorithmParameters* engines are not defined.
- Implementations of *AlgorithmParameterGeneratorSpi* were not provided for symmetric ciphers.
- Providers rarely implement *AlgorithmParametersSpi* classes for three symmetric ciphers.
- Method *Cipher.getParameters* is not properly supported in many Cipher implementations.
- Default conversion of opaque parameters to parameter specs is not supported properly.

Implementations of *AlgorithmParameterGeneratorSpi* are particularly important because they are a *sine quae non* for transparent initialization of ciphers in a way that is independent of a particular provider and algorithm. In this vein, listing A shows how cipher algorithms can be initialized transparently.

[5] We consulted for the developers of the CDC provider; this provider avoids said deficiencies.

Listing B shows a tool that provides forward and reverse mapping of engine names to OIDs, including the resolution of signature engine names to raw cipher and digest OIDS, and vice versa.

A sample provider class with extensive and accurate declaration of OIDs, aliases, and parameter engines is given in listing C. The full code including sample implementations of algorithm parameter generators and parameter representations is found at URL `http://www.semoa.org`.

The source code of a JCE cleanroom implementation with accurate alias resolving and class loading can also be downloaded from this site, as well as an implementation of ASN.1, DER, PKCS#7, X.501 names and further elements of PKCS and ITU standards. The PKCS#7 implementation uses the tool given in listing B, and does not depend on any particular provider.

References

[1] International Organization for Standardization, Geneva, Switzerland. *Information Processing – Modes of Operation for a 64–Bit Block Cipher Algorithm*, 1987. ISO/IEC 8372.

[2] International Organization for Standardization, Geneva, Switzerland. *Information technology – Open Systems Interconnection – The Directory: Authentication Framework*, nov 1993. ISO/IEC 9594-8, equivalent to ITU-T Rec. X.509, 1993.

[3] International Organization for Standardization, Geneva, Switzerland. *Information technology – Open Systems Interconnection – The Directory: Models*, nov 1993.

[4] International Telecommunication Union. *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, dec 1997. ITU-T Recommendation X.680, equivalent to ISO/IEC International Standard 8824-1.

[5] International Telecommunication Union. *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, dec 1997. ITU-T Recommendation X.690, equivalent to ISO/IEC International Standard 8825-1.

[6] Jonathan B. Knudsen. *Java Cryptography*. The Java Series. O'Reilly, Sebastopol, CA, USA, May 1998.

[7] Scott Oaks. *Java Security*. The Java Series. O'Reilly, Sebastopol, CA, USA, May 1998.

[8] RSA Laboratories. Certificate request syntax standard. Public Key–Cryptography Standards 10, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.

[9] RSA Laboratories. Cryptographic message syntax standard. Public Key–Cryptography Standards 7, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.

[10] RSA Laboratories. Diffie–Hellman key–agreement standard. Public Key–Cryptography Standards 3, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.

[11] RSA Laboratories. Password–based encryption standard. Public Key–Cryptography Standards 5, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.

[12] RSA Laboratories. Private–key information syntax standard. Public Key–Cryptography Standards 8, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.

[13] RSA Laboratories. RSA Encryption Standard. Public Key–Cryptography Standards 1, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.

[14] Sun Microsystems, Inc. *Java™ Cryptography Architecture API Specification & Reference*, July 1999. Internet document available at URL: `http://java.sun.com/j2se/sdk/1.3/docs/guide/security/CryptoSpec.html`.

[15] Sun Microsystems, Inc. *Java™ Cryptography Extension API Specification & Reference*, 1999. Not available outside the U.S.A.

A Opaque Initialization

```
public void opaqueInit(String alg, int size) throws GeneralSecurityException
{
    AlgorithmParameterGenerator pgen;
    AlgorithmParameters params;
    KeyGenerator kgen;
    SecretKey key;
    String alt;
    int n;

    n   = alg.indexOf('/');
    alt = (n > 0) ? alg.substring(0,n) : alg;

    try{
        pgen = AlgorithmParameterGenerator.getInstance(alt);
        pgen.init(size);
        params = pgen.generateParameters();
    }
    catch(NoSuchAlgorithmException e)
    {
        System.out.println("Warning, no parameter generator for "+alg);
        params = null;
    }
    kgen = KeyGenerator.getInstance(alt);
    kgen.init(size);
    key = kgen.generateKey();

    doSomething(key, params, alg);

    return;
}
```

B Engine Name Resolution

```
package codec.util;

import java.security.*;
import java.io.*;
import java.util.*;

/**
 * A number of invariants must hold for the properties defined
 * by the installed providers such that this class can work
 * properly:
 * <ul>
 * <li> Aliases must be mapped to standard JCA/JCE names
 *    whenever possible. All aliases for an engine must
 *    map to the same name.
 * <li> Two OID mappings with the same OID must map to
 *    the same name.
 * <li> The slashed form of signature names must be set up
 *    as an alias.
 * <li> Signature engines that do not have a corresponding
 *    cipher engine still require a reverse OID mapping of
 *    the form Alg.Alias.Cipher.OID.<i>oid</i> = <i>name</i>,
 *    where <i>name</i> is the cipher name component of a
 *    slashed form alias for that signature engine.
 * </ul>
 *
 * @author Volker Roth
 */
public class JCA extends Object
{
    /**
     * The digest/cipher name to signature algorithm name
     * mapping.
     */
    private static Map dc2s_ = new HashMap();

    /**
     * The signature algorithm name to digest/cipher name
     * mapping.
     */
    private static Map s2dc_ = new HashMap();

    /**
     * The root alias map. Each map entry consists of the
     * lower case engine name mapped to another map that
     * holds the aliases for that engine.
```

```
 */
protected static Map aliases_ = initAliasLookup();

/**
 * Let no-one create an instance.
 */
private JCA()
{}


/**
 * Reads the properties of the installed providers and
 * builds an optimized alias lookup table. All entries
 * of the form
 * <ol>
 * <li>
 *    &quot;Alg.Alias.&quot;+&lt;engine&gt;+&quot;.&quot;+&lt;alias&gt;
 *    = &lt;value&gt;
 * <li>
 *    &quot;Alg.Alias.&quot;+&lt;engine&gt;+&quot;.OID.&quot;+&lt;oid&gt;
 *    = &lt;value&gt;
 * <li> &quot;Alg.Alias.&quot;+&lt;engine&gt;+&quot;.&quot;+&lt;oid&gt;
 *    = &lt;value&gt;
 * </ol>
 * are transformed and stored in a hashmap which is used
 * by this class in order to do quick lookups of aliases
 * and OID mappings. The stored entries are of the form:
 * <ol>
 * <li> &lt;engine&gt;+&quot;.&quot;+&lt;alias&gt;
 *    = &lt;value&gt;
 * <li> &quot;oid.&quot;+&lt;value&gt;
 *    = &lt;oid&gt;
 * <li> &quot;oid.&quot;+&lt;oid&gt;
 *    = &lt;value&gt;
 * </ol>
 * In case multiple providers define mappings for the same
 * keys the mapping of the first registered provider wins.
 */
static private Map initAliasLookup()
{
    Enumeration e;
    Provider[] provider;
    String k; // key
    String v; // value
    String s; // string
    String p; // previous mapping
    Map map;
    int i;
    int j;
```

```
map       = new HashMap();
provider = Security.getProviders();

/* We start from the last provider and work our
 * way to the first one such that aliases of
 * preferred providers overwrite entries of
 * less favoured providers.
 */
for (i=provider.length-1; i>=0; i--)
{
    e = provider[i].propertyNames();

    while (e.hasMoreElements())
    {
        k = (String)e.nextElement();
        v = provider[i].getProperty(k);

        if (!k.startsWith("Alg.Alias."))
        {
            continue;
        }
        /* Truncate k to <engine>.<alias>
         */
        k = k.substring(10).toLowerCase();
        j = k.indexOf('.');

        if (j<1)
        {
            continue;
        }
        /* Copy <engine> to s
         * Truncate k to <alias>
         */
        s = k.substring(0,j);
        k = k.substring(j+1);

        if (k.length() < 1)
        {
            continue;
        }
        /* If <alias> starts with a digit then we
         * assume it is an OID. OIDs are uniquely
         * defined, hence we omit <engine> in the
         * oid mappings. But we also include the
         * alias mapping for this oid.
         */
        if (Character.isDigit(k.charAt(0)))
        {
```

```
                    p = (String)map.get("oid."+k);

                    if (p != null && p.length() >= v.length())
                    {
                        continue;
                    }
                    map.put("oid."+k,v);
                    map.put(s+"."+k,v);
                }
                /* If <alias> starts with the string "OID."
                 * then we found a reverse mapping. In that
                 * case we swap <alias> and the value of the
                 * mapping, and make an entry of the form
                 * "oid."+<value> = <oid>
                 */
                else if (k.startsWith("oid."))
                {
                    k = k.substring(4);
                    v = v.toLowerCase();

                    map.put("oid."+v, k);
                }
                /* In all other cases we make an entry of the
                 * form <engine>+"."+<alias> = <value> as is
                 * defined in the providers.
                 */
                else
                {
                    map.put(s+"."+k, v);
                }
            }
        }
        return map;
    }


    /**
     * Returns the JCA standard name for a given OID. The OID
     * must be a string of numbers separated by dots, and can
     * be preceded by the prefix &amp;quot;OID.&amp;quot;. If the OID
     * is not defined in a mapping of some registered provider
     * then <code>null</code> is returned.<p>
     *
     * OID mappings are unambigous; no engine type is required
     * for the mapping and no engine type is returned as part
     * of the result. The returned string consists only of the
     * name of the algorithm.
     *
     * @param oid The string with the OID that shall be resolved.
```

```
     * @return The standard JCA engine name for the given OID or
     *    <code>null</code> if no such OID is defined.
     */
    public static String getName(String oid)
    {
        if (oid == null)
        {
            throw new NullPointerException("OID is null!");
        }
        if (oid.startsWith("OID.") || oid.startsWith("oid."))
        {
            oid = oid.substring(4);
        }
        return (String)aliases_.get("oid."+oid);
    }


    /**
     * Resolves the given alias to the standard JCA name for the
     * given engine type. If no appropriate mapping is defined
     * then <code>null</code> is returned. If the given alias is
     * actually an OID string and there is an appropriate alias
     * mapping defined for that OID by some provider then the
     * corresponding JCA name is returned.
     *
     * @param engine The JCA engine type name.
     * @param alias The alias to resolve for the given engine type.
     * @return The standard JCA name or <code>null</code> if no
     *    appropriate mapping could be found.
     * @exception IllegalArgumentException if the alias is
     *    an empty string.
     */
    public static String resolveAlias(String engine, String alias)
    {
        if (alias == null || engine == null)
        {
            throw new NullPointerException("Engine or alias is null!");
        }
        if (alias.length() < 1)
        {
            throw new IllegalArgumentException("Zero-length alias!");
        }
        return (String)aliases_.get(
            engine.toLowerCase()+"."+alias.toLowerCase());
    }


    /**
     * Returns the OID of the given algorithm name. The given name
```

```
     * must be the JCA standard name of the algorithm and not an
     * alias. Use {@link #resolveAlias resolveAlias} to map aliases
     * onto their standard names.
     *
     * @param algorithm The JCA standard name of the algorithm
     *    for which the OID should be returned.
     * @return The OID or <code>null</code> if no appropriate
     *    mapping could be found.
     * @exception NullPointerException if engine or algorithm is
     *    <code>null</code>.
     */
    public static String getOID(String algorithm)
    {
        if (algorithm == null)
        {
            throw new NullPointerException("Algorithm is null!");
        }
        if (algorithm.length() < 1)
        {
            throw new IllegalArgumentException("Algorithm name is empty!");
        }
        if (Character.isDigit(algorithm.charAt(0)))
        {
            return algorithm;
        }
        return (String)aliases_.get("oid."+algorithm.toLowerCase());
    }


    /**
     * Returns the OID of the given algorithm name. The given
     * engine name is taken as a hint if the given algorithm
     * name is a non-standard name. In that case one shot is
     * given to alias resolving before a second attempt is
     * made to map the algorithm to an OID. Alias resolving
     * is done by means of the {@link #resolveAlias resolveAlias}
     * method.
     *
     * @param algorithm The JCA standard name of the algorithm
     *    for which the OID should be returned.
     * @param engine The engine name that is taken as a hint
     *    for alias resolving if the algorithm name cannot be
     *    resolved in the first attempt.
     * @return The OID or <code>null</code> if no appropriate
     *    mapping could be found.
     */
    public static String getOID(String algorithm, String engine)
    {
        String oid;
```

```
        oid = getOID(algorithm);

        if (oid != null)
        {
            return oid;
        }
        algorithm = resolveAlias(engine, algorithm);

        if (algorithm == null)
        {
            return null;
        }
        return getOID(algorithm);
    }


    /**
     * This method maps a given digest algorithm OID and
     * cipher algorithm OID onto the standard name of the
     * combined signature algorithm. For this to work the
     * aliases must be well defined such as described below:
     * <dl>
     * <dt> Digest Algorithm
     * <dd> Alg.Alias.MessageDigest.<i>oid</i><sub>1</sub>
     *    = <i>digestAlg</i>
     * <dt> Cipher Algorithm
     * <dd> Alg.Alias.Cipher.<i>oid</i><sub>2</sub>
     *    = <i>cipherAlg</i>
     * <dt> Signature Algorithm
     * <dd> Alg.Alias.Signature.<i>digestAlg</i>/<i>cipherAlg</i>
     *    = <i>signatureAlg</i>
     * </dl>
     * The <i>oid</i> denotes the sequence of OID numbers
     * separated by dots but without a leading &amp;quot;OID.&amp;quot;.
     * In some cases, such as the DSA, there is no cipher engine
     * corresponding to <i>oid</i><sub>2</sub>. In this case,
     * <i>oid</i><sub>2</sub> must be mapped to the corresponding
     * name by other engine types, such as a KeyFactory.<p>
     *
     * All found mappings are cached for future use, as well
     * as the reverse mapping, which is much more complicated
     * to synthesise.
     *
     * @param doid The string representation of the digest
     *    algorithm OID. The OID must have a &amp;quot;OID.&amp;quot;
     *    prefix.
     * @param doid The string representation of the cipher
     *    algorithm OID. The OID must have a &amp;quot;OID.&amp;quot;
```

```
     *     prefix.
     * @return The standard JCE name of the signature algorithm
     *     or <code>null</code> if no mapping could be found.
     */
    public static String getSignatureName(String doid, String coid)
    {
        String dn;
        String cn;
        String sn;
        String dc;

        dn = getName(doid);
        cn = getName(coid);

        if (dn == null || cn == null)
        {
            return null;
        }
        dc = dn+"/"+cn;

        synchronized(dc2s_)
        {
            sn = (String)dc2s_.get(dc);

            if (sn != null)
            {
                return sn;
            }
        }
        sn = resolveAlias("signature",dc);

        if (sn != null)
        {
            synchronized(dc2s_)
            {
                cn = dc.toLowerCase();

                if (!dc2s_.containsKey(cn))
                {
                    dc2s_.put(cn,sn);
                }
            }
            synchronized(s2dc_)
            {
                cn = sn.toLowerCase();

                if (!s2dc_.containsKey(cn))
                {
                    s2dc_.put(cn,dc);
```

```
                }
            }
        }
        return sn;
    }


    /**
     * This method maps the standard signature algorithm name
     * to the <i>digestAlg</i>/<i>cipherAlg</i> format. This
     * format can be used to retrieve the OID of the digest
     * algorithm and cipher algorithm respectively.
     * For this to work the aliases must be well defined such
     * as described below:
     * <dl>
     * <dt> Signature Algorithm
     * <dd> Alg.Alias.Signature.<i>d</i>/<i>c</i>
     *    = <i>sigAlg</i>
     * where <i>d</i> denotes the digest algorithm and <i>c</i>
     * the cipher algorithm. <i>sigAlg</i> must be the name
     * under which the algorithm engine is published.
     * </dl>
     *
     * If <code>sigAlg</code> contains a &quot;/&quot; then
     * we assume that the given algorithm name is already
     * of the desired form and return <code>sigAlg</code>.
     *
     * @param sigAlg The standard signature algorithm name.
     * @return The <i>digestAlg</i>/<i>cipherAlg</i> format
     *    of the given signature algorithm name or <code>
     *    null</code> if no suitable mapping could be found.
     */
    public static String getSlashedForm(String sigAlg)
    {
        String v;

        if (sigAlg.indexOf("/") > 0)
        {
            return sigAlg;
        }
        sigAlg = sigAlg.toLowerCase();

        synchronized(s2dc_)
        {
            v = (String)s2dc_.get(sigAlg);

            if (v != null)
            {
                return v;
```

```
        }
    }
    Iterator i;
    String k;
    Map map;
    int m;

    for (i=aliases_.keySet().iterator(); i.hasNext();)
    {
        k = (String)i.next();

        if (!k.startsWith("signature."))
        {
            continue;
        }
        v = (String)aliases_.get(k);

        if (!v.equalsIgnoreCase(sigAlg))
        {
            continue;
        }
        k = k.substring(10);
        m = k.indexOf("/");

        if (m < 0)
        {
            continue;
        }
        synchronized(s2dc_)
        {
            if (!s2dc_.containsKey(sigAlg))
            {
                s2dc_.put(sigAlg,k);
            }
        }
        return k;
    }
    return null;
}


/**
 * This method maps the given standard signature algorithm
 * name to the string representation of the OID associated
 * with the digest algorithm of the given signature algorithm.
 *
 * @param sigAlg The standard signature algorithm name.
 * @return The string representation of the OID associated
 *    with the digest alorithm used for <code>sigAlg</code>.
```

```
 */
public static String getDigestOID(String sigAlg)
{
    int n;
    String v;
    String h;
    String r;

    v = getSlashedForm(sigAlg);

    if (v == null)
    {
        return null;
    }
    n = v.indexOf("/");

    if (n < 0)
    {
        return null;
    }
    h = v.substring(0,n);
    r = getOID(h);

    if (r != null)
    {
        return r;
    }
    /* We now try to "repair" the bad algorithm
     * name if we find a fitting alias instead.
     */
    h = resolveAlias("MessageDigest",h);

    if (h == null)
    {
        return null;
    }
    r = getOID(h);

    if (r != null)
    {
        v = h+"/"+v.substring(n+1);

        synchronized(s2dc_)
        {
            s2dc_.put(sigAlg,v);
        }
    }
    return r;
}
```

```
/**
 * This method maps the given standard signature algorithm
 * name to the string representation of the OID associated
 * with the cipher algorithm of the given signature algorithm.
 * <p>
 * This conversion is a bit tricky. In cases such as DSA,
 * no corresponding Cipher engine exists, since DSA is not
 * designed to be used as a cipher. In such cases, some
 * provider needs to set up a bogus alias of the form:
 * <dl>
 * <dt> Signature Algorithm
 * <dd> Alg.Alias.Cipher.OID.<i>oid</i> = DSA
 * </dl>
 *
 * The <i>oid</i> denotes the sequence of OID numbers
 * separated by dots but without a leading &quot;OID.&quot;.
 *
 * @param sigAlg The standard signature algorithm name.
 * @return The string representation of the OID associated
 *    with the cipher alorithm used for <code>sigAlg</code>.
 */
public static String getCipherOID(String sigAlg)
{
    int n;
    String s;
    String v;
    String r;

    v = getSlashedForm(sigAlg);

    if (v == null)
    {
        return null;
    }
    n = v.indexOf("/");

    if (n < 0)
    {
        return null;
    }
    s = v.substring(n+1);
    r = getOID(s);

    if (r != null)
    {
        return r;
    }
```

```
        /* We now try to "repair" the bad algorithm
         * name if we find a fitting alias instead.
         */
        s = resolveAlias("Signature",s);

        if (s == null)
        {
            return null;
        }
        r = getOID(s);

        if (r != null)
        {
            v = v.substring(0,n)+"/"+s;

            synchronized(s2dc_)
            {
                s2dc_.put(sigAlg,v);
            }
        }
        return r;
    }

}
```

C Sample Provider Class

```
package DE.FhG.IGD.crypto;
import java.security.*;

/**
 * A Java Security Provider according to the JCE specification.
 * Provides basic cryptographic services, in particular those
 * which seem broken in other providers such as:
 * <ul>
 * <li> AlgorithmParameters
 * <li> AlgorithmParameterGenerators
 * </ul>
 *
 * This provider furthermore defines a number of aliases that
 * are ususally not found in other providers, but which are
 * neccessary for such providers to work with tools such as
 * the keytool and jarsigner.
 *
 * @author Volker Roth
 * @version "$Id: a8jce.java,v 1.1 2001/10/11 15:12:34 vroth Exp $"
 * @see A8KeyStore
 */
public class A8Provider extends Provider
{
    /**
     * The name of this provider.
     */
    public static final String name = "A8";


    /**
     * Creates the A8Provider. This provider constructor registers the
     * keystores, ciphers, key factories etc. supported by it.
     */
    public A8Provider()
        {
            super(name, 1.2, "A8 Basic Java Crypto Services");
            AccessController.doPrivileged(new Defines());
        }

    private class Defines extends Object implements PrivilegedAction
     {
         public Object run()
            {
                /* Message Digest Algorithm OID
                 * SHA, MD5, MD4, MD2
```

```
 */
put("Alg.Alias.MessageDigest.OID.1.3.14.3.2.26",
    "SHA");
put("Alg.Alias.MessageDigest.1.3.14.3.2.26",
    "SHA");
put("Alg.Alias.MessageDigest.OID.1.2.840.113549.2.5",
    "MD5");
put("Alg.Alias.MessageDigest.1.2.840.113549.2.5",
    "MD5");
put("Alg.Alias.MessageDigest.OID.1.2.840.113549.2.4",
    "MD4");
put("Alg.Alias.MessageDigest.1.2.840.113549.2.4",
    "MD4");
put("Alg.Alias.MessageDigest.OID.1.2.840.113549.2.2",
    "MD2");
put("Alg.Alias.MessageDigest.1.2.840.113549.2.2",
    "MD2");

/* Signature engines
 */
put("Alg.Alias.Signature.MD5/RSA",
    "MD5withRSA");
put("Alg.Alias.Signature.1.2.840.113549.1.1.4",
    "MD5withRSA");
put("Alg.Alias.Signature.1.3.14.3.2.25",
    "MD5withRSA");
put("Alg.Alias.Signature.OID.1.2.840.113549.1.1.4",
    "MD5withRSA");

/* Cipher engines
 */
put("Alg.Alias.Cipher.OID.1.2.840.113549.3.7",
    "DESede/CBC/PKCS5Padding");
put("Alg.Alias.Cipher.1.2.840.113549.3.7",
    "DESede/CBC/PKCS5Padding");

put("Alg.Alias.Cipher.1.2.840.113549.3.2",
    "RC2/CBC/PKCS5Padding");
put("Alg.Alias.Cipher.OID.1.2.840.113549.3.2",
    "RC2/CBC/PKCS5Padding");

put("Alg.Alias.Cipher.OID.1.2.840.113549.1.5.1",
    "PBEWithMD2AndDES");
put("Alg.Alias.Cipher.1.2.840.113549.1.5.1",
    "PBEWithMD2AndDES");

put("Alg.Alias.Cipher.OID.1.2.840.113549.1.5.3",
    "PBEWithMD5AndDES");
put("Alg.Alias.Cipher.1.2.840.113549.1.5.3",
```

```
    "PBEWithMD5AndDES");

put("Alg.Alias.Cipher.OID.1.2.840.113549.1.5.12",
    "PBEWithSHA1And128BitRC4");
put("Alg.Alias.Cipher.1.2.840.113549.1.5.12",
    "PBEWithSHA1And128BitRC4");

put("Alg.Alias.Cipher.OID.1.2.840.113549.1.1.1",
    "RSA");
put("Alg.Alias.Cipher.1.2.840.113549.1.1.1",
    "RSA");

/* (Secret)KeyFactories
 */
put("Alg.Alias.SecretKeyFactory.1.2.840.113549.3.7",
    "DESede/CBC/PKCS5Padding");
put("Alg.Alias.KeyFactory.1.2.840.113549.1.1.1",
    "RSA");

/* AlgorithmParameters
 */
put("Alg.Alias.AlgorithmParameters.1.2.840.113549.3.7",
    "DESede");
put("Alg.Alias.AlgorithmParameters.DESede/CBC/PKCS5Padding",
    "DESede");

put("Alg.Alias.AlgorithmParameters.1.2.840.113549.3.2",
    "RC2");
put("AlgorithmParameters.RC2",
    "DE.FhG.IGD.crypto.RC2Parameters");
put("AlgorithmParameterGenerator.RC2",
    "DE.FhG.IGD.crypto.RC2ParameterGenerator");

put("Alg.Alias.AlgorithmParameters.1.2.840.113549.1.5.1",
    "PBE");// PBEWithMD2AndDES
put("Alg.Alias.AlgorithmParameters.1.2.840.113549.1.5.3",
    "PBE");// PBEWithMD5AndDES
put("Alg.Alias.AlgorithmParameters.1.2.840.113549.1.5.12",
    "PBE");// PBEWithSHA1And128BitRC4

put("Alg.Alias.AlgorithmParameters.PBEWithMD2AndDES",
    "PBE");// PBEWithMD2AndDES
put("Alg.Alias.AlgorithmParameters.PBEWithMD5AndDES",
    "PBE");// PBEWithMD5AndDES
put("Alg.Alias.AlgorithmParameters.PBEWithSHA1And128BitRC4",
    "PBE");// PBEWithSHA1And128BitRC4

put("AlgorithmParameters.DESede",
    "DE.FhG.IGD.crypto.DESParameters");
```

```
            put("AlgorithmParameterGenerator.DESede",
                "DE.FhG.IGD.crypto.DESParameterGenerator");

            put("AlgorithmParameters.PBE",
                "DE.FhG.IGD.crypto.PBEParameters");
            put("AlgorithmParameterGenerator.PBE",
                "DE.FhG.IGD.crypto.PBEParameterGenerator");

            put("AlgorithmParameterGenerator.PBEWithMD2AndDES",
                "DE.FhG.IGD.crypto.PBEParameterGenerator");

            put("AlgorithmParameterGenerator.PBEWithMD5AndDES",
                "DE.FhG.IGD.crypto.PBEParameterGenerator");

            put("AlgorithmParameterGenerator.PBEWithSHA1And128BitRC4",
                "DE.FhG.IGD.crypto.PBEParameterGenerator");

            put("Alg.Alias.AlgorithmParameters.1.3.14.3.2.12",
                "DSA");
            put("Alg.Alias.AlgorithmParameters.1.2.840.10040.4.1",
                "DSA");

            return null;
        }
    }


}
```