

# **Bachelorarbeit im Fachgebiet Kryptographie und Computeralgebra**

**Sommersemester 2007**



Fachgebiet Kryptographie und Computeralgebra

Fachbereich Informatik

TU Darmstadt

## **Flexible – Eine erweiterbare GUI für den FlexiProvider (Backend)**

**Andreas Roth**

Betreuer: Erik Dahmen  
Martin Döring

7. August 2007

# Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 7. August 2007

# Zusammenfassung

Aufgrund der schnellen Ausbreitung des Internets und dessen Möglichkeiten, wird es immer wichtiger, persönliche Daten zu schützen. Deshalb gewinnen kryptographische Verfahren heutzutage immer mehr an Wichtigkeit. Digitale Signaturen und Verschlüsselung sollen die kryptographischen Schutzziele (*Vertraulichkeit, Integrität, Authentizität, Verbindlichkeit*) gewährleisten. Die *Java Cryptography Architecture* bietet zur Implementierung von vorhandenen, wie auch von neuen kryptographischen Verfahren, Schnittstellen, die so allgemein gehalten sind, dass herkömmliche Konzepte einfach umgesetzt werden können. Ziel dieser Schnittstellen ist es, dem Entwickler möglichst viel Freiraum zu lassen, um beliebige Algorithmen umsetzen zu können und dem Benutzer die Anwendung der Algorithmen so einfach wie möglich zu machen. Der *FlexiProvider* stellt eine Implementierung der JCA Schnittstellen dar, die vor allem auf die Effizienz der umgesetzten Algorithmen ausgelegt ist. Im Sommersemester 2006 ist in einem Bachelorpraktikum erstmals eine grafische Benutzerschnittstelle für den FlexiProvider entstanden, die vor allem das Problem der Schlüsselaufbewahrung implementiert und die Anwendung aller vorhandenen Algorithmen unterstützen soll. Ziel dieser Arbeit ist es, den Client *Flexible* um eine flexiblere Initialisierung und erweiterte Einstellungsmöglichkeiten von Algorithmen in der Oberfläche zu erweitern und fehlende Funktionalitäten zu ergänzen.

## Abstract

Today's fast growth of the internet and its possibilities calls for effective protection of personal data. In this respect, cryptographic applications are more and more gaining in importance. Digital signatures and encryption techniques are designed to maintain the cryptographic protection goals (confidentiality, integrity, authenticity and non-repudiability). The Java Cryptography Architecture (JCA) provides interfaces that are universally applicable to any conventional concept. The goal is, therefore, to provide the developer with as many alternatives for algorithm implementation and parametrization as possible and, furthermore, to simplify their application. The FlexiProvider is an implementation of the JCA interfaces. It is optimized for the efficiency of the implemented algorithms. During a practical course in Sommersemester 2006, the graphical userinterface client Flexible was first developed. Its overall aim was to solve the problem of key storage in the application. The purpose of this thesis is to achieve a more flexible initialization of algorithms and to improve and extend the functionality of the userinterface Flexible.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Java Cryptography Architecture</b>	<b>3</b>
2.1	Wichtige Releases . . . . .	3
2.2	Wichtige Klassen . . . . .	4
<b>3</b>	<b>Fraunhofer FhG ASN.1 CoDec</b>	<b>6</b>
3.1	Public Key Cryptography Standards (PKCS) . . . . .	6
3.2	X.500 Standards . . . . .	6
<b>4</b>	<b>FlexiProvider</b>	<b>7</b>
<b>5</b>	<b>Flexible</b>	<b>9</b>
5.1	Systemvoraussetzungen für Flexible . . . . .	9
5.2	Installation von Flexible . . . . .	9
5.3	Ersetzen des FlexiProviders durch eine andere Version . . . . .	10
5.4	Implementierung einer Algorithmenstruktur . . . . .	11
5.4.1	Einlesen der Informationen aus dem FlexiProvider . . . . .	12
5.4.2	Speichern der ausgelesenen Informationen als XML . . . . .	16
5.4.3	Verbliebene Probleme . . . . .	16
<b>6</b>	<b>Schlüsselgenerierung und Schlüsselverwaltung</b>	<b>18</b>
6.1	Generierung von Schlüsseln . . . . .	18
6.1.1	Symmetrische Schlüssel . . . . .	18
6.1.2	Asymmetrische Schlüssel . . . . .	19
6.2	Java Keystore Schlüsselverwaltung . . . . .	19
6.2.1	Schutz von Integrität und Vertraulichkeit . . . . .	20
6.2.2	Nachteile des Keystore Ansatzes . . . . .	22
<b>7</b>	<b>Kryptographische Anwendungen für Schlüssel</b>	<b>23</b>
7.1	Verschlüsselung . . . . .	23
7.1.1	Symmetrische Verfahren . . . . .	25
7.1.2	Asymmetrische Verfahren . . . . .	26
7.1.3	Hybride Verfahren . . . . .	27
7.2	Signaturverfahren . . . . .	28

<b>8 Implementierung einer Public-Key-Infrastruktur</b>	<b>30</b>
8.1 Certificate Authority (CA) . . . . .	30
8.2 X.509 Zertifikate . . . . .	31
8.2.1 DER kodierte X.509 Zertifikate . . . . .	32
8.2.2 Base64 kodierte X.509 Zertifikate . . . . .	32
8.2.3 Das PKCS#12 Transportformat für Zertifikate und Privatekey . . .	32
8.3 Certificate Signing Request (CSR) . . . . .	33
8.4 Certificate Revocation List (CRL) . . . . .	35
<b>9 Ausblick</b>	<b>36</b>
<b>Literatur</b>	<b>39</b>

## Abbildungsverzeichnis

1 Die JCA/JCE Architektur . . . . .	8
2 Algorithmen Struktur in Flexible . . . . .	11
3 Paket de.samd.initialize . . . . .	14
4 Mögliche Einträge in einem Java Keystore . . . . .	21
5 Symmetrische und asymmetrische Verschlüsselung . . . . .	23
6 Bearbeiten einer CSR Anfrage . . . . .	34

## Listings

1 Anmelden des FlexiCoreProvider in der Security Klasse . . . . .	7
2 Starterskript für Flexible anpassen . . . . .	11
3 Auslesen der Algorithmen aus dem FlexiProvider . . . . .	13
4 Javadoc Konvention für Implementierungen von AlgorithmParameterSpec .	15

# 1 Einleitung

Die Aufgabenstellung dieser Arbeit ist es, eine Benutzerschnittstelle für den *FlexiProvider* [2] zu implementieren, die vor allem die Leistungsfähigkeit und Flexibilität der einzelnen Provider demonstrieren soll. Diese Benutzerschnittstelle soll so flexibel wie möglich sein, um dem versierten Benutzer möglichst viele Einstellungsmöglichkeiten – wie das Parametrisieren von Algorithmen bei der Schlüsselerzeugung – zu ermöglichen. Jedoch muss die Oberfläche zugleich intuitiv genug sein, um den unversierten Benutzer nicht zu überlasten.

Eine weitere wichtige Anforderung an die Applikation ist die Erweiterbarkeit der grafischen Oberfläche. Wenn neue Algorithmen hinzukommen, soll der FlexiProvider ohne viel Aufwand ausgetauscht werden können, so dass die Anwendung immernoch fehlerfrei funktioniert. Desweiteren ist auf die Ansprechbarkeit der Anwendung zu achten. Diese soll mit den Möglichkeiten, die das *Standard Widget Tool* (SWT) in Java bietet, optisch aufgewertet werden.

Ein zentrales Problem, welches Flexible löst, ist die Aufbewahrung von Schlüsseln. Alle Schlüssel, die im Programm aufbewahrt werden, sollen mit den kryptographischen Verfahren, die der FlexiProvider implementiert, funktionsfähig sein. Sie sollen bei Bedarf zu Backup-Zwecken exportiert werden können und bei Bedarf mit anderen Benutzern ausgetauscht werden können. Dazu soll der Java Keystore dienen, der es ermöglicht Schlüssel zu verwalten und persistent und kryptographisch sicher gegen unbefugten Zugriff abzuspeichern.

Es soll weiterhin laut Aufgabenstellung eine kleine *Public Key Infrastruktur* (PKI) entstehen, die es ermöglicht, Zertifikate auf Anfrage zu erstellen, alle ausgestellten Zertifikate zu verwalten und Sperrinformationen über Zertifikate zu veröffentlichen. Der Fokus der Anwendung soll jedoch hauptsächlich auf der Anwendung von Schlüsseln liegen. Die Funktionen einer PKI werden nur exemplarisch umgesetzt und sind zunächst experimentell. Für einfache Szenarien mag diese Funktionalität ausreichen, jedoch sollte bei sicherheitskritischen Aufgaben auf alternative PKI Lösungen wie etwa *FlexiTrust* von FlexSecure [3] zurückgegriffen werden.

Die erste Version von Flexible ist bereits im Sommersemester 2006 entstanden und löst schon einige der oben beschriebenen Probleme, wie etwa die Schlüsselverwaltung mittels Keystore. Jedoch sind wegen der Kürze der Zeit einige wichtige Funktionen offengeblieben, wie eine dynamische Initialisierung von vorhandenen Providern und deren Algorithmen. Dies soll in dieser Arbeit geändert werden. Die bestehende Anwendung wird zunächst einem globalen Refactoring unterzogen, um Programmierfehler ausfindig zu machen und wird anschließend um die noch fehlende Funktionalität erweitert. Die Oberfläche soll noch intuitiver und ansprechender werden (siehe dazu den Frontend Teil der Dokumentation in [8]).

Thema dieser Ausarbeitung ist es weniger die kryptographischen Konzepte, die hinter dem FlexiProvider stecken – wie die umgesetzten Algorithmen – zu vermitteln, denn das würde den Rahmen der Ausarbeitung sprengen, sondern den Aufbau und die Designentscheidungen der Anwendung zu erläutern. An einigen Stellen sind Literaturhinweise angegeben, die diese Konzepte ausführlicher behandeln. Es ist jedoch im Allgemeinen nicht notwendig, diese Konzepte im Einzelnen verstanden zu haben, um Flexible benutzen zu können, oder diese Ausarbeitung zu verstehen.

Der Aufbau der Arbeit sieht wie folgt aus. Nachdem in Kapitel 2 das Konzept der JCA und der Erweiterung durch die JCE besprochen wird, gehe ich in Kapitel 3 auf den Fraunhofer CoDec ein, den sowohl der FlexiProvider, als auch Flexible selbst nutzen, um ASN.1 Strukturen zu kodieren. Kapitel 4 stellt dann die einzelnen Provider und das Konzept hinter dem FlexiProvider vor und beschreibt, wie diese in einer Anwendung eingebunden werden können. Nachdem diese Grundlagen gelegt sind, kann in Kapitel 5 auf Flexible selbst eingegangen werden. Hier geht es vor allem um die dynamische Initialisierung von Algorithmen und die allgemeinen Hinweise zur Installation der Anwendung. Das anschließende Kapitel 6 betrachtet im Großen und Ganzen, wie der Umgang mit Schlüsseln in Flexible aussieht, also wie diese erzeugt und im Keystore aufbewahrt werden. Kapitel 7 geht dann speziell darauf ein, was mit den Schlüsseln innerhalb der Anwendung gemacht werden kann. Hier wird es vor allem um die Verschlüsselung und die digitale Signatur von Daten gehen. In Kapitel 8 widme ich mich anschließend der Implementierung der PKI Funktionalität in Flexible. Hier wird lediglich darauf eingegangen, wie Zertifikatsanfragen erstellt und bearbeitet, Zertifikate ausgestellt und revoziert werden können. Abgeschlossen wird die Arbeit von einem kurzen Ausblick – in Kapitel 9 – auf mögliche, zukünftige Erweiterungen und Verbesserungen an Flexible.

## 2 Java Cryptography Architecture

Die Security API ist ein wichtiger Bestandteil der Programmiersprache Java. Diese konzentriert sich im Großen und Ganzen um das `java.security` Paket und dessen Subpakete. Die API ist entwickelt worden mit dem Ziel, Entwicklern von kryptographischen Anwendungen einfache und allgemeine Schnittstellen zur Verfügung zu stellen, um sowohl auf niedriger, als auch auf hoher Ebene Funktionalität, die der Sicherheit dient, umsetzen zu können.

Dabei sind die Schnittstellen sehr einfach gehalten, um Anwendern die Einbindung der Implementierungen der *Java Cryptography Architecture* (JCA) Schnittstellen – durch das gute Provider Konzept – auf einfachste Weise zu ermöglichen. Jede Implementierung der JCA kann als Provider in der Klasse `java.security.Security` angemeldet werden. Danach macht es keinen Unterschied mehr, welcher Provider welche Art von Algorithmen implementiert, man kann uniform auf die bereitgestellten Mechanismen zugreifen. Dieses Konzept wird im Fokus des *FlexiProviders* (siehe dazu Kapitel 4) noch näher erläutert.

### 2.1 Wichtige Releases

Im Folgenden gehe ich kurz näher auf die wichtigsten Entwicklungen, Funktionalitäten und Releases der JCA ein und beschreibe inwiefern diese für Flexible relevant sind.

- Das erste Release der Security API kam in JDK 1.1 und leitete die *Java Cryptography Architecture* als Framework für die Benutzung und Entwicklung von kryptographischen Modulen in Java ein. Zunächst hatte die JCA im JDK 1.1 nur APIs für digitale Signaturen und für die Berechnung von Hashwerten.
- Eine weitere wichtige Änderung war die Einführung der Klasse für den Java KeyStore (`java.security.KeyStore`) in JDK 1.2. Der Keystore hatte die vorher relativ unübersichtliche identitätsbasierte Schlüsselverwaltung abgelöst. Auch hier wird das Provider Konzept umgesetzt. Die JCA lieferte gleichzeitig einen Provider für den Keystore mit, den *JKS*, der allerdings nur einen relativ schlechten Schutz zur Integrität bietet.
- In den nächsten Releases hatte das Java 2 SDK die Funktionalität der JCA stark erweitert. Es wurden unter anderem Schnittstellen für eine Infrastruktur zur Zertifikatsverwaltung bereitgestellt, die X.509 v3 Zertifikate (definiert in [9]) unterstützt. Die Zugriffsmechanismen wurden dahingehend erweitert, dass diese einen einfacheren und flexibleren Zugriff auf die Schnittstellen ermöglichten.
- Die JCA hat eine gute Provider Architektur, die es ermöglicht mehrere Provider gleichzeitig zu nutzen, die unterschiedliche kryptographische Funktionalität bereitstellen. Damit ist es möglich zwischen verschiedenen kryptographischen Verfahren zu wechseln, ohne zu wissen, welcher Provider diese implementiert. Auf dieser Provider Architektur setzt der *FlexiProvider* auf, weshalb er relativ einfach in eine JCA Anwendung integriert werden kann.

- Die *Java Cryptography Extension* (JCE) wurde als Ergänzung zur JCA in den ersten Versionen des Java 2 SDK eingeführt. Diese war zunächst wirklich nur eine Erweiterung, also nicht fester Bestandteil des SDK und nur optional benutzbar. Ab Java 2 SDK 1.3 wurde sie jedoch fest in die Java Architektur integriert. Die JCE erweitert die JCA unter anderem um APIs zur Verschlüsselung, MAC Berechnung und Schnittstellen für einen sicheren Austausch von symmetrischen Schlüsseln nach dem Diffie-Hellman Verfahren. Zusammen bilden die JCA und die JCE eine vollständige API für kryptographische Anwendungen.
- Die JCE lieferte ebenfalls eine eigene Implementierung eines Keystore Providers (*JCEKS*) mit, die im Gegensatz zum JKS-Provider einen guten Schutz zur Integrität des Keystores bereitstellt. Dieser Provider wird von Flexible genutzt, um das Schlüsselaufbewahrungsproblem zu lösen.

## 2.2 Wichtige Klassen

Ich möchte einige wichtige Klassen der JCA/JCE nicht unerwähnt lassen. Diese werden in Flexible über das sogenannte *Fassade Pattern* [7] benutzt und von den einzelnen Algorithmen im FlexiProvider implementiert.

`javax.crypto.Cipher` Diese JCE Klasse ist für die symmetrische und asymmetrische Verschlüsselung von beliebigen byte Arrays zuständig. Sie wird mit einem symmetrischen (asymmetrischen) Schlüssel (Schlüsselpaar) initialisiert.

`java.security.PublicKey` ist die JCA Klasse für den öffentlichen Teil der Information eines `java.security.KeyPair`.

`java.security.PrivateKey` Der `PrivateKey` ist das Pendant im `java.security.KeyPair`, welches die geheime Information eines Schlüsselpaares darstellt.

`java.security.KeyPairGenerator` Dies ist die JCA Klasse, die ein Schlüsselpaar erzeugt. Initialisiert wird der Generator mit einem Algorithmus und einer Schlüsselgröße oder mit einer Menge von Parametern für den Algorithmus.

`javax.crypto.SecretKey` Dies ist die JCE Klasse, die einen symmetrischen Schlüssel repräsentiert, mit dem unter anderem verschlüsselt werden kann.

`javax.crypto.KeyGenerator` Das ist der Generator, der symmetrische Schlüssel erzeugt. Dieser wird genauso wie der Generator für Schlüsselpaare mit dem Verfahren und der Schlüsselgröße oder Parametern initialisiert.

`java.security.cert.Certificate` Diese abstrakte Klasse repräsentiert ein Zertifikat, welches einen Publickey an den dazugehörigen Benutzer bindet. Hierfür gibt es keine Art Generator. Flexible nutzt zur Erzeugung von Zertifikaten den Fraunhofer CoDec, der anschließend erläutert wird.

`java.security.Signature` Diese Klasse ist für das digitale Signieren von beliebigen Daten geeignet. Sie wird mit dem privaten Schlüssel initialisiert, mit dem signiert werden soll.

`java.security.KeyStore` ist die Java Klasse, die Schlüssel aufbewahren kann. Diese wird in Flexible zum Speichern von symmetrischen Schlüsseln, privaten Schlüsseln und Zertifikaten verwendet. Dazu wird der JCEKS Provider eingesetzt.

`java.security.Security` ist die Klasse, bei der alle Provider angemeldet werden. Diese kann benutzt werden, um die angemeldeten Provider zu befragen, welche Algorithmen diese bereitstellen.

`java.security.MessageDigest` Die Klasse `MessageDigest` berechnet die Funktionswerte von kryptographischen Hashfunktionen.

`java.security.SecureRandom` Diese JCA Schnittstelle bietet die Möglichkeit einen Zufallszahlengenerator zu implementieren, der für die unterschiedlichen Algorithmen zum Beispiel zur Schlüsselerzeugung eingesetzt werden kann.

Man sieht an den Klassennamen, dass die JCA – die im Wesentlichen aus dem Paket `java.security` besteht – die asymmetrischen Verfahren abdeckt und die JCE – die im Wesentlichen um das Paket `javax.crypto` gruppiert ist – diese um die symmetrischen Verfahren und die Verschlüsselung erweitert. Leider ist die Verwendung von Klassen an einigen Stellen etwas unübersichtlich. Es gibt etwa im Falle von `Certificate` sowohl Klassen in der JCA als auch in der JCE, was die Benutzung schwieriger macht.

Das einfache und flexible Konzept der JCA/JCE hat auch einige Schwachstellen. Dadurch, dass alles sehr variabel und einfach gehalten ist und für alle möglichen Arten von Algorithmen verwendet werden kann, ist es nicht möglich spezielles Verhalten von Algorithmen zu erfassen und gegen andere Algorithmen abgrenzen zu können. Das Problem, das sich dadurch stellt und in dieser Arbeit behandelt wird, ist die Parametrisierung von Algorithmen.

Auf eine detailliertere Beschreibung der Funktionsweise der einzelnen Klassen und deren Methoden und an welchen Stellen diese Klassen in Flexible eingesetzt werden, gehe ich in den entsprechenden Unterkapiteln zu Flexible näher ein.

## 3 Fraunhofer FhG ASN.1 CoDec

CoDec ist ein Java Paket, für das Kodieren und Dekodieren von ASN.1 Datenstrukturen. Es wurde am Fraunhofer *Institut Graphische Datenverarbeitung* in Darmstadt entwickelt, als Teil des SeMoA Projektes [4] und ist open source.

Das Projekt stellt eine einfach zu benutzende API zur Verfügung, die einige wichtige Standards für Public-Key-Kryptographie und Zertifikate implementiert.

### 3.1 Public Key Cryptography Standards (PKCS)

PKCS steht für *Public Key Cryptography Standards* und bezeichnet eine Reihe von kryptographischen Spezifikationen. Diese wurden von den RSA Laboratorien in Zusammenarbeit mit anderen ab 1991 entwickelt [1]. Die Entwicklung geschah mit dem Ziel, die Verbreitung der Public-Key-Kryptographie zu beschleunigen. Die Standards, die von CoDec implementiert werden, sind:

**PKCS#1** : *RSA Cryptography Standard*. Definiert das Format der RSA Verschlüsselung.

**PKCS#7** : *Cryptographic Message Syntax Standard*. Bildet die Basis für S/MIME und wird zum Signieren und/oder Verschlüsseln von Nachrichten einer PKI genutzt.

**PKCS#8** : *Private-Key Information Syntax Standard*. Dieser Standard beschreibt eine Syntax für einen Privatekey und dazugehörige Attribute, ebenso wie für verschlüsselte private Keys.

**PKCS#10** : *Certification Request Syntax Standard*. Format der Nachrichten, die zu einer Zertifizierungsstelle gesendet werden, um die Zertifizierung eines öffentlichen Schlüssels zu erfragen.

**PKCS#12** : *Personal Information Exchange Syntax Standard*. Definiert ein Dateiformat, das dazu benutzt wird, private Schlüssel mit dem zugehörigen Zertifikat mit Schutz für Integrität und Vertraulichkeit zu speichern.

### 3.2 X.500 Standards

**X.501** : Directory: *Models* und

**X.509** : Directory: *Public-key and attribute certificate frameworks* ist ein ITU-T Standard für eine Public-Key-Infrastruktur und derzeit der wichtigste Standard für digitale Zertifikate. Die aktuelle Version ist X.509 v3.

CoDec enthält einen Base64 Encoder und Decoder, die in Flexible dazu benutzt werden, um Zertifikate, Zertifikatsanfragen und Zertifikatssperllisten im DER und PEM Format (siehe dazu 8.2) kodiert zu exportieren und nach dem Importieren wieder zu dekodieren.

Auf die anderen verwendeten Klassen des CoDec gehe ich im Kapitel 8 näher ein.

## 4 FlexiProvider

Der *FlexiProvider* ist eine modular aufgebaute und erweiterbare Implementierung der *Java Cryptography Architecture* (JCA) und der *Java Cryptography Extension* (JCE). Diese Implementierung ist vor allem auf die Effizienz der umgesetzten Algorithmen optimiert. Public-Key-Anwendungen, die auf den JCA/JCE Schnittstellen aufsetzen, können die einzelnen Module und die umgesetzten Algorithmen des FlexiProviders einfach benutzen.

Zur Zeit wird der FlexiProvider am Fachbereich Informatik der Technischen Universität Darmstadt von der *Cryptography and Computeralgebra* Forschungsgruppe, die von Prof. Dr. Johannes Buchmann geleitet wird, entwickelt. Alle weiteren Informationen und die aktuelle Version des Providers können der aktuellen Webseite [2] entnommen werden.

Das Ziel des Projektes ist es, eine effiziente und einfach zu benutzende Implementierung von kryptographischen Verfahren bereitzustellen.

Der FlexiProvider bietet unter anderem folgende, für Flexible relevante, Funktionen:

- Standard Verfahren wie RSA, DSA und ElGamal (zu finden im *FlexiCore* Provider)
- Innovative Techniken wie die Kryptographie mit Elliptischen Kurven und Quadratischen Formen (zu finden im *FlexiEC* und *FlexiPQC* Provider)
- Symmetrische Verschlüsselungsverfahren wie DES, DESede, IDEA, RC2, RC4, RC5, RC6, MARS, SERPENT, TWOFISH, RIJNDAEL, E2, SaferPlus (zu finden im *FlexiCore* Provider)
- kryptographische Hashfunktionen wie MD4/5, DHA256, SHA1/224/256/384/512, Ripemd 128/160 etc. (zu finden im *FlexiCore* Provider)

Der FlexiProvider ist im Grunde eine Sammlung von mehreren JCA/JCE Providern. Die folgenden drei Provider werden zur Zeit von Flexible unterstützt. Der *FlexiCore* Provider ist für alle symmetrischen und asymmetrischen Standard Verfahren und Hashfunktionen zuständig. Der *FlexiEC* Provider implementiert die Elliptische-Kurven-Kryptographie und der *FlexiPQC* Provider, der relativ neue Verfahren, von denen viele nicht standardisiert aber beweisbar sicher gegen bestimmte Angriffsarten sind, implementiert. Wenn im Folgenden von *FlexiProvider* die Rede ist, dann meine ich im Allgemeinen einen der drei Provider, wobei es keine Rolle spielt welchen.

Die Benutzung des FlexiProviders in einem JCA Hintergrund ist durch den bereits erwähnten Provider Mechanismus denkbar einfach und geschieht durch die folgende Zeile (hier exemplarisch für den Fall des *FlexiCore* Providers).

```
1 Security.addProvider(new de.flexiprovider.core.FlexiCoreProvider());
```

Listing 1: Anmelden des FlexiCoreProvider in der Security Klasse

Dadurch wird der Provider in der JCA Klasse `java.security.Security` registriert und kann uniform zu allen anderen angemeldeten Providern – wie dem von SUN, der standardmäßig registriert ist – verwendet werden. Dadurch werden alle Algorithmen, die der FlexiProvider zur Verfügung stellt im JCA/JCE Kontext verfügbar gemacht.

In Abbildung 1 ist dargestellt, wie die Architektur einer Anwendung aussieht, die auf den Schnittstellen der JCA und deren Erweiterung, basiert. Dies ist hier für den speziellen Fall von Flexible dargestellt, das neben den Schnittstellen der JCA und JCE noch auf die Klassen aus CoDec zugreift.

Man sieht in der Abbildung den *SUN* Provider, den die JCA standardmäßig mitliefert. Dieser Provider implementiert lediglich die JCA Schnittstellen. Die JCE stellt dem Benutzer auch einen eigenen Provider zur Verfügung, den *SunJCE* Provider. Dieser kann zum Beispiel mit dem RSA Verfahren [11] verschlüsseln und beherrscht darüber hinaus einige symmetrische Standard Verfahren.

Der *FlexiProvider* implementiert sowohl die JCA, als auch die JCE Schnittstellen. Dadurch bietet er dem Benutzer eine vollständige Bibliothek mit kryptographischen Funktionen. Der Übersicht halber ist hier die Abhängigkeit des FlexiProviders zum Fraunhofer CoDec nicht eingezeichnet, den der FlexiProvider für die ASN.1 Kodierung verwendet.

Alle Pfeile, die von der Anwendung selbst ausgehen, beziehen sich lediglich auf die beiden Security APIs (bis auf den Pfeil zum CoDec im speziellen Fall von Flexible). Das hat den großen Vorteil, dass man die Implementierung des entsprechenden Providers nicht kennen muss. Die Schnittstellen abstrahieren sehr gut von Implementierungsdetails, die dem Benutzer des Providers verborgen bleiben. Der Benutzer muss lediglich den Namen des Providers kennen und den richtigen Provider – wie oben beschrieben – registrieren.

Im Falle von Flexible sind alle ausgehenden Pfeile durch *Fassaden* realisiert, so dass eine hohe Erweiterbarkeit der Anwendung und Transparenz bei der Benutzung gewährleistet ist.

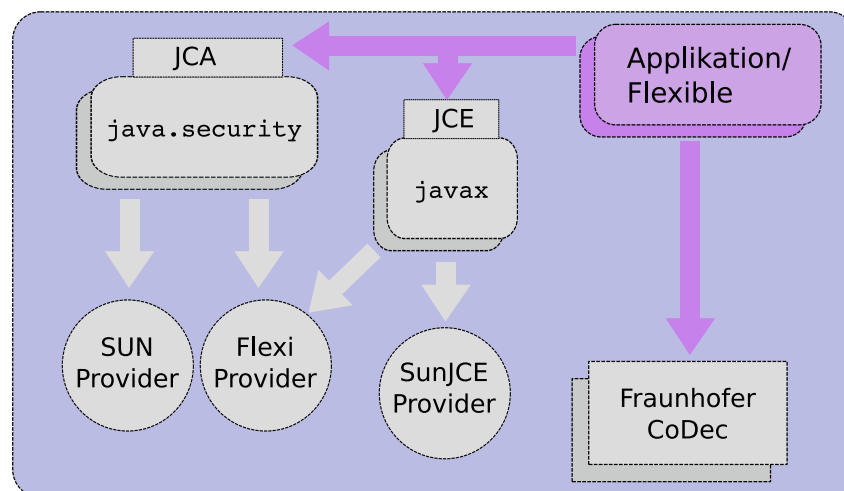


Abbildung 1: Die JCA/JCE Architektur

## 5 Flexible

Der Name Flexible ist ein Hinweis auf den verwendeten *FlexiProvider* und zugleich eine Andeutung auf die hohe Flexibilität, die die Benutzerschnittstelle laut Aufgabenstellung bieten soll. Die Anwendung war erstmals ein Projekt der Gruppe SAMD, das in einem Bachelorpraktikum des Fachgebiets *Kryptographie und Computeralgebra* am Fachbereich Informatik der Technischen Universität Darmstadt im Sommersemester 2006 entstanden ist. Es wurde damals in der ersten Iteration (v1.0.1) abgeschlossen und vom Fachgebiet abgenommen. Die aktuelle Version von Flexible ist 1.1 und befindet sich zur Zeit auf einem nicht öffentlichen SVN Server des Fachgebiets in der Revision 903 vom 22.07.2007. Es ist plattformunabhängig, da bei der Programmierung in Java und SWT darauf geachtet und mehrmals darauf getestet wurde, dass die Anwendung auf allen unterstützten Plattformen das gleiche Verhalten zeigt.

Flexible setzt das Vorhandensein des kompilierten FlexiProviders und der FlexiProvider Quellen voraus. Standardmäßig wird die FlexiProvider Version 1.4p3 zusammen mit der Applikation ausgeliefert. Im Folgenden beschreibe ich kurz, wie man die Anwendung auf einem der drei unterstützten Betriebssysteme installiert und welche Voraussetzungen an die jeweilige Umgebung gestellt werden.

### 5.1 Systemvoraussetzungen für Flexible

Folgendes sollte auf dem System, auf welchem Flexible ausgeführt werden soll, vorhanden sein:

- Java Runtime Environment (JRE) 5.0 oder höher
- Aktuelle kompilierte Version und Quellen des FlexiProviders (falls die mitgelieferte Version nicht genutzt werden soll)
- Ein Programm mit dem zip Archive entpackt werden können
- Eines der folgenden Betriebssysteme
  - Windows 98/2000/XP
  - Linux
  - Mac OS X

### 5.2 Installation von Flexible

Flexible liefert alle für die Ausführung benötigten Pakete mit. Zum Installieren der Anwendung muss lediglich der Inhalt des zip Archivs an eine geeignete Stelle entpackt und einige Einstellungen, die im Folgenden beschrieben werden, vorgenommen werden.

1. Das zip Archiv, das die Flexible jar enthält herunterladen
2. In einen Ordner, der die nötigen Rechte hat entpacken

3. Gegebenenfalls den Pfad zu der `algorithms.xml` und den Pfad zu den FlexiProvider Quellen in der `config` Datei anpassen
4. Die Anwendung mit dem entsprechenden Skript starten

**Windows** Flexible\_1.1\_Win32.bat

**Linux** Flexible\_1.1\_Linux.sh

**Mac OS X** Flexible\_1.1\_MacOSX.sh

Des Weiteren muss gegebenenfalls der JCA Patch für unbegrenzte Schlüssellängen heruntergeladen und installiert werden, denn sonst werden etwa bei RSA [11] nur Schlüsselgrößen mit einem Modul bis 1024 Bit unterstützt.

Dieser ist für Java 5 auf [http://java.sun.com/javase/downloads/index\\_jdk5.jsp](http://java.sun.com/javase/downloads/index_jdk5.jsp) im Bereich "Other Downloads" zu finden.

### 5.3 Ersetzen des FlexiProviders durch eine andere Version

Falls eine neuere Version des FlexiProviders verfügbar ist oder aus irgendeinem Grund eine ältere Version des Providers die aktuelle mit Flexible ausgelieferte Version ersetzen soll, sind folgende Schritte auszuführen.

1. Der kompilierte FlexiProvider (Version mit allen Providern) und die Quellen des FlexiProviders müssen von [2] heruntergeladen werden und in den Flexible Ordner kopiert werden
2. Die `config` Datei im Flexible Verzeichnis muss editiert werden. Dort muss im Eintrag `flexiproviderpath` der relative Pfad zu dem zip Archiv mit den FlexiProvider Quellen angegeben werden
3. Die Zeile in dem jeweiligen Starterskript muss wie in Listing 2 angegeben, angepasst werden:

#### Windows

```

1 "java -Djava.library.path=
2 ./lib/windows -cp Flexible_1.1.jar ; ./lib/windows/swt.jar ;
3 ./lib/FlexiProvider-VERSION.signed.jar de.samd.gui.Starter"

```

#### Linux

```

1 "java -Djava.library.path=
2 ./lib/linux -cp Flexible_1.1.jar : ./lib/linux/swt.jar :
3 ./lib/FlexiProvider-VERSION.signed.jar de.samd.gui.Starter"

```

#### Mac OS X

```

1  " java -XstartOnFirstThread -Djava.library.path=
2  ./lib/macosx -cp Flexible_1.1.jar:./lib/macosx/swt.jar:
3  ./lib/FlexiProvider-VERSION.signed.jar de.samd.gui.Starter"

```

Listing 2: Starterskript für Flexible anpassen

**VERSION** ist durch die jeweilige Version des FlexiProviders zu ersetzen

Nach dem Austauschen des FlexiProviders kann es sein, dass einige Algorithmen in einem der Provider hinzugekommen sind oder entfernt wurden. Um die Liste der Algorithmen zu aktualisieren muss Flexible gestartet werden. Im Menü wird *Algorithms* -> *Create new algorithms XML* ausgewählt und mit *Ja* bestätigt. Achtung: wenn man mit *Ja* bestätigt, wird die vorhandene XML Datei mit Algorithmen überschrieben und somit auch alle initialen Einstellungen, wie die Schlüsselgrößen für Algorithmen.

## 5.4 Implementierung einer Algorithmenstruktur

Um einfacher auf die im FlexiProvider vorhandenen Algorithmen zugreifen und diese benutzen zu können, wurde in Flexible eine Algorithmenstruktur implementiert, die alle nötigen Informationen zu den Algorithmen enthält. Diese Informationen werden nach dem Einlesen in eine XML Struktur geschrieben, da das Einlesen rechenzeitaufwändig ist. Nachdem die Informationen einmal eingelesen wurden, wird nur noch auf die Algorithmenstruktur mittels der *algorithms.xml* zugegriffen. Falls diese Informationen veraltet sind, kann diese Datei über das Menü in Flexible aktualisiert werden (siehe dazu vorherigen Abschnitt).

Die Klassen, die die Algorithmenstruktur darstellen, befinden sich in dem Java Paket *de.samd.algorithm*. Das UML Diagramm in Abbildung 2 zeigt den Aufbau des Pakets und der Klassen. Die Klasse *Algorithm* repräsentiert dabei einen Algorithmus. Hier

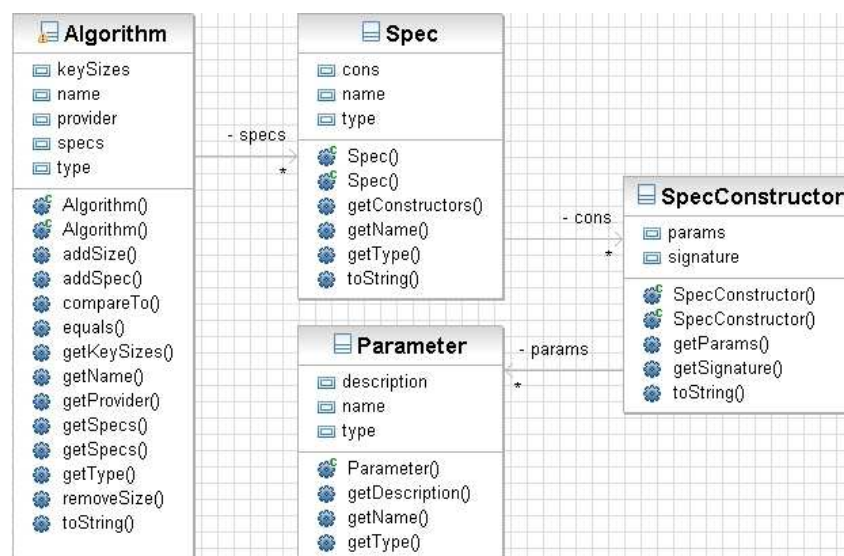


Abbildung 2: Algorithmen Struktur in Flexible

gibt es Felder für den Namen des Algorithmus, den Provider, der diesen Algorithmus

anbietet (*FlexiCore*, *FlexiEC* oder *FlexiPQC*) und den Typ des Algorithmus. Flexible unterscheidet drei verschiedene Typen von Algorithmen: die symmetrischen (*symmetric*), die asymmetrischen (*asymmetric*) und die Algorithmen, die Hashwerte berechnen (*md*). Hier findet keine Unterscheidung statt, ob der Algorithmus zum Signieren oder zum Verschlüsseln (oder auch beides wie im Falle von RSA) gedacht ist. Diese Unterscheidung stellt ein nicht triviales Problem dar, da sie nicht direkt über die JCA/JCE Schnittstellen geschehen kann. Wie diese Unterscheidung gemacht wird, wird in Kapitel 7 beschrieben.

Desweiteren enthält die Klasse eine `Integer` Liste *keySizes*, die alle unterstützten Schlüsselgrößen speichert. Diese Information kann ebenfalls nicht über die JCA/JCE ausgelesen werden und stellt sozusagen Expertenwissen dar. Flexible wird mit einer Liste von unterstützten Schlüsselgrößen ausgeliefert. Falls der Anwender weiß, dass eine weitere Schlüsselgröße für ein Verfahren unterstützt wird, kann er diese im Menü über *Algorithms* -> *Edit algorithms XMLfile* hinzufügen.

Es gibt hier auch eine Liste für die Parametermenge, die dieser Algorithmus zur Parametrisierung verwenden kann. Diese werden mit Hilfe eines Javadoc Parsers in den FlexiProvider Quellen ausgelesen, da die JCA/JCE Schnittstellen hier ebenfalls keine Zuordnung vorsehen.

Die Klasse `Spec` ist die Repräsentation einer Parametermenge. Diese Menge kann einen Namen haben und hat den Typ `AlgorithmParameterSpec` beziehungsweise `KeySpec`. Der erste Typ dient zur Parametrisierung von Algorithmen und der zweite zur Erstellung von Schlüsseln. Die Klasse enthält eine Liste von Konstruktoren, die diese Parametermenge erzeugen.

Die Klasse `SpecConstructor` repräsentiert einen Konstruktor für eine Parametermenge. Sie hat ein Feld für die Signatur des Konstruktors, die als Information für den Benutzer dient, damit dieser weiß, wie viele Parameter der Konstruktor akzeptiert. Die Parameterliste gibt an, welche und wieviele Parameter der Konstruktor braucht.

Die Klasse `Parameter` ist die Repräsentation eines Parameters, der im Konstruktor verwendet werden darf. Der Parameter hat einen Namen, einen Datentyp und die Javadoc Beschreibung, die aus den FlexiProvider Quellen ausgelesen wird.

#### 5.4.1 Einlesen der Informationen aus dem FlexiProvider

Um diese Struktur mit den richtigen Daten zu füllen, müssen diese aus dem FlexiProvider erst einmal ausgelesen werden. Die grafische Oberfläche soll so flexibel und erweiterbar wie möglich gehalten werden. Dazu müssen die Algorithmen dynamisch ausgelesen und dürfen nicht im Quelltext der Anwendung „hardcodiert“ werden. Es sollen alle Algorithmen mit unterschiedlichen Parametrisierungen durch Flexible unterstützt werden.

Das Problem liegt in dem Konzept der Schnittstellen der JCA/JCE: dadurch, dass viele verschiedene Algorithmen unterstützt werden sollen, sind die Schnittstellen sehr breit, um möglichst alle denkbaren kryptographischen Verfahren umzusetzen. Deswegen kann man allerdings auf bestimmte Details von Algorithmen nicht zugreifen. Beim RSA Verfahren zum Beispiel reicht es, den Schlüsselgenerator nur mit der Größe des RSA-Moduls zu initialisieren. Wohingegen der Benutzer etwa beim ECDSA Verfahren [10] unter Umständen noch die OID der elliptischen Kurve, die benutzt werden soll, mit angeben will. Da

die Schnittstelle für beide Algorithmen die gleiche ist (`KeyPairGenerator`), ist es nicht möglich zwischen diesen Verfahren zu unterscheiden. Deswegen sieht die JCA/JCE die Möglichkeit der Parametrisierung von Schlüssel-Generatoren, Cipher Objekten und weiteren Objekten mit Parametermengen vor.

Zunächst einmal müssen alle Algorithmen im `FlexiProvider` ausgelesen werden. Dies geschieht auf folgende Weise: erst wird der entsprechende Provider geholt, der in der `Security` Klasse registriert sein muss. Hier am Beispiel des *FlexiCore* Providers.

```
1 Provider p = java.security.Security.getProvider(" FlexiCore");
```

Anschließend werden alle Algorithmen geholt, die der Provider zur Verfügung stellt. Diese sind in der Datenstruktur `java.security.Provider.Service` gespeichert.

```
1 Set<Service> services = p.getServices();
2 Iterator<Service> it = services.iterator();
```

Mit dem `Iterator` kann jetzt die Menge der Services durchlaufen werden. Dabei werden die Services nach dem Typ des bereitgestellten Verfahrens in drei Kategorien eingeordnet und in die Algorithmenliste eingelesen.

```
1 List<Algorithm> algos = new ArrayList<Algorithm>();
2 while (it.hasNext()) {
3     s = it.next();
4     Algorithm alg = null;
5
6     if (s.getType().equals(" MessageDigest")) {
7         alg = new Algorithm(s.getAlgorithm(), "md", provider.getName());
8     } else if (s.getType().equals(" KeyGenerator")) {
9         alg = new Algorithm(s.getAlgorithm(), "symmetric",
10                             provider.getName(), getSpecs(s.getAlgorithm()));
11     } else if (s.getType().equals(" KeyPairGenerator")) {
12         alg = new Algorithm(s.getAlgorithm(), "asymmetric",
13                             provider.getName(), getSpecs(s.getAlgorithm()));
14     }
15
16     algos.add(alg);
17 }
```

Listing 3: Auslesen der Algorithmen aus dem `FlexiProvider`

Hier wird die oben angesprochene Unterscheidung gemacht. Falls der Algorithmus vom Typ `KeyGenerator` ist, dann ist es ein symmetrischer Algorithmus, denn mit ihm können symmetrische Schlüssel erzeugt werden. Das gleiche gilt für die asymmetrischen Algorithmen und den `KeyPairGenerator`. Die dritte Möglichkeit (im Listing 3 die erste Abfrage) ist, dass der Algorithmus einen Hashwert berechnet. Für die Fälle symmetrisch und asymmetrisch wird außerdem eine Funktion aufgerufen, die den Algorithmen die richtige Klasse mit der Parametermenge zuordnet. Anschließend werden die Algorithmen in einer Liste mit der Algorithmen Struktur gespeichert.

**Parametrisierung von Algorithmen** Nachdem alle Algorithmen ausgelesen wurden, fehlt die Information über die mögliche Parametermenge, die einem `KeyPairGenerator` zur Erzeugung des Schlüssels mit dem angegebenen Verfahren übergeben werden kann.

Wie bereits erwähnt kann diese Information leider nicht direkt aus der JCA/JCE Struktur extrahiert werden. Dazu verwendet Flexible direkt die Quelltexte des FlexiProviders. Es werden darin die richtigen Klassen gesucht und in diesen der Javadoc Eintrag zu den verschiedenen Konstruktoren ausgewertet, um die Beschreibung der Parameterliste zu erhalten.

Die beiden Klassen, die für das Einlesen der Algorithmeninformationen zuständig sind liegen im Paket `de.samd.initialize`. Das UML Diagramm in Abbildung 3 zeigt die beiden Klassen. Die Klasse `InitAlgorithms` liest alle Algorithmen des FlexiProviders aus, indem

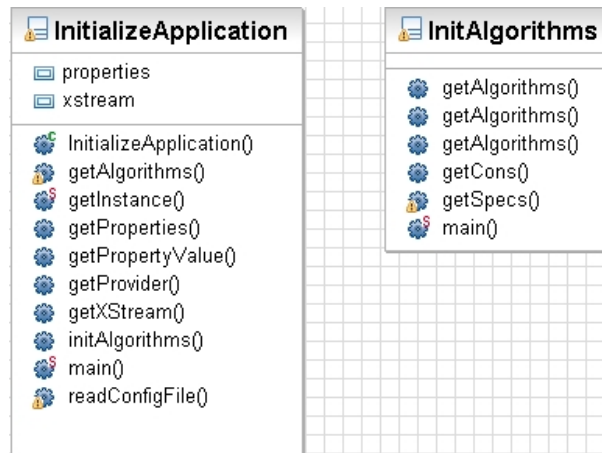


Abbildung 3: Paket `de.samd.initialize`

erst für alle drei Provider das oben beschriebene Verfahren angewendet wird. Anschließend parst diese Klasse die Javadoc Informationen, um die Parametermenge zuzuordnen zu können, was im nächsten Abschnitt beschrieben wird. Die Klasse hat eine Methode (`getAlgorithms(String provider, String type)`), die es ermöglicht, Algorithmen von nur einem bestimmten Typen und aus einem bestimmten Provider auszulesen.

Die Klasse `InitializeApplication` ist eine *Singleton* [7] Instanz. Diese übernimmt das Schreiben, der durch `InitAlgorithms` ausgelesenen Algorithmen Struktur, in eine XML Struktur.

Im Folgenden gehe ich nur auf die `AlgorithmParameterSpec` ein, da Flexible zur Zeit keine Verwendung für die `KeySpec` hat.

**AlgorithmParameterSpec** Die Schnittstelle `AlgorithmParameterSpec` ist leer. Genau hierin liegt das Problem mit der Parametrisierung. Dadurch, dass die Schnittstelle leer ist, kann man Algorithmen im Grunde mit jeder Art von Parametern parametrisieren. Dies erhöht die Flexibilität, da für jeden neuen Algorithmus eine Klasse geschrieben werden kann, die diese Schnittstelle implementiert. Somit bleiben die Generatoren unverändert und allgemeingültig. Allerdings hat dies zur Folge, dass man keine Rückschlussmöglichkeit mehr auf die Spec hat, wenn man nur den Algorithmus kennt. Man weiß nur, dass man den Algorithmus mit einer `AlgorithmParameterSpec` parametrisieren darf, aber nicht, welche Parameter darin festgelegt sein müssen. Man muss jetzt also auf die konkrete Implementierung der `AlgorithmParameterSpec` gelangen um feststellen zu können, welche Parameter gültig sind. Die flexibelste und einfachste Möglichkeit

scheint hier, die Auflösung über den Namen zu gestalten. Weiterhin muss man nach dem Finden der richtigen Spec die Parameter (Typ und Javadoc Kommentar) aus der Klasse extrahieren.

**Zuordnung von Algorithmus zu AlgorithmParameterSpec** Flexible geht von der Konvention aus, dass alle Specs, die zur Parametrisierung benutzt werden sollen, die Schnittstelle `AlgorithmParameterSpec` implementieren müssen. Desweiteren muss der Klassenname einer Spec mit dem Namen des Algorithmus – für den diese Spec gelten soll – beginnen und mit "ParameterSpec" enden.

Die Methode, die diese Zuordnung durchführt, ist in der Klasse `InitAlgorithms` implementiert und hat die Signatur `List<Spec> getSpecs(String algoname)`. Hier werden die Klassen und deren Methoden aus dem Paket `java.util.zip` genutzt, um das zip Archiv mit den Quellen des FlexiProviders zu durchlaufen und die richtige Klasse mit den Konstruktoren und deren Javadoc Kommentar zu finden. Enthält ein Eintrag den Algorithmennamen und endet mit `ParameterSpec`, so ist dies eine Klasse, die eine gültige Parametermenge für diesen Algorithmennamen darstellt.

Nachdem die richtige Klasse gefunden wurde, werden per Java Reflection die Schnittstellen der Klasse geholt und durchlaufen. Es wird für jede Schnittstelle geschaut, ob diese die Schnittstelle `AlgorithmParameterSpec` implementiert. Falls dies der Fall ist, wird diese Spec der Liste von Parametermengen dieses Algorithmus hinzugefügt. Dabei wird vorher noch die Methode zum parsen der Parameter und Konstruktoren aufgerufen.

**Parsen der Javadoc Daten** Für das Parsen der Javadoc Daten im Sourcecode der gefundenen Spec Klasse wird der Parser *QDox* [6] eingesetzt. QDox ist ein schneller und kleiner Parser zum Extrahieren von Klassen, Schnittstellen und Methoden Definitionen aus dem Sourcecode, welche mit den Javadoc *@tags* versehen sind. Das Projekt steht unter der Apache Software Lizenz, Version 2.0.

Der FlexiProvider muß sich an die gewöhnlichen Javadoc Konventionen zur Kommentierung von Klassen Konstruktoren halten. Nur so kann eine richtige Benutzung der Parameter gewährleistet sein. Hier beispielhaft für den Fall einer im FlexiProvider vorhandenen `DESedeParameterSpec`:

```

1  /**
2   * @param iv
3   *         the buffer with the IV
4   * @param offset
5   *         the offset in iv, where the IV starts
6   * @param len
7   *         the number of IV bytes
8   */
9  public DESedeParameterSpec(byte[] iv, int offset, int len) {
10     this.iv = new byte[len];
11     System.arraycopy(iv, offset, this.iv, 0, len);
12 }

```

Listing 4: Javadoc Konvention für Implementierungen von `AlgorithmParameterSpec`

Die Methode `List<SpecConstructor> getCons(String zipentryName)` in der Klasse

`InitAlgorithms` übernimmt das Parsen dieser Kommentare. Dazu wird jeder Kommentar zerlegt in den Namen des Parameters, der der erste Eintrag nach dem `@param` Tag sein muss und in die Beschreibung dieses Parameters, die den zweiten Eintrag im Tag darstellt. Zusätzlich kann QDox die Signatur einer Methode zerlegen. Damit bekommt man die Typen der einzelnen Parameter. Nachdem man dadurch alle Informationen hat, werden die Parameter der Parameterliste des `SpecConstructor` hinzugefügt.

### 5.4.2 Speichern der ausgelesenen Informationen als XML

Nachdem alle Informationen in die Algorithmenstruktur (Liste vom Typ `Algorithm`) eingelesen wurden, werden diese Informationen persistent abgespeichert, um sie nicht jedesmal auslesen zu müssen – denn das Auslesen ist sehr rechenintensiv – und um anschließend per Hand, Änderungen vornehmen zu können, wie etwa das Hinzufügen von unterstützten Schlüsselgrößen bei symmetrischen und asymmetrischen Verfahren. Dazu wird eine XML Struktur für die Daten verwendet, da diese gut strukturiert ist und deshalb einfach weiterverarbeitet werden kann.

Um die Algorithmen Struktur in eine XML Struktur umzuwandeln nutzt Flexible das open source Paket XStream [5]. Diese Bibliothek erlaubt es auf einfache Weise Java Objekte in XML zu serialisieren und umgekehrt. XStream ist unter der Open-Source-Lizenz frei verfügbar. Die Methode `getAlgorithms(String type, boolean forceWrite)` in der Klasse `InitializeApplication` (siehe Abbildung 3) gibt eine Liste von Algorithmen zurück, die vom Typ `type` sind. Wenn `forceWrite` mit `true` übergeben wird, dann wird in jedem Fall die `algorithms.xml` Datei (die sich im in der config angegebenen Pfad befindet) neu geschrieben. Falls `false` übergeben wird, wird diese Datei nicht überschrieben, außer diese existiert nicht, dann wird sie erzeugt.

Nachdem die XML Datei gespeichert wurde, kann sie mit XStream wieder in Java Objekte zurück serialisiert werden.

### 5.4.3 Verbliebene Probleme

Dieser Ansatz ist so in Flexible umgesetzt. Falls die Konventionen mit dem Namen und der Javadoc im FlexiProvider eingehalten werden, werden alle Informationen richtig ausgelesen und gespeichert. Das Problem ist jedoch an anderer Stelle: die Datentypen der Spec Konstruktoren werden erst zur Laufzeit bekannt, da diese aus der XML Datei, beziehungsweise aus dem Provider ausgelesen werden. Das heißt, es ist nicht möglich, eine Benutzereingabe, die vom Datentyp `String` ist, in den richtigen Parametertyp zu `casten`. Es ist zur Laufzeit zwar bekannt, wieviele und welche Parameter, ein Konstruktor akzeptiert, jedoch gibt es in Java keine Möglichkeit, den `cast` dynamisch durchzuführen. Das heißt bei einem `cast` muss der explizite Typ zur Compilezeit bekannt sein.

Dieses Problem lässt sich leider auch nicht durch Java Reflections in den Griff bekommen. Hiermit kann man zwar den Konstruktor der Klasse benutzen, jedoch müssen die Typen der Parameter des Konstruktors wiederum zur Compilezeit bekannt sein.

Es gibt zwei Möglichkeiten zur Lösung des Problems.

**Die erste Lösung** ist die Einfachere, jedoch extrem unflexible und deshalb nicht umge-

setzte. Es werden alle unterstützten Datentypen – mit dem *instanceof* Operator von Java – zentral überprüft und anschließend statisch herunter gecastet. Der Nachteil ist, dass alle in den Specs verwendeten Datentypen abgefragt werden müssten. Neu hinzukommende Klassen, die als Parametertypen dienen, könnten so nicht verwendet oder müssten per Hand an den entsprechenden Stellen eingefügt werden.

Dadurch müsste Flexible selbst angepasst und neu kompiliert werden. Alte Versionen von Flexible würden dann nicht mehr mit neueren Providern zusammenarbeiten können und umgekehrt könnte man in einer neueren Version von Flexible keinen älteren Provider benutzen.

**Die zweite Lösung** erfordert viel Änderungsaufwand im FlexiProvider selbst, ist jedoch viel flexibler. Jede Spec müsste um einen Konstruktor erweitert werden, der ein Array von Strings akzeptiert. Dieser Konstruktor kann dann selbst abgleichen, ob die Parameter gültig sind und gegebenenfalls an den richtigen Konstruktor weiterleiten. Dadurch muss zwar jede Spec Klasse erweitert werden, jedoch ist dieser Ansatz viel „sauberer“ als der Erste. Beim Hinzufügen von neuen Typen in einem Spec Konstruktor müsste nur der Konstruktor dieser Klasse angepasst werden, alle anderen bleiben davon unbetroffen. Flexible selbst bleibt dadurch von Änderungen im FlexiProvider unberührt. Alte Versionen von Flexible können so unverändert mit neueren Providern arbeiten.

Leider war der zeitliche Rahmen in dem diese Arbeit entstanden ist, nicht ausreichend, um die große Änderung, die der zweite Lösungsansatz mit sich bringt, im FlexiProvider zu realisieren. Bei Nachfolgeversionen kann jedoch die bestehende Architektur von Flexible leicht erweitert werden, um diesen Lösungsansatz umzusetzen.

## 6 Schlüsselgenerierung und Schlüsselverwaltung

Schlüssel stellen in der Kryptographie eine Information dar, die entweder öffentlich verfügbar ist oder geheim gehalten wird. Diese Information kann dazu verwendet werden, kryptographische Anwendungen wie Signaturen und Verschlüsselung zu verwenden (siehe Kapitel 7). Die Anwendung Flexible basiert auf zwei Arten von Schlüsseln: die Schlüssel für die symmetrischen Verfahren (im Folgenden auch als geheime Schlüssel oder *secret keys* bezeichnet) und die Schlüsselpaare für die asymmetrischen Verfahren (auch als *Keypairs* bezeichnet). Schlüsselpaare bestehen dabei aus je einem öffentlichen Schlüssel (den *Publickey*) und einem zugehörigen privaten Schlüssel (*Privatekey*). In einer PKI (siehe Kapitel 8) ist es gängig, ein Zertifikat für den Publickey zu erstellen, welches aussagt, dass die im Zertifikat angegebene Person auch über den zugehörigen Privatekey verfügt. Es dient also dazu, den Besitz (*proof-of-possession*) für den Privatekey sicherzustellen.

Die Datenstruktur, in der Flexible alle Schlüssel und Zertifikate verwaltet und aus der heraus es die Schlüssel anwendet, stellt die Java Klasse `java.security.KeyStore` dar.

### 6.1 Generierung von Schlüsseln

In Flexible können zwei Arten von Schlüsseln erzeugt werden. Die geheimen symmetrischen Schlüssel, für die im FlexiProvider implementierten symmetrischen Verschlüsselungsverfahren und die Schlüsselpaare für die asymmetrischen Verschlüsselungsverfahren und Signaturverfahren. Die Generatorklasse, die in Flexible diese Schlüssel erzeugt, heißt `de.samd.functions.key.KeyManager`.

#### 6.1.1 Symmetrische Schlüssel

Zum Erzeugen von *secret keys* wird die JCE Klasse `javax.crypto.KeyGenerator` benutzt. Dabei sind im Prinzip die folgenden drei Schritte notwendig, um einen Schlüssel zu erzeugen:

1. Über die Factory Methode `getInstance()` wird eine Instanz der Klasse geholt. Diese Methode wird mit dem Algorithmus aufgerufen, für den der Schlüssel erstellt werden soll;
2. Anschließend kann der `KeyGenerator` optional mit Parametern initialisiert werden über die `init()` Methode;
3. Zuletzt wird der symmetrische Schlüssel mittels `generateKey()` erzeugt.

Folgende Funktionen im `KeyManager` von Flexible erzeugen symmetrische Schlüssel:

**`generateSecretKey(String algorithm, int strength)`** Erzeugt einen symmetrischen Schlüssel für den Algorithmus, dessen Name in *algorithm* übergeben wird. Dieser Schlüssel wird für die Schlüsselgröße *strength* erzeugt.

**`generateSecretKey(String algorithm)`** ruft die obige Methode auf mit `strength = -1`. Dadurch wird der Schlüssel für die Standardgröße, die im FlexiProvider definiert ist, erzeugt.

**generateSecretKey(String algorithm, AlgorithmParameterSpec spec)** Alternativ kann der Schlüsselgenerator auch mit einer Parametermenge, passend zu diesem Algorithmus, initialisiert werden.

### 6.1.2 Asymmetrische Schlüssel

Zum Erzeugen von Schlüsselpaaren wird die Klasse `java.security.KeyPairGenerator` verwendet. Die Schritte zur Erzeugung von Schlüsselpaaren gleichen im Prinzip denen von symmetrischen Schlüsseln.

Folgende Funktionen im `KeyManager` von Flexible erzeugen Schlüsselpaare:

**generateKeyPair(String algorithm, int keyStrength)** erzeugt ein Schlüsselpaar für das asymmetrische Verfahren *algorithm* mit der Schlüsselgröße *keyStrength*. Wie der Parameter *keyStrength* interpretiert wird, ist in der Implementierung des Algorithmus festgelegt. Beim RSA Verfahren etwa gibt dieser an, wie groß der RSA-Modul ist. Bei alternativen Algorithmen kann der Wert allerdings zu hoch sein, etwa bei Algorithmen, die einen Baum mit zwei hoch *keyStrength* Blättern aufbauen.

**generateKeyPair(String algorithm, AlgorithmParameterSpec spec)** Alternativ kann der Generator auch mit einer Parametermenge initialisiert werden.

Nach dem Erzeugen werden die Schlüssel direkt in den Keystore gespeichert. Dabei werden diese mit dem Namen und dem Passwort – den der Benutzer über die Benutzerschnittstelle festlegt – versehen, um den Schlüsseleintrag eindeutig zu machen.

Die angezeigte Schlüsselgröße im Keystore für Schlüsselpaare berechnet sich folgendermaßen: für public keys und für private keys wird jeweils mit `getEncoded().length` die Größe des kodierten Schlüssels bestimmt. Wie der Schlüssel kodiert ist, wird durch die jeweilige Implementierung für das Verfahren im FlexiProvider festgelegt. Dieser Wert ist die Größe in Byte für den jeweiligen Schlüssel. Für das gesamte Schlüsselpaar werden die beiden Werte addiert. Dieser berechnete Wert sollte nicht verwechselt werden mit dem, der in der Benutzerschnittstelle beim Erstellen des Schlüssels eingegeben wurde, da der kodierte Schlüssel mehr Informationen hat, als nur den Parameterwert, der durch den übergebenen Wert festgelegt wurde. Ein Beispiel: Wird ein RSA Schlüssel mit dem Wert 1024 erstellt, so ist der public key anschliessend 162 Bytes (=1296 Bit) groß.

## 6.2 Java Keystore Schlüsselverwaltung

Die Schlüsselverwaltung stellt in einer *Public Key Infrastruktur* (PKI) eines der zentralen, nicht trivialen, Probleme dar. Ab JDK 1.2 wurde das Schlüsselverwaltungsparadigma in Java durch den Keystore gelöst. Es werden dadurch im Wesentlichen drei Probleme gelöst:

- man kann seine eigene Identität durch einen Privatekey beweisen (wie etwa durch das Signieren von Dokumenten);
- man kann fremde Identitäten verifizieren, durch deren Zertifikate;

- man kann secret keys für symmetrische Verfahren (in Flexible nur Ver- und Entschlüsselung) verwenden um Geheimnisse auszutauschen.

Im ersten Fall nutzt man den privaten Schlüssel, um Daten zu signieren. Das Zertifikat, das zum privaten Schlüssel passt, kann anschließend benutzt werden, um die Signatur zu verifizieren. Im zweiten Fall kann man das Zertifikat einer fremden Person importieren und damit die von dieser Person erstellten Signaturen auf ihre Gültigkeit prüfen. Im dritten Fall können symmetrische Schlüssel – etwa nach einem sicheren Schlüsselaustausch – verwendet werden, um sich gegenseitig verschlüsselte Nachrichten zu verschicken.

Den Java `KeyStore` kann man sich wie eine Art Adressbuch in einer Email Anwendung vorstellen. Hier können verschiedene Schlüssel und Zertifikate einem Alias (der aus einem String besteht) zugeordnet und abgespeichert werden. Konkret kann man im Keystore secret keys, private keys und Zertifikate speichern. Die Funktion des Keystore ist ähnlich zu der einer `java.util.Hashtable`. Dadurch gibt es die Möglichkeit, mehrere Arten von Schlüsseln, der gleichen Person zuzuordnen. Dies ist sehr sinnvoll, da man im Allgemeinen unterschiedliche Keypairs für unterschiedliche Anwendungen bereithalten will. Man möchte etwa aus Sicherheitsgründen ein anderes Schlüsselpaar für Signaturen verwenden als für die Authentifikation auf einem Webserver.

Ein Beispiel eines Keystore mit mehreren Einträgen sieht man in Abbildung 4. Hier sieht man, dass Bob ein Alias „Code Keys“ im Keystore hat, der seine Schlüssel zum Signieren von Code enthält. Mit dem privaten Schlüssel würde Bob seinen Code signieren. Das Zertifikat könnte er auf einen zentralen Server laden, von dem aus andere Benutzer seine Signaturen verifizieren könnten. Die selbe Funktion haben Bobs „Email Keys“. Mit diesen kann er seine Emails signieren oder entschlüsseln.

Desweiteren besitzt Bob ein Zertifikat von Alice. Mit diesem kann er signierte Emails von Alice verifizieren oder Emails an Alice verschlüsseln. Bob und Alice haben einen symmetrischen Schlüssel ausgetauscht. Dieser Schlüssel kann nun von Alice und Bob dazu verwendet werden, symmetrisch verschlüsselte Daten auszutauschen.

Von Carl hat Bob nur dessen Zertifikat. Dieses kann er verwenden um Carls Signaturen zu verifizieren oder Daten für Carl zu verschlüsseln. Mit Doris hat Bob einen symmetrischen Schlüssel ausgetauscht, den beide zum Verschlüsseln verwenden können.

Bob hat eine Sammlung von Schlüsseln, die er etwa zur Authentifikation auf einem Server verwenden kann. Außerdem hat er einen symmetrischen Schlüssel erzeugt, den er noch nicht ausgetauscht hat.

### 6.2.1 Schutz von Integrität und Vertraulichkeit

Der Java Keystore sieht eine Möglichkeit zum Schutz der Integrität vor. Dies ist wichtig, um den Keystore vor unerlaubten Veränderungen zu schützen und diese erkennen zu können. Die einfachste Lösung zum Schutz der Integrität ist es, beim Speichern den Hashwert des Keystore zu berechnen und diesen an das Ende des Keystore anzuhängen. Beim Laden des Keystore wird ein frischer Hashwert berechnet und mit dem angehängten verglichen. Dieser Ansatz ist jedoch nicht sicher gegen Angriffe. Ein Angreifer könnte die Schlüsseldaten des Keystore modifizieren und den Hashwert des modifizierten Keystore berechnen und anhängen.

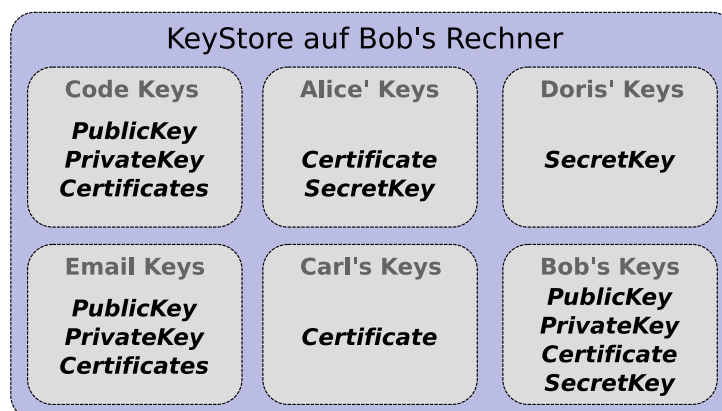


Abbildung 4: Mögliche Einträge in einem Java Keystore

Um diesen Angriff zu verhindern wird der Keystore passwortgeschützt. Dazu wird das Passwort zusammen mit dem Rest des Keystore gehasht und beim Speichern an den Keystore angehängt. Beim Laden des Keystore wird das Passwort erneut mit dem Keystore gehasht und mit dem abgespeicherten Hash verglichen. Falls der Hash nicht übereinstimmt, wurde der Keystore modifiziert oder das falsche Passwort wurde eingegeben.

Der Integritätsschutz des Keystore bietet jedoch nur den Schutz vor unerlaubtem Verändern. Damit sind noch nicht die einzelnen geheimen und privaten Schlüssel des Keystore geschützt, denn dieser wird im Klartext abgespeichert. Der Java Keystore bietet dazu die Möglichkeit jeden geheimen oder privaten Schlüsseleintrag auch mit einem Passwort zu schützen. Dabei bleibt die Implementierung des Verfahrens zum Schutz der *Vertraulichkeit* offen. Jeder Keystore Provider implementiert das Verfahren zum Schutz selbst.

Es gibt im Prinzip drei verschiedene Ansätze zum Sicherstellen der Vertraulichkeit von geheimen und privaten Schlüsseldaten [vgl. 12, S.62].

**Brain-dead-Schutz** Die einfachste Möglichkeit des „Schutzes“ der geheimen und privaten Schlüssel wäre, den Schlüssel mit einem Passwort zu speichern. Dieses Passwort wird einfach an den Schlüssel im Klartext angehängt. Wenn der Schlüssel aus dem Keystore wiederhergestellt werden soll, dann wird das übergebene Passwort mit dem gespeicherten Klartext Passwort verglichen. Stimmen diese überein, kann der Schlüssel geladen werden. Diese Methode bietet jedoch keinerlei Sicherheit, denn ein Angreifer könnte die Keystore Datei durchsuchen und so die Passwörter im Klartext finden. Somit würde er nicht nur die geheimen und privaten Schlüssel wiederherstellen können, sondern hätte alle vom Benutzer verwendeten Passwörter.

**Schwache Verschlüsselung** Eine andere Möglichkeit ist es, die geheimen und privaten Schlüssel zu verschlüsseln. Dazu nimmt man das Passwort und kombiniert dieses in irgendeiner Weise mit dem privaten Schlüssel, so dass man Passwort und Schlüssel nicht auf Anhieb finden kann. Dadurch hängt das Passwort nicht einfach nur im Klartext an den Schlüsseln dran. Diese Variante hält einer Kryptanalyse nicht stand, denn sie kann durch einen systematischen Angriff gebrochen werden, jedoch bietet sie eine sehr effektive und gegen einfachere Angriffe sichere Methode. Dieses Schema wird von dem Sun Provider, der standardmäßig mit der JCA ausgeliefert wird, umgesetzt. Der Keystore Provider von Sun heißt *JKS* und ist in der Security Klasse

initial angemeldet.

**Starke Verschlüsselung** Die sicherste Variante zum Schutz der Vertraulichkeit, ist die Passwort basierte Verschlüsselung. In diesem Verfahren wird das Passwort dazu benutzt, einen symmetrischen session key zu erzeugen. Dieser wird benutzt, um den Schlüssel – der im Keystore gespeichert werden soll – mit einem symmetrischen Verfahren zu verschlüsseln. Um den gespeicherten Schlüssel wiederherstellen zu können, wird das Schema rückwärts angewandt: aus dem übergebenen Passwort wird auf gleiche Weise ein Schlüssel berechnet und mit diesem versucht, den verschlüsselten Schlüssel zu entschlüsseln. War dies erfolgreich, so kann der Schlüssel wiederhergestellt werden. Diese Variante wird vom SunJCE Provider umgesetzt. Der Keystore Provider heißt *JCEKS* und ist standardmäßig in der Security Klasse angemeldet. Dieser Provider nutzt als symmetrisches Verschlüsselungsverfahren das *DESede* Verfahren. Flexible verwendet diesen Keystore Provider, wegen dem Bedarf an hohem Schutz des Keystore.

### 6.2.2 Nachteile des Keystore Ansatzes

Ein großer Nachteil des Keystore ist die Auflösung über den Alias. Es macht zwar die Verwaltung der Schlüssel, wie das Laden und Speichern, sehr einfach, jedoch müssen dadurch für zwei Privatekey Einträge zwei unterschiedliche Aliase gewählt werden. Flexible löst dieses Problem über eine Art Alias Verwaltung (der genaue Aufbau ist dem Frontend Teil der Arbeit zu entnehmen [8]). Dabei werden den gespeicherten Schlüsseln unter anderem Hashwerte zugeordnet, um sie später dem richtigen Alias zuzuordnen zu können.

Außerdem ist der Keystore *case-insensitiv*, das heißt alle Einträge werden in Kleinbuchstaben im Keystore gespeichert. Deswegen wird der erste Teil des Keystore Alias, der den Namen enthält, in seinen ASCII Wert umgewandelt und abgespeichert. Beim Laden aus dem Keystore wird dieser Alias zur Anzeige wieder in die lesbare Form zurück transformiert, wodurch man Klein- und Großschreibung unterscheiden kann.

Ein weiteres Problem ist, dass im Keystore private keys nur mit einer Zertifikatskette abgespeichert werden können. Dabei stellt der erste Eintrag der Kette das zu diesem private key passende Zertifikat dar. Jedoch hat man nach dem Erstellen eines Schlüsselpaars noch kein Zertifikat, dieses muss erst bei einer CA beantragt werden (siehe 8.3). Deswegen wird in Flexible zunächst ein sogenanntes *dummy* Zertifikat erstellt, welches außer dem Publickey keine sinnvollen Werte beinhaltet. Dieses Zertifikat dient nun für zwei Zwecke: einerseits zum Speichern des Publickey und zum anderen als Zertifikatskette (mit nur einem Eintrag) zum Speichern des Privatekey. Falls ein Zertifikat zu diesem Schlüssel erstellt wird – etwa durch das Bearbeiten des entsprechenden Zertifikatsantrages durch eine CA – wird das dummy Zertifikat mit dem Publickey aus dem Keystore gelöscht und durch das neue Zertifikat ersetzt.

## 7 Kryptographische Anwendungen für Schlüssel

Die beiden wichtigsten kryptographischen Anwendungen in Flexible, die man mit den im Keystore gespeicherten Schlüsseln ausführen kann, sind *Signieren* und *Verschlüsseln*. Es gibt außerdem in der Anwendung die Möglichkeit, direkt Hashwerte von Dateien, die geöffnet wurden, mit den im FlexiProvider implementierten Hashfunktionen zu berechnen und anzuzeigen. Auf den genauen Mechanismus zum Verschlüsseln, Signieren und berechnen von Hashwerten in der JCA/JCE gehe ich nur sehr oberflächlich ein, die Details können in [12] nachgelesen werden.

Wie bereits erwähnt, gibt es in den Klassen der JCA/JCE keinen Mechanismus, um abzufragen, ob ein Schlüssel für die Verschlüsselung, Signaturen oder gar beides geeignet ist. Dazu gibt es in Flexible in der Klasse `de.samd.functions.Functions` die beiden Methoden `boolean isSignatureAlg(String algname)` und `boolean isCipherAlg(String algname)` um festzustellen, welche Art vorliegt. Dabei werden in beiden Fällen alle Algorithmen des jeweiligen Typs im FlexiProvider ausgelesen und durchlaufen. Falls ein Algorithmus mit dem Namen `algname` vorkommt, so ist es ein Algorithmus des entsprechenden Typs und es wird `true` zurück gegeben.

Im Folgenden beschreibe ich kurz die Architektur der beiden Anwendungen – Signieren und Verschlüsseln – und die wichtigsten Funktionen der zugehörigen Klassen.

### 7.1 Verschlüsselung

Es gibt zwei Arten von Verfahren, die bei der Verschlüsselung unterschieden werden: die symmetrischen und die asymmetrischen Verfahren. In Abbildung 5 ist der Unterschied der beiden Varianten abgebildet.

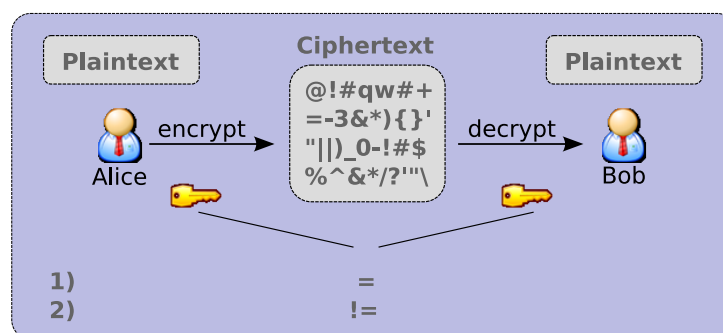


Abbildung 5: Symmetrische und asymmetrische Verschlüsselung

Bei der symmetrischen Verschlüsselung – der erste Fall in der Abbildung – ist der Verschlüsselungsschlüssel gleich dem Entschlüsselungsschlüssel. Falls dieses Verfahren benutzt werden soll, dann muss vorher der symmetrische Schlüssel ausgetauscht worden sein. Bei der asymmetrischen Verschlüsselung – der zweite Fall in der Abbildung – sind Verschlüsselungsschlüssel und Entschlüsselungsschlüssel ungleich. Falls Alice und Bob verschlüsselte Daten austauschen wollen, so müssen beide den jeweils öffentlichen Schlüssel des Anderen haben. Zum Beispiel könnten beide ihren öffentlichen Schlüssel von einer Zertifizierungsstelle (CA) unterschreiben lassen und ihr erhaltenes Zertifikat auf einem Public-Key-

Server veröffentlichen. Es gibt auch die Möglichkeit beide Verfahren zu einem hybriden Verschlüsselungsverfahren zu kombinieren (siehe dazu Abschnitt 7.1.3).

In Flexible liegen alle Schlüssel, wie bereits beschrieben, im Keystore. Die public keys sind dabei nur temporär gespeichert, diese können in der Anwendung nicht direkt benutzt werden. Die Anwendung von public keys geht nur über Zertifikate, aus denen der public key herausgeholt werden kann.

Der FlexiProvider unterstützt nur das Ver- und Entschlüsseln eines einzigen Blocks mit bestimmter Blockgröße. Diese Größe ist abhängig von dem Algorithmus und der Schlüsselgröße, die zum Verschlüsseln verwendet werden. Falls mehrere Blöcke, wie etwa ganze Dateien, verschlüsselt werden sollen, so muss dies der Client selber regeln. Flexible setzt dazu den *Electronic codebook* (ECB) Modus um. Dies ist der einfachste Modus. Dabei werden die Daten in Blöcke aufgeteilt und einzeln verschlüsselt. Anschließend werden die einzeln verschlüsselten Blöcke wieder zusammengesetzt und bilden den Schlüsseltext zum gesamten Klartext. Beim Entschlüsseln wird genauso verfahren. Die einzelnen Blöcke werden entschlüsselt und die einzelnen entschlüsselten Blöcke werden wieder zum ursprünglichen Klartext zusammengesetzt. Die Blockgröße für symmetrische Verfahren lässt sich über die JCE Schnittstellen abfragen, für die asymmetrischen Verfahren ist dies jedoch in den JCE Schnittstellen nicht vorgesehen. Die Blockgröße lässt sich nur über die FlexiProvider Klasse `de.flexiprovider.common.api.AsymmetricCipherSpi` herausfinden. Somit können ganze Dateien in Flexible sowohl symmetrisch als auch asymmetrisch verschlüsselt werden.

Die JCE Klasse für die Verschlüsselung ist `javax.crypto.Cipher`. Die Fassaden für die Verschlüsselung in der Anwendung liegen in dem Paket `de.samd.functions.crypting`. Jede Klasse, die Verschlüsselung durchführt, implementiert die entsprechende Schnittstelle für die Art des Verfahrens (`SymmetricCrypting` für symmetrische Verschlüsselung und `AsymmetricCrypting` für die asymmetrischen Verfahren) und erweitert die Klasse `de.samd.functions.Observable`. Hier wird das *Observer* [7] Pattern umgesetzt. Sobald in diesen Klassen eine Funktion, die ver- beziehungsweise entschlüsselt, aufgerufen wird, werden bei der Durchführung Funktionen aus der `Observable` Klasse aufgerufen. Als Observer dieser Klasse wird die GUI Klasse `de.samd.gui.VorgangGroup` angemeldet und über alle Änderungen informiert. Die Struktur dient unter anderem dazu, das Aktualisieren der Fortschrittsanzeige und Ausgeben von Meldungen, nach erfolgreichem Durchführen einer Aktion, an der richtigen Stelle zu ermöglichen.

Nach dem Verschlüsseln wird eine XML Session Datei erzeugt, die alle Einträge enthält, die zum Entschlüsseln der Datei benötigt werden. Diese Session Datei enthält die relativen Dateipfade zu den verschlüsselten Dateien und den Hash des benutzten Verschlüsselungsschlüssels. Außerdem gibt das Attribut *mode* an, welche Methode (0 für symmetrisch, 1 für asymmetrisch und 2 für hybrid) zum Verschlüsseln verwendet wurde. Wenn diese Datei in Flexible geöffnet wird, dann werden alle Daten automatisch gesetzt (falls der Schlüssel vorhanden ist) und es kann entschlüsselt werden, ohne etwa den passenden Schlüssel im Keystore suchen zu müssen.

Die JCE Klasse `Cipher` ist eine abstrakte Klasse, kann also nicht direkt instanziiert werden. Es gibt jedoch *Factory* Methoden, mit denen man sich Instanzen dieser Klasse holen kann. Um das `Cipher` Objekt benutzen zu können, müssen folgende Schritte ausgeführt werden:

1. Man holt sich ein Cipher Objekt mit `getInstance()`
2. Anschließend initialisiert man das Objekt für die Verschlüsselung mit der Methode `init()`. Diese Methode erwartet ein Argument für den Modus, für den das Cipher Objekt benutzt werden soll (`Cipher.ENCRYPT_MODE` zum Verschlüsseln und `Cipher.DECRYPT_MODE` zum Entschlüsseln) und den Schlüssel mit dem operiert werden soll. Außerdem kann man hier noch eine Parametermenge übergeben.
3. Nun kann mit der Methode `doFinal()` verschlüsselt oder entschlüsselt werden

### 7.1.1 Symmetrische Verfahren

Welche Schritte in der Benutzeroberfläche nötig sind, um symmetrisch zu verschlüsseln, kann dem Frontend Teil der Dokumentation entnommen werden. Hier gehe ich lediglich auf die involvierten Klassen und Methoden ein:

**SymmetricCrypting** ist die Schnittstelle, für das symmetrische Verschlüsseln und Entschlüsseln. Diese Schnittstelle stellt vier Methoden für die Ver- und Entschlüsselung zur Verfügung und wird von `SymmetricFileCrypt` implementiert.

Die vier Methoden sind:

- `encryptSymmetric(String clearTextFile, String cipherTextFile, SecretKey key, AlgorithmParameterSpec spec)`  
Verschlüsselt die Datei, die sich unter dem Pfad `clearTextFile` befindet und schreibt das Ergebnis in die `cipherTextFile` Datei. Dazu wird der symmetrische Schlüssel `key` zum Verschlüsseln verwendet. Die Verschlüsselung kann mit einer Parametermenge initialisiert werden, die in `spec` enthalten ist.
- `decryptSymmetric(String clearTextFile, String cipherTextFile, SecretKey key, AlgorithmParameterSpec spec)`  
Entschlüsselt die Datei, die sich unter dem Pfad `cipherTextFile` befindet und schreibt das Ergebnis in die `clearTextFile` Datei. Dazu wird der symmetrische Schlüssel `key` zum Entschlüsseln verwendet. Die Entschlüsselung kann ebenfalls mit einer Parametermenge initialisiert werden, die in `spec` enthalten ist.
- `encryptSymmetric(String clearTextFile, String cipherTextFile, SecretKey key)`  
Methode zum Verschlüsseln ohne eine Parametermenge.
- `decryptSymmetric(String clearTextFile, String cipherTextFile, SecretKey key)`  
Methode zum Entschlüsseln ohne eine Parametermenge.

**SymmetricFileCrypt** Diese Klasse implementiert die `SymmetricCrypting` Schnittstelle und implementiert die `Observable` Schnittstelle. Die dritte und vierte Methode von oben werden hier so implementiert, dass die entsprechende erste beziehungsweise zweite Methode mit `spec = null` aufgerufen wird.

Die einzelnen Schritte zum symmetrischen Ver- bzw. Entschlüsseln über die JCE Schnittstellen können hier [12, S.100-101] nachgelesen werden.

**PasswordEncryption** Die Passwortverschlüsselung dient in Flexible nur dazu, symmetrische Schlüssel aus dem Keystore zu exportieren, beziehungsweise um diese wieder zu importieren.

- *encryptSymmetricPwd(byte[] input, String cipherTextFile, String passphrase, boolean doDecrypt)*

Verschlüsselt das übergebene `byte` Array *input* und schreibt den Schlüsseltext an den Pfad, der in *cipherTextFile* angegeben wird. Dabei wird *passphrase* zum Verschlüsseln benutzt, außer *doDecrypt* ist `false`, dann wird der Klartext einfach durchgeschrieben.

- *byte[] decryptSymmetricPwd(String cipherTextFile, String passphrase, boolean doDecrypt)*

Diese Methode entschlüsselt die mit *passphrase* verschlüsselte Datei *cipherTextFile*, wenn *doDecrypt* `true` ist. Der Klartext wird anschließend in ein `byte` Array geschrieben und zurückgeliefert.

### 7.1.2 Asymmetrische Verfahren

Das Verschlüsseln mit einem asymmetrischen Verfahren kann in Flexible nur mit einem gültigen Zertifikat durchgeführt werden. Das Zertifikat muss dafür die *Key Usage* Erweiterung [9, 4.2.1.3] beinhalten und darin muss das *dataEncipherment* Bit auf `true` gesetzt sein, damit die Verschlüsselung in Flexible ausgeführt werden kann. Es kann allerdings mit jedem Privatekey entschlüsselt werden.

Vor dem Durchführen der Verschlüsselung wird weiterhin geprüft, ob der Algorithmus des verwendeten Schlüssels auch für die Verschlüsselung geeignet ist. Die beiden Klassen für die asymmetrische Verschlüsselung und deren Methoden sind:

**AsymmetricCryption** Diese Schnittstelle hat vier Methoden zum Ver- und Entschlüsseln von Dateien mit asymmetrischen Schlüsseln und Verfahren. Die vier Methoden sind:

- *encryptAsymmetric(String clearTextFile, String cipherTextFile, Key key, String algorithm)*

Verschlüsselt die unter *clearTextFile* angegebene Klartextdatei in *cipherTextFile* mit dem Schlüssel *key*. Der Algorithmus *algorithm* muss dabei explizit angegeben werden, da der gleiche Schlüssel im Allgemeinen mit mehreren Verfahren funktioniert. Mit einem RSA Keypair kann man etwa das Verfahren nach PKCS#1 v1.5 oder das Verfahren nach PKCS#1 v2.1 verwenden, die beide im FlexiProvider implementiert sind.

- *decryptAsymmetric(String clearTextFile, String cipherTextFile, Key key, String algorithm)*

Diese Methode entschlüsselt die Datei *cipherTextFile* unter Verwendung des Schlüssels *key* und des Algorithmus *algorithm* und schreibt den Klartext in die Datei *clearTextFile*.

- *encryptAsymmetric(String clearTextFile, String cipherTextFile, Key key, AlgorithmParameterSpec spec, String algorithm)*

Diese Methode verschlüsselt auf gleiche Weise wie die erste Methode, nur wird hier das `Cipher` Objekt mit der Parametermenge `spec` initialisiert.

- `decryptAsymmetric(String clearTextFile, String cipherTextFile, Key key, AlgorithmParameterSpec spec, String algorithm)`

Diese Methode entschlüsselt auf gleiche Weise wie die zweite Methode, nur wird hier das `Cipher` Objekt mit der Parametermenge `spec` initialisiert.

**AsymmetricFileCrypt** implementiert die obige Schnittstelle um die Funktionalität der Ver- und Entschlüsselung und erweitert `Observable`. Dabei ruft die erste (zweite) Methode die dritte (vierte) Methode auf, mit `spec = null`.

### 7.1.3 Hybride Verfahren

Bei der hybriden Verschlüsselung wird die symmetrische und die asymmetrische Variante kombiniert. Es wird zunächst ein symmetrischer `session key` erzeugt. Dabei kann der Benutzer das Verfahren und die Schlüsselgröße des symmetrischen Verfahrens über die Benutzerschnittstelle wählen. Mit diesem Schlüssel werden zuerst die Klartext Dateien verschlüsselt. Der session key wird anschließend mit dem vom Benutzer aus dem Keystore gewählten Zertifikat (genauer mit dem Publickey im Zertifikat) verschlüsselt und in einer separaten Datei gespeichert.

Die Schnittstelle `HybridCrypting` wird von `HybridFileCrypt` implementiert. Die Methoden sind kombiniert aus denen von symmetrischer und asymmetrischer Verschlüsselung. Um mit einem Zertifikat hybrid verschlüsseln zu können, muss in der `Key Usage` Erweiterung das `keyEncipherment` Bit gesetzt werden, da es dazu verwendet wird, den symmetrischen Schlüssel zu verschlüsseln.

Das Vorgehen zum Verschlüsseln ist dabei folgendes:

1. Es wird zunächst ein `session key` für das symmetrische Verfahren und die übergebene Schlüsselgröße erzeugt;
2. anschließend wird die übergebene Klartext Datei mit dem symmetrischen Schlüssel mit der Klasse `SymmetricFileCrypt` verschlüsselt;
3. danach wird der symmetrische Schlüssel in einen `ObjectOutputStream` geschrieben und mit dem angegebenen Publickey und dem übergebenen Algorithmus asymmetrisch verschlüsselt;
4. zuletzt wird der verschlüsselte symmetrische session key in eine Datei geschrieben.

Um hybrid verschlüsselte Daten wieder zu entschlüsseln werden folgende Schritte durchgeführt:

1. Es wird zunächst der `session key` unter der angegebenen Datei eingelesen;
2. dieser wird mit dem übergebenen privaten Schlüssel und asymmetrischen Algorithmus entschlüsselt und in ein `ObjectInputStream` geschrieben;
3. aus diesem Stream kann das Schlüsselobjekt wieder extrahiert werden;

4. mit dem dadurch erhaltenen symmetrischen Schlüssel wird nun die symmetrisch verschlüsselte Datei entschlüsselt.

## 7.2 Signaturverfahren

Für Signaturverfahren können nur Schlüsselpaare verwendet werden. Dabei wird der private Schlüssel zum Signieren und der öffentliche zum Verifizieren der Signaturen verwendet. Das Signieren wird in Flexible mit dem privaten Schlüssel im Keystore durchgeführt. Zum Verifizieren benötigt man jedoch das zugehörige Zertifikat, der Publickey reicht nicht aus (da dieser in einem dummy Zertifikat enthalten ist wie in 6.2.2 beschrieben). Falls man ein Zertifikat hat, welches zum privaten Schlüssel passt, mit dem signiert wurde und das *digitalSignature* Bit (siehe [9, 4.2.1.3]) in der *KeyUsage* Erweiterung gesetzt ist, so kann die Signatur korrekt verifiziert werden.

In Flexible können beliebige Dateien signiert werden. Nach dem Signieren liegen dann die Signaturen neben den signierten Dateien vor und haben die Endung ".sig". Es wird zusätzlich eine Session XML Datei erzeugt, die die Pfade aller signierten Dateien und den zugehörigen Signaturen enthält. Außerdem wird das verwendete Signaturverfahren und der Hash des verwendeten Schlüssels gespeichert. Nach dem Öffnen in der Anwendung liegen alle Daten bereit, so dass der Benutzer nichts mehr einstellen muss.

Die Klassen, die in Flexible am Erstellen und Verifizieren von digitalen Signaturen beteiligt sind, befinden sich in dem Paket `de.samd.functions.sign`. Das Konzept ist ähnlich zu dem der Verschlüsselung. Es wird wieder die `de.samd.functions.Observable` Klasse erweitert um Meldungen an die Benutzerschnittstelle weitergeben zu können.

Die JCA Schnittstelle für Signaturen heißt `java.security.Signature`. Um mit dieser Klasse Signaturen zu erstellen sind die folgenden Schritte auszuführen:

1. Es wird ein `Signature` Objekt über die `getInstance()` Factory Methode geholt. Hier muss der Algorithmus, der zum Signieren oder Verifizieren benutzt werden soll, angegeben werden. Der Algorithmus ist dabei zusammengesetzt aus zwei verschiedenen Algorithmen (zum Beispiel *SHA256withRSA*). Dabei gibt der erste Teil des Algorithmus (*SHA256*) an welche Hashfunktion benutzt werden soll um den Hashwert der zu signierenden Daten zu bilden. Der zweite Teil des Algorithmus (*RSA*) gibt an, mit welchem Verfahren der zuvor berechnete Hashwert signiert wird.
2. Das `Signature` Objekt wird durch die Methode `initSign()` mit dem Privatekey, mit dem signiert werden soll, initialisiert.
3. Die Daten, die signiert werden sollen, werden als `byte` Array mittels der Funktion `update()` hinzugefügt. Diese Methode kann beliebig oft hintereinander ausgeführt werden, so dass man etwa ganze Dateien signieren kann.
4. Mit der `sign()` Funktion wird dann letztendlich die Signatur der Daten, die vorher eingelesen wurden, berechnet. Die Methode liefert ein `byte` Array zurück, das die Signatur enthält.

Die Fassade Klasse, die digitale Signaturen von Dateien in Flexible erstellt und mit der diese verifiziert werden können, heißt `FileSign`. Die vier Methoden in dieser Klasse sind:

- *sign(InputStream is, OutputStream os, PrivateKey privKey, String digestAlg)*

Diese Methode signiert einen beliebigen *InputStream is* mit dem Publickey *pubKey* und schreibt die Signatur in den **OutputStream os**. Dazu wird als Signaturalgorithmus *digestAlg* benutzt. Diese Methode ist also für das Signieren von beliebigen Daten geeignet und wird in Flexible etwa auch für das Signieren von Zertifikatssperllisten, Zertifikatsanfragen und Zertifikaten benutzt.

- *boolean verify(InputStream sig, InputStream fileForVerify, PublicKey pubKey, String digestAlg)*

Mit dieser Methode können erstellte Signaturen verifiziert werden. Dabei werden sowohl die Signatur, als auch die zugehörige Nachricht als **InputStream** übergeben. Die Verifikation wird mit dem Publickey *pubKey* und dem Signaturalgorithmus *digestAlg* durchgeführt. Falls die Verifikation erfolgreich war, so wird **true** ansonsten **false** zurückgeliefert.

- *signFile(String fileToSign, String destFolder, PrivateKey privKey, String digestAlg)*

Diese Methode signiert die unter *fileToSign* befindliche Datei und schreibt die Signatur dieser Datei in den Ordner *destFolder*. Dabei ist die Signatur anschließend unter dem Dateinamen *destFolder/fileToSign.sig* zu finden. Es wird hier zum signieren die erste Methode aufgerufen.

- *boolean verifySignature(String fileForVerify, String signatureFile, String digestAlg, PublicKey pubKey)*

Diese Methode ist analog zu der zweiten Methode. Hier werden jedoch Dateinamen mit der Signatur und dem Klartext übergeben. Es wird die zweite Methode mit den entsprechenden Parametern aufgerufen und deren Rückgabewert zurückgeliefert.

## 8 Implementierung einer Public-Key-Infrastruktur

Eine *Public-Key-Infrastruktur* ist eine global verfügbare Infrastruktur, die kryptographische Sicherheit auf Basis von Public-Key-Kryptographie bereitstellt. Die PKI bildet die Basis, auf der andere Applikationen, Systeme und sicherheitskritische Netzwerke aufbauen. Sie ist dabei der Kern der gesamten Sicherheitsstrategie und muss daher mit anderen kryptographischen Mechanismen verbunden sein und mit diesen zusammenarbeiten können [vgl. 14, S.3].

Die Hauptfunktion einer PKI besteht darin, Schlüssel an Benutzer zu verteilen (zum Beispiel in Form von X.509 oder PKCS#12 Zertifikaten) und deren *Integrität* und *Authentizität* sicherzustellen. Flexible hat ein eigenes PKI Modul, deren Funktionen sich im *Admin* Bereich sammeln. Dieses Modul erlaubt es, Zertifikate auszustellen, die dann über den Keystore für die kryptographischen Anwendungen benutzt werden können. Im Folgenden sind die wichtigsten PKI Funktionen, die in der Anwendung umgesetzt sind, näher erläutert.

### 8.1 Certificate Authority (CA)

Die *Certificate Authority* ist die Institution, die digitale Zertifikate für andere Personen ausstellt. Sie stellt die Basis einer *Public Key Infrastruktur* dar. Die CA ist in der PKI die *trusted-third-party*. Das heißt, sie sorgt dafür, dass zwei Parteien, die sich gegenseitig nicht direkt vertrauen, das Vertrauen zueinander über diese dritte Partei herstellen können, der beide direkt vertrauen. Dazu können diese ein Zertifikat von der CA anfordern, welches sie nach der Ausstellung der jeweils anderen Partei zum Beweis der Identität bieten können.

In Flexible wird eine CA eindeutig durch das CA Zertifikat und den zugehörigen privaten Schlüssel identifiziert. Dabei ist dieses Zertifikat immer selbstsigniert und für jeden Keystore eindeutig. Das heißt, dass pro Keystore nur eine CA möglich ist. Es ist somit nicht möglich Zertifikatsketten aufzubauen. Das Wurzelzertifikat unterschreibt direkt alle durch die CA ausgestellten Benutzerzertifikate.

Folgende Funktionalität steht in Flexible für eine Certificate Authority zur Verfügung:

**Ausstellen von Zertifikaten** Die CA signiert das Zertifikat und authentifiziert somit die Identität des Antragstellers. Damit muss diese dafür Sorge tragen, dass der Antragsteller per *proof-of-possession* den Besitz des privaten Schlüssels nachweisen kann. Die CA muss vor dem Signieren des Zertifikats die Signatur der Zertifikatsanfrage auf Gültigkeit prüfen. Wenn diese korrekt ist, versieht die CA das Zertifikat, das sie ausstellt, mit einem Ablaufdatum. Desweiteren hat die CA die Möglichkeit sogenannte *Key Usages* zu vergeben. Dies ist eine X.509 Erweiterung, die als „critical“ markiert ist, die besagt, was der Antragsteller mit dem Schlüssel alles machen darf. Nach dem Bearbeiten eines Antrags kann die CA das Zertifikat veröffentlichen. In Flexible gibt es dazu zwei Möglichkeiten: das Speichern im Keystore des Antragstellers oder das Exportieren auf die Festplatte oder ein anderes Medium.

**Verlängern von Zertifikaten** Nach Ablauf des im Zertifikatsfeld *notAfter* [9, 4.1.2.5] angegebenen Zeitraums, kann das Zertifikat nicht mehr zum Prüfen von Signaturen

oder zum Verschlüsseln von Daten eingesetzt werden. Jedoch ist der Lebenszyklus des Schlüsselpaares meist länger als die Gültigkeit eines Zertifikates. Deswegen hat eine CA die Möglichkeit die Gültigkeit eines ausgestellten Zertifikates zu verlängern.

Dabei stellt die CA einfach ein neues Zertifikat für den selben Schlüssel aus dem alten Zertifikat aus. In Flexible werden bei der Verlängerung eines Zertifikates alle alten Zertifikatsfelder und Erweiterungen übernommen, bis auf den Zeitraum der Gültigkeit, der manuell festgelegt werden kann.

**Revozieren von Zertifikaten** Es gibt mehrere Gründe, warum die CA ein ausgestelltes Zertifikat wieder zurückziehen will. Der Benutzer kann seinen privaten Schlüssel verloren haben, seine Daten könnten sich geändert haben, wie etwa seine Anstellung. In solchen Fällen wird das Zertifikat unwiederbringlich für ungültig erklärt. Die CA kann anschließend eine Liste von ungültigen Zertifikaten veröffentlichen, damit Nutzer diese Information etwa beim Verifizieren von Signaturen nutzen können.

**Sicheres Verwalten und Bereitstellen von Schlüsseln und Zertifikaten** Wie bereits im Kapitel 6.2 beschrieben, wird diese Aufgabe mit Hilfe des Java Keystore gelöst. Schlüssel können erstellt und anschließend zertifiziert werden. Das erhaltene Zertifikat kann exportiert werden. Exportierte Zertifikate können in den Keystore wieder importiert werden. Die Vertraulichkeit aller Schlüssel, die sich im Keystore befinden ist durch ein vom Benutzer vergebenes Passwort geschützt und somit vor unbefugtem Zugriff sicher.

Das Erstellen von Schlüsseln ist ebenfalls eine sicherheitskritische Angelegenheit, da gewährleistet sein muss, dass die Algorithmen, die dazu verwendet werden, kryptographisch sicher sind. Dies wird durch die Implementierung des FlexiProviders gewährleistet.

## 8.2 X.509 Zertifikate

X.509 ist der derzeit wichtigste ITU-T Standard für digitale Zertifikate im Internet. Die Felder, die ein X.509 Zertifikat besitzen muss und die Erweiterungen, die es haben kann, sind im RFC 3280 [9] definiert. Hier ist auch die ASN.1 Notation für den gültigen Aufbau des Zertifikats angegeben. Im Allgemeinen sagt ein X.509 Zertifikat aus, dass der Name des Antragsstellers (*Subject*) gebunden wird an dessen Publickey (*Subject Public Key Info*). Die Authentizität und Integrität dieser Bindung wird durch eine CA (*Issuer*) etabliert.

Es sind eine Vielzahl von Erweiterungen in X.509 vorgesehen. Wird eine Erweiterung im Zertifikat gesetzt, so muss das X.509 Zertifikat die Versionsnummer v3 haben. In Flexible wird zur Zeit lediglich die *Key Usage* Erweiterung unterstützt.

Die Klasse `codec.x509.X509Certificate` aus `CoDec` dient dazu, die ASN.1 Struktur von X.509 Zertifikaten zu kodieren und wieder zu dekodieren.

In Flexible hat der Benutzer drei Formate, die ihm für Zertifikate zur Verfügung stehen: DER kodierte Zertifikate, Base64 kodierte Zertifikate und im Transport Format PKCS#12 kodierte Zertifikate mit Privatekey.

### 8.2.1 DER kodierte X.509 Zertifikate

Die *Distinguished Encoding Rules* sind eine Untermenge der *Basic Encoding Rules* (BER). Die Kodierung von ASN.1 Strukturen in DER ist eindeutig, das heißt, es gibt für jeden ASN.1 Wert im Gegensatz zu BER nur genau eine Kodierung. Dadurch eignet sich dieses Format vor allem dann, wenn ASN.1 Werte signiert werden sollen, da die Signaturen eindeutig sein müssen, was bei X.509 Zertifikaten der Fall ist oder wenn die Daten auf unterschiedliche Plattformen übertragbar sein sollen.

Das DER Format kann in Flexible beim Exportieren und Importieren von Zertifikaten verwendet werden. Zum DER kodierten Abspeichern wird mittels der Methode `getEncoded()` aus der CoDec Klasse `X509Certificate` der DER kodierte Inhalt des Zertifikats geholt. Dieser kann anschließend einfach in eine Datei geschrieben werden.

Beim Import können die aus der Datei ausgelesenen DER Daten als byte Array dem Konstruktor der Klasse `X509Certificate` übergeben werden und das Zertifikat wird korrekt in ein Objekt eingelesen.

### 8.2.2 Base64 kodierte X.509 Zertifikate

Um ein Zertifikat Base64 kodiert abzuspeichern wird zunächst das DER kodierte Zertifikat mittels der Methode `getEncoded()` geholt und anschließend mit Hilfe des Base64 Encoders aus CoDec `Base64.encode()` kodiert. Diese Daten werden anschließend umschlossen von "-----BEGIN CERTIFICATE-----" und "-----END CERTIFICATE-----" in eine Datei geschrieben.

Beim Import wird der Base64 Inhalt durch den Dekoder aus dem CoDec dekodiert und in das Zertifikat über den Konstruktor eingelesen.

### 8.2.3 Das PKCS#12 Transportformat für Zertifikate und Privatekey

Der PKCS#12 Standard dient zum Transport für private Schlüssel. Es gibt vier Modi zum Schutz der Integrität und Vertraulichkeit:

- Integritätsschutz durch ein Passwort, symmetrisch verschlüsselt mit einem Passwort
- Integritätsschutz durch ein Passwort, asymmetrisch verschlüsselt mit dem Publickey
- Integritätsschutz durch Signatur mit dem Privatekey, symmetrisch verschlüsselt mit einem Passwort
- Integritätsschutz durch Signatur mit dem Privatekey, asymmetrisch verschlüsselt mit dem Publickey

Am meisten verbreitet ist dabei die erste Variante, die auch von Flexible mit Hilfe von CoDec umgesetzt wird. Dazu wird der PFX Container von CoDec benutzt, an den der private Schlüssel, das zugehörige Zertifikat und das Passwort übergeben wird. Die Daten im Container werden anschließend mit dem `codec.asn1.DEREncoder` DER kodiert und in eine Datei geschrieben.

Der Weg zum Importieren eines PKCS#12 Zertifikats ist etwas umständlicher. Dazu wird der Java Keystore verwendet:

1. Zuerst wird mit `KeyStore.getInstance("pkcs12")` ein Keystore Objekt geholt, der das PKCS#12 Format unterstützt;
2. anschließend wird auf dieses Objekt die Methode `load` aufgerufen. Dieser werden als Argumente, die PKCS#12 Datei als `InputStream` und das zugehörige Passwort als String übergeben;
3. Aus dem Keystore können nun alle Schlüsseleinträge abgefragt werden (es gibt genau einen für den Privatekey) und die zugehörige Zertifikatskette ausgelesen werden, die das passende Zertifikat enthält.

### 8.3 Certificate Signing Request (CSR)

Der Standard für eine Zertifikatsanfrage ist im *PKCS#10* festgelegt. Das RFC 2986 [13] beschreibt, wie eine gültige Anfrage aufgebaut sein muss. Der CSR ist im Grunde ein selbstsigniertes CA Zertifikat, das von einer anderen Instanz (*trusted-third-party*) unterschrieben werden soll. Dabei ist definiert, dass eine gültige PKCS#10 Zertifikatsanfrage immer mit dem Privatekey signiert werden muss, der zum Publickey passt, der in der Anfrage selbst enthalten ist. Das bringt jedoch das Problem mit sich, dass keine Zertifikatsanfrage für ein Schlüsselpaar erstellt werden kann, mit dem man nicht signieren kann.

Da wir in Flexible auch Verschlüsselungsschlüssel benutzen wollen, ist das Problem auf folgende Weise gelöst: falls der Schlüssel signieren kann, wird der CSR mit dem zugehörigen privaten Schlüssel RFC 2986 konform signiert. Falls das Schlüsselpaar allerdings einen Verschlüsselungsschlüssel darstellt, mit dem nicht signiert werden kann, so bleiben die Felder für *signatureAlgorithm* und *signature* leer (sie werden mit `null` belegt). Solche Zertifikatsanfragen können in Flexible nicht exportiert werden. Da hier kein *proof-of-possession* sichergestellt werden kann, verlassen diese Anfragen die Anwendung nicht und können lediglich durch die interne CA Funktionalität bearbeitet werden. Nachdem solch ein CSR signiert wurde, wird das entsprechende Zertifikat in den Keystore gelegt und kann bei Bedarf vom Benutzer auch exportiert werden. CSRs die korrekt im Sinne von RFC 2986 erstellt sind, können auch aus dem Keystore exportiert und von einer externen Anwendung (wie OpenSSL) bearbeitet werden. Umgekehrt kann auch eine gültige Zertifikatsanfrage – also nur solche, die mit dem privaten Schlüssel signiert wurden – in Flexible importiert und durch die CA bearbeitet werden. Diese werden abgelehnt, falls die Anfrage eine ungültige Signatur enthält. Somit ist innerhalb von Flexible in jedem Fall *proof-of-possession* gewährleistet.

Eine Zertifikatsanfrage kann erst nach dem Erstellen eines Schlüsselpaares erstellt werden. Dabei wird zunächst wieder nur ein *dummy* Zertifikat für den CSR erstellt, um es im Keystore speichern zu können. Der Benutzer wählt hier lediglich die Felder für den *distinguished name*. Nach dem wechseln zur CA Funktionalität in der Benutzerschnittstelle, werden alle CSRs angezeigt, die sich im Keystore befinden. Es kann hier auch über die entsprechende Schaltfläche ein externes CSR importiert werden. Falls ein CA-Privatekey vorhanden ist, kann der CSR signiert werden. Dabei wird wie folgt verfahren:

1. Es wird ein „richtiges“ CSR mit Hilfe der entsprechenden Klasse aus dem Fraunhofer CoDec erstellt (`codec.pkcs10.CertificationRequest`);
2. es wird ein X.509 Zertifikat aus dem CSR erstellt, das mit dem Privatekey der CA signiert wird;
3. das so entstandene Zertifikat wird entweder in den Keystore zurückgeschrieben (das *dummy* CSR wird entfernt) oder falls es importiert wurde, wird das Zertifikat exportiert.

In Abbildung 6 ist das Vorgehen noch einmal bildlich dargestellt: Alice erstellt ein Schlüsselpaar, mit dem sie nur signieren kann, Bob hingegen eines, mit dem er nur verschlüsseln kann. Anschließend erstellen beide jeweils ein CSR für ihr Schlüsselpaar und geben die Werte für *SubjectDN* in der Eingabemaske ein. Dieser CSR wird dann im Keystore von Alice beziehungsweise von Bob gespeichert.

Nachdem nun in die *Admin* Oberfläche gewechselt wird, kann der CSR von der CA bearbeitet werden. Dazu wählt diese den entsprechenden CSR aus, welcher im aktiven Keystore liegt oder importiert einen externen CSR, was in der Abbildung der Einfachheit halber weggelassen wurde. Wenn die entsprechende Schaltfläche betätigt wird, wird nun ein Zertifikat erstellt, in welchem der Eintrag für *SubjectDN* aus dem CSR übernommen wird. Dieses wird anschließend mit dem privaten Schlüssel der CA aus dem aktuellen Keystore signiert. Nachdem das Zertifikat erzeugt wurde, wird es in den Keystore gespeichert. Die Zuordnung des Zertifikats zum entsprechenden Keypair Eintrag im Keystore wird dabei über den Hash des Publickey vorgenommen, Details dazu sind in [8] zu finden.

Falls ein Schlüsselpaar sowohl zum Signieren, als auch zum Verschlüsseln geeignet ist und der dazu erstellte CSR exportiert werden soll, dann wird der CSR beim Exportieren signiert, mit einem Algorithmus, den der Benutzer wählen darf. Es ist dann die Aufgabe der CA, die diesen CSR bearbeitet, zu entscheiden, für welchen Zweck der Benutzer das Zertifikat nutzen darf (etwa durch entsprechendes Setzen der *Key Usages*).

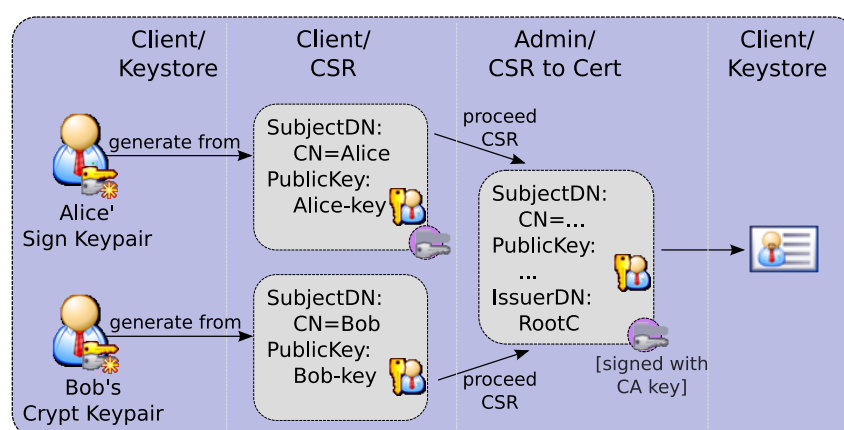


Abbildung 6: Bearbeiten einer CSR Anfrage

## 8.4 Certificate Revocation List (CRL)

Die *Certificate Revocation List* ist eine signierte Liste von ungültigen Zertifikaten. Die Struktur einer CRL wird in [9, Kapitel 5] definiert. Ein Eintrag in der CRL enthält die Seriennummer des Zertifikats, das revoziert wurde und das Datum, an dem es revoziert wurde. Es gibt hier – wie bei X.509 Zertifikaten – auch die Möglichkeit Erweiterungen anzugeben, wie etwa den Grund einer Revokation. Es ist wichtig zu beachten, dass die CRL keine abgelaufenen Zertifikate enthält, sondern nur von der CA, die die CRL ausstellt, für ungültig erklärte Zertifikate.

In Flexible ist die CRL immer mit dem privaten Schlüssel der CA signiert, der auch für das Signieren der einzelnen ausgestellten Zertifikate verwendet wurde. Somit ist eine CRL nur eindeutig für eine bestimmte CA, dessen *DN* im Feld *Issuer* angegeben wird. Die CRL enthält außerdem die Felder *thisUpdate* und *nextUpdate*, die angeben, wann die CRL ausgestellt, beziehungsweise wann die nächste CRL veröffentlicht wird. Eine CRL enthält also alle von der CA revozierten Zertifikate, die im Zeitraum zwischen der Ausstellung des CA Zertifikates und dem Datum in *thisUpdate* revoziert wurden. Abgelaufene Zertifikate werden in Flexible nicht aus der CRL entfernt. Die CRL sieht auch X.509 Erweiterungen wie eine *crlNumber* vor, die allerdings in Flexible zur Zeit nicht beachtet werden.

Sobald die CA in Flexible das erste Zertifikat ausstellt, wird eine Index Datei erstellt, in der alle ausgestellten Zertifikate mit Datum, Seriennummer, SubjectDN und Status (V für gültig, E für abgelaufen und R für revoziert) aufgehoben werden. Diese Liste wird im CA Bereich angezeigt. Falls ein gültiges, nicht abgelaufenes Zertifikat revoziert werden soll, so wird der entsprechende Eintrag in der Liste ausgewählt und über das Kontextmenü revoziert. Anschließend kann die CRL exportiert werden. Dabei wird aus der Index Datei automatisch eine CRL generiert und mit dem in Flexible eindeutigen CA Privatekey des aktiven Keystore signiert.

## 9 Ausblick

Flexible bietet eine flexible Benutzerschnittstelle zum Anwenden der im FlexiProvider umgesetzten kryptographischen Verfahren für die Verschlüsselung und die Berechnung von digitalen Signaturen von Dateien. Dies war das Hauptziel der Arbeit. Desweiteren ist Dank des Java Keystore die Schlüsselverwaltung in der Anwendung relativ übersichtlich und einfach gehalten, so dass die Schlüssel in einer Art Adressbuch angeordnet sind. Schlüssel können erzeugt, zertifiziert und das Zertifikat anschließend exportiert werden. Diese exportierten Zertifikate, Zertifikatsanfragen und Zertifikatssperrlisten können mit anderen Programmen (getestet wurde mit OpenSSL und Firefox) weiterverarbeitet werden. Genauso können Zertifikate und Zertifikatsanfragen, die von anderen Programmen erstellt wurden, in Flexible importiert und benutzt werden.

Es gibt in der Anwendung auch erste Ansätze der Umsetzung einer *Public Key Infrastruktur*. Über die *Admin* Oberfläche wird eine CA simuliert. Dieser CA wird ein eindeutiges Schlüsselpaar und ein selbstsigniertes Wurzelzertifikat im Keystore zugeordnet. Die CA hat einen Überblick über alle von ihr ausgestellten Zertifikate und kann einzelne Zertifikate sperren. Diese Sperrinformationen können anschließend aus Flexible exportiert und zum Beispiel auf einem zentralen Server platziert werden, über den die Sperrinformationen für alle Benutzer verfügbar sind. Außerdem kann die CA abgelaufene Zertifikate verlängern, im Falle, dass der öffentliche Schlüssel aus dem Zertifikat weiterbenutzt werden soll.

Es konnte wegen der dreimonatigen Laufzeit der Arbeit nicht alles in vollem Umfang umgesetzt werden, was in einer solchen Anwendung sinnvoll gewesen wäre. Viele interessante und sinnvolle Funktionen, wie die Validierung von Zertifikaten oder das Erstellen von Zertifikatsketten sind deshalb nur teilweise oder garnicht umgesetzt. Einige wichtige CA Funktionen, die in der Zukunft sinnvoll wären, sind:

- Erstellen von Zertifikatsketten

Bisher ist es lediglich möglich ein einziges CA Zertifikat pro Keystore zu haben. Dieses ist zwangsläufig in der Anwendung selbst erstellt worden und kann nicht importiert werden. Es sollte möglich sein, mit diesem Zertifikat weitere CA Zertifikate erstellen zu können, die dann entsprechend im Keystore markiert werden und in der Anwendung als solche benutzt werden. Desweiteren sollten CA Zertifikate, die mit anderen Anwendungen erstellt wurden auch in Flexible als CA Zertifikate importiert werden können.

- Validierung von Zertifikatsketten

Da es in Flexible zur Zeit nicht möglich ist selbst Zertifikatsketten im Keystore aufzubauen, stellt sich das Problem der Validierung eines Zertifizierungspfades nicht. Würde jedoch das vorher angesprochene Feature umgesetzt, so wäre es sinnvoll, die Zertifizierungspfade von importierten Zertifikaten – denn in der Anwendung selbst erstellte sind immer gültig – auf Gültigkeit zu prüfen. Dazu gibt es im Grunde drei verschiedene Modelle, nach denen der Pfad validiert werden kann:

- das Schalenmodell. Hier wird zum Zeitpunkt der Validierung eines Zertifikats geguckt, ob alle im Pfad darüberliegenden Zertifikate gültig sind [vgl. 9, Kapitel 6, S.62ff].

- das Hybridmodell. Ist im Grunde ein erweitertes Schalenmodell. Hier wird beim Validieren eines Zertifikats geschaut, ob zum Zeitpunkt der Signaturerstellung des Zertifikats alle darüberliegenden Zertifikate gültig sind. Man kann den Prüfalgorithmus für das Schalenmodell im RFC 3280 so anpassen, dass man statt dem Validierungszeitpunkt den der Signaturerstellung angibt.
- das Kettenmodell. Ist ein deutsches Modell und zur Zeit nicht standardisiert. Es verhält sich anders als die beiden vorhergehenden: Es wird zum Zeitpunkt der Signatur geschaut, ob das darüberliegende Zertifikat gültig ist. Für das darüberliegende wird wiederum geschaut, ob zur Zeit dessen Signaturerstellung das darüberliegende gültig ist, etc.

Alle drei Modelle haben Vor- und Nachteile, die man abwägen müsste. Es wäre sinnvoll die Pfadvalidierung so umzusetzen, dass der Benutzer im Keystore angezeigt bekommt, ob das Zertifikat valide ist, nicht validiert werden konnte (etwa wenn nicht jedes Zertifikat in der Kette verfügbar ist) oder ungültig ist, aufgrund des Scheiterns der Prüfung.

Außerhalb der PKI Funktionalität gibt es auch einige Stellen, an denen Erweiterungen wichtig sind. Die beiden wichtigsten betreffen die Anwendung von kryptographischen Funktionen und Schlüsseln:

- Fertigstellen der Implementierung einer Algorithmenstruktur

In Kapitel 5.4 wurde beschrieben, wie die Algorithmenstruktur in Flexible implementiert ist. Hier gab es jedoch noch das verbliebene Problem der Parametrisierung. Dieses konnte im zeitlichen Rahmen nicht beseitigt werden, da zu viele Änderungen im FlexiProvider notwendig wären. Für die Anwendung selbst wäre eine solche Parametrisierung jedoch wichtig, da der versierte Anwender mehr Flexibilität bei der Erstellung und Verwendung von Schlüsseln innerhalb der Anwendung hätte.

- PKCS#7 Standard für verschlüsselte oder signierte Daten

Der *PKCS#7* spezifiziert ein Format für verschlüsselte und/oder signierte Nachrichten oder Dateien auf der Grundlage von S/MIME, er ist im RFC 2315 beschrieben. Die gesammelten Daten (verschlüsselter Hashwert, Dokument, Zertifikat inklusive öffentlicher Schlüssel) werden im PKCS#7 Format an den Empfänger übermittelt oder auf einem Datenträger gespeichert. Durch diesen Standard ist es möglich, die von Flexible erzeugten Dateien auch in anderen Anwendungen, die diesen Standard umsetzen, zu benutzen. Der Standard löst auch das Problem, dass man nach dem Ausführen einer Funktion auf der Senderseite nicht mehr weiß, zum Beispiel welcher Signaturalgorithmus zum Signieren verwendet wurde. Außerdem ist in PKCS#7 der öffentliche Schlüssel enthalten, so dass der Empfänger einer verschlüsselten Datei seinen privaten Schlüssel etwa anhand des Keyhashes finden kann. Die Probleme, die dieser Standard löst, sind in Flexible durch die Session XML Dateien gelöst, jedoch ist dies kein Standard und funktioniert nicht in anderen Anwendungen.

Außerdem sind in der Anwendung selbst, einige Sicherheitsaspekte nicht umgesetzt, die wichtig wären. Es wird etwa beim Verifizieren einer Signatur nicht geprüft, ob das Zertifikat, mit dem verifiziert wird, überhaupt gültig ist. Dieses Problem würde durch das Umsetzen eines Prüfalgorithmus für Zertifikatspfade mit gelöst werden.

Viele Zertifikatserweiterungen, wie *Policies* und *Basic Constraints*, die gerade für eine CA sehr wichtig sind, werden bisher in der Anwendung nicht unterstützt.

Ein weiterer fehlender Sicherheitsaspekt ist das *critical* Flag bei Zertifikatserweiterungen. Ist dieses gesetzt und wird die Erweiterung von der Anwendung, die das Zertifikat verarbeitet nicht verstanden, so muss das Zertifikat abgelehnt werden. In Flexible werden zur Zeit alle Erweiterungen, außer der *Key Usage* Erweiterung, nicht beachtet, auch wenn das *critical* Flag gesetzt wurde.

## Literatur

- [1] Public Key Cryptography Standards. RSA Laboratorien, Verfügbar unter, <http://www.rsa.com/rsalabs/node.asp?id=2124>, 1991.
- [2] FlexiProvider. Fachgebiet für Kryptographie und Computeralgebra, TU Darmstadt, Verfügbar unter, <http://www.flexiprovider.de>, 2001.
- [3] FlexiTRUST. Fachgebiet für Kryptographie und Computeralgebra, TU Darmstadt, Verfügbar unter, <http://www.flexsecure.de/ojava/flexitrust.html>, 2001.
- [4] SeMoA. Fraunhofer Institut Graphische Datenverarbeitung, Verfügbar unter, <http://www.semoa.org>, 2002.
- [5] XStream. J. Walnes, J. Schaible, Verfügbar unter, <http://xstream.codehaus.org/changes.html>, 2004.
- [6] QDox. J. Walnes, Verfügbar unter, <http://qdox.codehaus.org/index.html>, 2005.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- [8] M. Ghiglieri. Flexible - Eine erweiterbare GUI für den FlexiProvider - Frontend. Bachelorarbeit, TU Darmstadt, Fachbereich Informatik, Fachgebiet CDC, Darmstadt, Germany, 2007.
- [9] R. Housley, W. Polk, W. Ford, and D. Solo. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile, RFC Editor, 2002.
- [10] D. B. Johnson and A. J. Menezes. Elliptic curve dsa (ecsda): an enhanced dsa. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998*, pages 13–13, Berkeley, CA, USA, 1998. USENIX Association.
- [11] J. Jonsson and B. Kaliski. Public-key cryptography standards (pkcs) #1: Rsa cryptography version 2.1, RFC Editor, 2003.
- [12] J. Knudsen. *Java cryptography*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.
- [13] M. Nystrom and B. Kaliski. Certification request syntax specification version 1.7, RFC Editor, 2000.
- [14] B. J. Weise. Public Key Infrastructure Overview. Sun blueprints, Sun Microsystems, Inc., <http://www.sun.com/blueprints/0801/publickey.pdf>, 2001.