

Javaseminar WS 98/99  
Java und die Sicherheit

Patrick Kaell  
Matr. 927.461

Januar 1999

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>2</b>
<b>2</b>	<b>Das Java-Sicherheits API</b>	<b>2</b>
2.1	Einführung . . . . .	2
2.2	API-Konzepte . . . . .	2
2.2.1	<i>Engine</i> Klassen und Algorithmen . . . . .	2
2.2.2	Implementierungen und <i>providers</i> . . . . .	3
2.2.3	<i>Factory</i> Methoden . . . . .	3
2.3	Kernklassen und Schnittstellen . . . . .	3
2.3.1	Die <i>Provider</i> Klasse . . . . .	3
2.3.2	Die <i>Security</i> Klasse . . . . .	4
2.3.3	Die <i>MessageDigest</i> Klasse . . . . .	5
2.3.4	Die <i>Signature</i> Klasse . . . . .	6
2.3.5	Die <i>Key</i> Schnittstelle . . . . .	8
2.3.6	Die <i>PublicKey</i> und <i>PrivateKey</i> Schnittstellen . . . . .	8
2.3.7	Die <i>KeyPair</i> Klasse . . . . .	8
2.3.8	Die <i>KeyPairGenerator</i> Klasse . . . . .	9
2.3.9	Die <i>SecureRandom</i> Klasse . . . . .	10
2.4	Ein kleines Beispiel . . . . .	10
<b>3</b>	<b>Java Sicherheits-Werkzeuge</b>	<b>11</b>
3.1	Das JAR Archiv-Format . . . . .	11
3.1.1	Einführung . . . . .	11
3.1.2	Das <i>APPLET tag</i> . . . . .	11
3.1.3	Das JAR-Werkzeug . . . . .	12
3.2	Das <i>jarsigner</i> -Werkzeug . . . . .	12
3.3	Das <i>keytool</i> -Werkzeug . . . . .	13
<b>4</b>	<b>Die Java-<i>Sandbox</i></b>	<b>13</b>
4.1	Einführung . . . . .	13
4.2	Die <i>Sandbox</i> . . . . .	13
<b>5</b>	<b>Verschiedene kryptographische Algorithmen</b>	<b>14</b>
5.1	Einführung . . . . .	14
5.2	Symmetrische Verschlüsselungsverfahren . . . . .	14
5.2.1	Einführung . . . . .	14
5.2.2	DES . . . . .	14
5.2.3	IDEA . . . . .	16
5.3	Asymmetrische Verschlüsselungsverfahren . . . . .	16
5.3.1	Einführung . . . . .	16
5.3.2	RSA . . . . .	17
5.4	<i>Message Digests</i> . . . . .	17
5.5	Digitale Unterschriften . . . . .	17

# 1 Einführung

Diese Ausarbeitung befaßt sich mit den verschiedenen Sicherheitsaspekten der Java-Programmiersprache und Java-Umgebung. Zuerst wird das Java-Sicherheits API vorgestellt. Diese Vorstellung ist unvollständig und enthält zum Beispiel nicht die Klassen, die benutzt werden um *keystore*-Datenbanken zu verwalten. Im nächsten Kapitel werden die Java Sicherheits-Werkzeuge vorgestellt, die hauptsächlich Kommandozeilenschnittstellen zu der Java-Sicherheits API darstellen. Anschließend geht diese Ausarbeitung auf die *Java-Sandbox* ein und stellt verschiedene kryptographische Algorithmen vor.

## 2 Das Java-Sicherheits API

### 2.1 Einführung

Dieses Kapitel stellt das Java-Sicherheits API (**A**pplication **P**rogramming **I**nterface, *Programmierschnittstelle*) vor. Es erlaubt dem Java-Programmierer auf relativ einfache Weise, seine Programme mit kryptographischen Fähigkeiten auszustatten.

Das JCA (**J**ava **C**ryptography **A**rchitecture) bietet ein Grundgerüst um die Java-Plattform mit verschiedenen kryptographischen Algorithmen und Implementierungen auszustatten, bzw. zu erweitern. Dabei bietet das API dem Anwendungsprogrammierer eine einheitliche Schnittstelle zu verschiedenen kryptographischen Funktionalitäten, unabhängig davon welcher konkreter Algorithmus im Hintergrund arbeitet.

Das JCA wurde nach folgenden Kriterien entworfen:

- Implementierungsunabhängigkeit und Austauschbarkeit
- Algorithmusunabhängigkeit und Erweiterbarkeit

**Implementierungsunabhängigkeit** wird mit Hilfe der *Cryptography Package Provider*-Architektur realisiert. Ein *provider* ist ein Paket (oder eine Menge von Paketen), das Implementierungen von bestimmten Algorithmen enthält. Dabei kann eine Anwendung ein bestimmter Algorithmus verlangen, und eine Implementierung irgendeines installierten *providers* bekommen, oder nach einer Implementierung eines bestimmten *providers* nachfragen.

**Austauschbarkeit** bedeutet, daß verschiedene Implementierungen eines gleichen Algorithmus *kompatibel* sind, d.h.: Verschiedene Implementierungen arbeiten zusammen, z.B.: benutzen die gleichen Schlüssel oder lesen die gleichen Unterschriften.

**Algorithmusunabhängigkeit** bedeutet, daß eine Anwendung eine bestimmte kryptographische Funktionalität verlangen kann, unabhängig vom verwendeten Algorithmus.

**Algorithmuserweiterbarkeit** bedeutet, daß man neue Algorithmen, die im Java-API schon vorgesehene Funktionalitäten implementieren, einfach hinzufügen kann.

### 2.2 API-Konzepte

#### 2.2.1 *Engine* Klassen und Algorithmen

*Engine* Klassen sind Klassen die die Funktionalität eines bestimmten Typs von kryptographischen Algorithmen zur Verfügung stellen. Beispiele solcher Klassen sind: *MessageDigest*, *Signature* und *KeyPairGenerator*.

*Engine* Klassen bieten dem Anwendungsprogrammierer eine Schnittstelle zu einem bestimmten Typ von Algorithmen, wobei die Implementierung vom gerade gewählten *provider* abhängt. Dabei generiert der Programmierer einfach ein Objekt (Instanz) einer bestimmten *engine* Klasse, um eine bestimmte Operation auszuführen. Zum Beispiel kann man mit Hilfe eines *Signature* Objektes digitale Unterschriften generieren und überprüfen, ohne sich um den verwendeten Algorithmus oder Implementierung zu kümmern.

Zum Beispiel:

**Signature** Mögliche Algorithmen: SHA-1 mit DSA, SHA-1 mit RSA oder MD5 mit RSA.

**MessageDigest** Mögliche Algorithmen: SHA-1, MD5 oder MD2

Um reine Verschlüsselungsalgorithmen (z.B.: DES, RSA) zu implementieren oder zu benutzen, braucht man die Java API Erweiterung JCE (**J**ava **C**ryptography **E**xtension), die momentan nicht ausserhalb den Vereinigten Staaten exportiert werden darf, da starke Verschlüsselungstechnik in den USA unter das Waffenexportverbot fällt.

### 2.2.2 Implementierungen und *providers*

Die Implementierungen der *engine* Klassen werden von den sogenannten *providers* zur Verfügung gestellt. Dabei können diese Pakete mehrere *engine* Klassen bedienen.

Das JDK 1.1 wird standardmäßig mit dem *provider* „SUN“ ausgeliefert, das folgende Implementierungen enthält:

- Eine Implementierung des *Digital Signature Algorithm* (NIST FIPS 186).
- Eine Implementierung der *Message Digest* Algorithmen MD5 (RFC 1321) und SHA-1 (NIST FIPS 180-1).

Jede JDK-Installation enthält ein oder mehrere *providers*. Neue *providers* können statisch oder dynamisch hinzugefügt werden.

Eine Anwendung kann auf mehrere *providers* zurückgreifen, wobei man eine Prioritätsliste angeben kann, die bestimmt nach welcher Reihenfolge die *provider* nach einem bestimmten Algorithmus durchsucht werden sollen.

### 2.2.3 *Factory* Methoden

Ein Objekt einer bestimmten *engine* Klasse wird generiert, indem man die *factory* Methode dieser *engine* Klasse aufruft. Eine *factory* Methode ist eine statische Methode die das Objekt (Instanz) einer Klasse zurückgibt.

So kann der Anwendungsprogrammierer zum Beispiel ein *Signature* Objekt erzeugen, indem er die `getInstance` Methode der *Signature* Klasse aufruft und als Parameter den Namen des *Signature* Algorithmus (z.B.: „DSA“) und optional den Namen des gewünschten *providers* übergibt. Die `getInstance` Methode sucht nach einer *Signature* Unterklasse, die den übergebenen Parametern genügt. Wenn kein *provider* angegeben wurde, geht die Methode die Prioritätsliste durch, bis eine Unterklasse gefunden wurde, die den angegebenen Algorithmus implementiert.

## 2.3 Kernklassen und Schnittstellen

### 2.3.1 Die *Provider* Klasse

Eine *provider* Klasse ist die Schnittstelle zu dem dazugehörigen *provider*. Eine solche Klasse enthält Methoden um den Namen des *providers*, die Versionsnummer oder eine andere Information zu erfragen.

Um eine Implementierung eines kryptographischen Algorithmus zur Verfügung zu stellen, generiert der Entwickler eine Unterklasse der *provider* Klasse, die den Code der Implementierung enthält. Der Konstruktor der Unterklasse überliefert dem Java Sicherheits API Informationen über die Implementierung (z.B.: Namen der Klassen die den Algorithmus implementieren), so daß das API nach den Algorithmen suchen kann.

Um einen *provider* zu installieren, muß man zuerst die *provider* Paket Klassen installieren, und dann den *provider* konfigurieren.

Um die Paket Klassen zu installieren, kopiert man einfach die `.class` Dateien in das `classes` Unterverzeichnis der JDK-Installation. Alternativ kann man, wenn die Dateien in einem zip oder JAR (**J**ava **A**Rchive) Archiv gepackt sind, diese in irgendein Verzeichnis kopieren wohin die Variable `CLASSPATH` zeigt.

Um den *provider* zu konfigurieren, muß man zuerst den *provider* in die *provider*-Liste eintragen. Dazu muß man den *provider* statisch in die Datei `jdk1.1.1/lib/security/java.security` (wobei `jdk1.1.1` das JDK Installationsverzeichnis ist) eintragen. Folgender Eintrag ist möglich:

```
security.provider.n=masterClassName
```

*n* gibt die Priorität des *providers* an, also nach welcher Reihenfolge die *providers* nach einem passenden Algorithmus durchsucht werden sollen. *masterClassName* gibt den Namen der *master* Klasse des *providers* an.

Jedes Objekt einer *provider* Klasse besitzt Methoden um den Namen des *providers*, die Versionsnummer, und die Zeichenkette die den *provider* und seine Funktionen beschreibt zu erfragen.

Zum Beispiel:

```
public String getName()
public double getVersion()
public String getInfo()
```

### 2.3.2 Die *Security* Klasse

Die *Security* Klasse verwaltet installierte *provider* und sicherheitsrelevante Einstellungen. Die Klasse enthält nur statische Methoden und kann nie ein Objekt erzeugen.

Diese Methoden können nur aufgerufen werden wenn das Java Programm eine lokale Applikation ist, oder über die Rechte einer lokalen Applikation verfügt.

Ein *Applet* verfügt über die gleichen Rechte wie eine lokale Applikation, wenn:

- es sich in einer JAR Datei befindet, die mit dem *javakey* (*jarsigner* in JDK 2) Werkzeug signiert worden ist, und
- die von *javakey* verwaltete Datenbank enthält eine Kopie des Zertifikats des öffentlichen Schlüssels der Organisation, die die JAR Datei signiert hat

Die *Security* Klasse wird benutzt, um nachzufragen welche *providers* schon installiert sind, oder um neue zu installieren.

#### **providers suchen**

Folgende Methode gibt ein Feld mit den installierten *providers* zurück. Das Feld ist nach den Prioritäten der *provider* sortiert.

```
public Provider [] getProviders()
```

Folgende Methode gibt den *provider* zurück, dessen Name mit der Variable `providerName` übereinstimmt.

```
public Provider getProvider(String providerName)
```

Wenn kein *provider* gefunden wurde, gibt diese Methode den Wert `null` zurück.

#### ***providers* hinzufügen**

Folgende Methode fügt einen *provider* in die nächste verfügbare Prioritätsposition ein. Wenn der *provider* hinzugefügt wurde, gibt die Methode die Prioritätsposition zurück, anderenfalls wenn der *provider* nicht hinzugefügt wurde, weil der *provider* schon installiert war, gibt die Methode den Wert `-1` zurück.

```
public void addProvider(Provider provider)
```

Folgende Methode fügt einen neuen *provider* an einer bestimmten Position ein. Position 1 hat die größte Priorität, gefolgt von 2 usw. Das Java API garantiert allerdings nicht, daß dieser *provider* tatsächlich an der angegebenen Position installiert wird.

```
public int insertProviderAt(Provider provider, int  
    position)
```

Die Methode gibt die Prioritätsposition zurück, an der der *provider* tatsächlich installiert wurde, oder den Wert `-1`, wenn der *provider* nicht installiert wurde.

#### ***providers* entfernen**

Folgende Methode entfernt den *provider* mit dem angegebenen Namen. Die Methode gibt kein Wert zurück, auch wenn kein *provider* installiert war.

```
public void removeProvider(String name)
```

#### **Sicherheitseinstellungen**

Die *Security* Klasse verwaltet eine systemweite Liste, die sicherheitsrelevante Einstellungen enthält. Programme, die die nötigen Rechte besitzen, (lokale Applikation, oder signiertes Applet) können diese Einstellungen mit den folgenden Methoden auslesen und verändern.

```
public static String getProperty(String key)  
public static void setProperty(String key, String datum)
```

#### **2.3.3 Die *MessageDigest* Klasse**

Die *MessageDigest* Klasse ist eine *engine* Klasse, die die Funktionalität von kryptographisch sicheren *message digest* (Fingerabdruck) Algorithmen (wie SHA-1 oder MD5) zur Verfügung stellt. So ein Algorithmus bekommt eine längenvariable Eingabe (ein Feld von Bytes) und generiert daraus eine längenkonstante Ausgabe, den *digest*, mit folgenden Eigenschaften:

- Es soll rechnerisch unmöglich sein, eine andere Eingabe zu finden, die die gleiche Ausgabe liefert.
- Die Ausgabe soll keine Informationen enthalten, die es ermöglichen würde, etwas über die Eingabe herauszufinden.

*Message digests* werden benutzt, um einmalige und zuverlässige Identifizierer digitaler Daten zu erzeugen. Sie werden manchmal auch „digitale Fingerabdrücke“ von Daten genannt.

### Ein *MessageDigest* Objekt erzeugen

Um ein *MessageDigest* Objekt zu erzeugen, ruft man die statische *factory* Methode der *MessageDigest* Klasse auf:

```
public static MessageDigest getInstance(String algorithm)
```

Optional kann man auch den Namen des gewünschten *providers* angeben:

```
public static MessageDigest getInstance(String algorithm,  
String provider)
```

Die `getInstance` Methode initialisiert das Objekt, es braucht also nicht mehr weiter initialisiert werden.

### Das *MessageDigest* Objekt mit einer Eingabe versorgen

Im nächsten Schritt wird das initialisierte *MessageDigest* Objekt mit Hilfe einer oder mehreren Aufrufen folgender Methoden mit einer Eingabe versorgt.

```
public void update(byte input)  
public void update(byte[] input)  
public void update(byte[] input, int offset, int len)
```

### Den *digest* ausrechnen

Nachdem das *MessageDigest* Objekt seine Eingabedaten erhalten hat, kann man den *digest* mit Hilfe einer der folgenden Methoden ausrechnen.

```
public byte[] digest()  
public byte[] digest(byte[] input)
```

Die letzte Methode ist äquivalent zu folgenden Aufrufen:

```
public void update(byte[] input)  
public byte[] digest()
```

#### 2.3.4 Die *Signature* Klasse

Die *Signature* Klasse ist eine *engine* Klasse, die die Funktionalität von kryptographischen Algorithmen für digitale Unterschriften (wie DSA oder RSA mit MD5) zur Verfügung stellt. So ein Algorithmus bekommt eine längenvariable Eingabe (ein Feld von Bytes) und ein privater Schlüssel und generiert daraus eine relativ kurze (oft längenkonstante) Ausgabe, die Unterschrift, mit folgenden Eigenschaften:

- Mit Hilfe des öffentlichen Schlüssels, der zum privaten Schlüssel gehört, mit dem die Eingabe generiert wurde, soll es möglich sein die Authentizität und die Integrität der Eingabe zu überprüfen.
- Die Unterschrift und der öffentlicher Schlüssel sollen keine Informationen enthalten, die es ermöglichen würde, etwas über den privaten Schlüssel herauszufinden.

### Zustände eines *Signature* Objektes

Ein *Signature* Objekt ist immer in einem bestimmten Zustand, wo es genau einen bestimmten Typ von Operation ausführen kann. Zustände werden als *final integer* Konstanten repräsentiert, die in den entsprechenden Klassen definiert sind.

Die drei möglichen Zustände eines *Signature* Objektes sind:

- UNINITIALIZED (nicht initialisiert)
- SIGN (unterschreiben)
- VERIFY (überprüfen)

### Ein *Signature* Objekt erzeugen

Um ein *Signature* Objekt zu erzeugen, ruft man die statische *factory* Methode der *Signature* Klasse auf:

```
public static Signature getInstance(String algorithm)
```

Optional kann man auch den Namen des gewünschten *providers* angeben:

```
public static Signature getInstance(String algorithm,  
String provider)
```

### Ein *Signature* Objekt initialisieren

Nach dem Erzeugen befindet sich ein *Signature* Objekt in dem Zustand UNINITIALIZED. Die Initialisierungsmethode hängt davon ab, ob das Objekt zuerst zum Unterschreiben oder zum Überprüfen benutzt werden soll.

Wenn das Objekt zunächst zum Unterschreiben benutzt werden soll, muß es zuerst mit dem privaten Schlüssel (der zum Unterschreiben benötigt wird) initialisiert werden. Dies wird mit der folgenden Methode, die das Objekt in den SIGN Zustand versetzt, realisiert:

```
public final void initSign(PrivateKey privateKey)
```

Wenn andererseits das Objekt zunächst zum Überprüfen benutzt werden soll, muß es zuerst mit dem öffentlichen Schlüssel (der zum Überprüfen benötigt wird) initialisiert werden. Dies wird mit der folgenden Methode, die das Objekt in den VERIFY Zustand versetzt, realisiert:

```
public final void initVerify(PublicKey publicKey)
```

### Unterschreiben

Wenn das *Signature* Objekt zum Unterschreiben initialisiert wurde (d.h.: wenn es sich im SIGN Zustand befindet), kann das Objekt mit den zu unterschreibenden Daten versorgt werden. Dies wird mit Hilfe einer oder mehreren Aufrufen folgender Methoden realisiert.

```
public final void update(byte b)  
public final void update(byte[] data)  
public final void update(byte[] data, int off, int len)
```

Um die Unterschrift zu generieren, muß man die `sign` Methode aufrufen:

```
public final byte[] sign()
```

Nachdem die `sign` Methode aufgerufen wurde, geht das Objekt wieder in den Zustand zurück, in dem es sich nach dem letzten Aufruf von `initSign` befand. Das Objekt ist jetzt wieder bereit, neue Daten mit dem gleichen privaten Schlüssel zu unterschreiben. Alternativ kann man mit `initSign` einen neuen privaten Schlüssel angeben, oder mit `initVerify` das Objekt in den Zustand bringen, in dem man es zum Überprüfen (Verifizieren) von Daten einsetzen kann.

### Verifizieren (Überprüfen)

Wenn das *Signature* Objekt zum Überprüfen initialisiert wurde (d.h.: wenn es sich im VERIFY Zustand befindet), kann das Objekt mit den zu überprüfenden Daten versorgt werden. Dies wird mit Hilfe einer oder mehreren Aufrufen folgender Methoden realisiert.

```

public final void update(byte b)
public final void update(byte[] data)
public final void update(byte[] data, int off, int len)

```

Um die Daten zu überprüfen, muß man die `verify` Methode aufrufen:

```

public final boolean verify(byte[] encodedSignature)

```

Die `verify` Methode gibt einen Wert des Typs `boolean` zurück. Je nachdem ob die Unterschrift der Daten authentisch ist, antwortet die Methode mit `true` oder mit `false`.

Nachdem die `verify` Methode aufgerufen wurde, geht das Objekt wieder in den Zustand zurück, in dem es sich nach dem letzten Aufruf von `initVerify` befand. Das Objekt ist jetzt wieder bereit, neue Daten mit dem gleichen öffentlichen Schlüssel zu überprüfen. Alternativ kann man mit `initVerify` einen neuen öffentlichen Schlüssel angeben, oder mit `initSign` das Objekt in den Zustand bringen, in dem man es zum Unterschreiben von Daten einsetzen kann.

### 2.3.5 Die *Key* Schnittstelle

Die *Key* Schnittstelle (interface) definiert die Funktionalität die alle *Key* Objekte gemeinsam haben. Alle *Key* Objekte besitzen die drei folgenden Eigenschaften:

- Einen Algorithmus, der benutzt wird um den Schlüssel (des *Key* Objektes) selbst zu verschlüsseln. Den Namen des Algorithmus wird mit der folgenden Methode erfragt:

```

public String getAlgorithm()

```

- Eine verschlüsselte Form, die benutzt wird, um den Schlüssel ausserhalb der virtuellen Maschine in einer standardisierten Form darzustellen. Die verschlüsselte Form wird von der folgenden Methode zurückgegeben:

```

public byte[] getEncoded()

```

- Das Format des verschlüsselten Schlüssels, wird von der folgenden Methode zurückgegeben:

```

public String getFormat()

```

### 2.3.6 Die *PublicKey* und *PrivateKey* Schnittstellen

Die *PublicKey* und *PrivateKey* Schnittstellen haben keine Methoden, und werden nur zur Typ-Sicherheit und Typ-Identifikation benutzt.

### 2.3.7 Die *KeyPair* Klasse

Ein Objekt der *KeyPair* Klasse enthält ein Schlüsselpaar (ein öffentlicher Schlüssel und ein privater Schlüssel). Mit zwei verschiedenen Methoden kann man den privaten und den öffentlichen Schlüssel erfragen:

```

public PrivateKey getPrivate()
public PublicKey getPublic()

```

### 2.3.8 Die *KeyPairGenerator* Klasse

Die *KeyPairGenerator* Klasse ist eine *engine* Klasse, die benutzt wird um ein Schlüsselpaar (privat und öffentlich) zu erzeugen. Die Erzeugung von Schlüsseln ist ein Gebiet das sich nicht besonders gut zur Algorithmusunabhängigkeit eignet. Zum Beispiel kann man bei der Erzeugung eines DSA Schlüsselpaares Parameter angeben, während dies bei RSA nicht möglich ist. Es gibt deswegen zwei Möglichkeiten ein Schlüsselpaar zu erzeugen: algorithmusunabhängig und algorithmusspezifisch.

#### Ein *KeyPairGenerator* Objekt erzeugen

Um ein *KeyPairGenerator* Objekt zu erzeugen, ruft man eine der beiden statischen *factory* Methoden der *KeyPairGenerator* Klasse auf:

```
public static KeyPairGenerator getInstance(String
    algorithm)
public static KeyPairGenerator getInstance(String
    algorithm, String provider)
```

#### Ein *KeyPairGenerator* Objekt initialisieren

Ein *KeyPairGenerator* Objekt muß vor der Benutzung initialisiert werden. In den meisten Fällen reicht eine algorithmusunabhängige Initialisierung. Ist jedoch die Kontrolle über spezifische Parameter eines bestimmten Algorithmus erwünscht, soll man eine algorithmusspezifische Initialisierung verwenden.

#### Algorithmusunabhängige Initialisierung

Alle *KeyPairGenerator* Objekte benutzen zur Initialisierung die beiden Parameter *strength* und *random*. Der Parameter *strength* regelt die Stärke des kryptographischen Verfahrens und hängt in den meisten Fällen mit der Schlüssellänge zusammen. *Strength* wird jedoch von unterschiedlichen Algorithmen unterschiedlich interpretiert. *Random* ist vom Typ *SecureRandom* und stellt eine zuverlässige Quelle von Zufallsdaten dar. Folgende Methode führt eine algorithmusunabhängige Initialisierung eines *KeyPairGenerator* Objektes durch:

```
public void initialize(int strength, SecureRandom random)
```

#### Algorithmusspezifische Initialisierung

Folgende Methode wird benutzt um ein *KeyPairGenerator* Objekt algorithmusspezifisch zu initialisieren: (Beispielalgorithmus: DSA)

```
public void initialize(DSAParams params, SecureRandom
    random)
```

Beispiel eines Codefragments, das eine algorithmusspezifische Initialisierung durchführt:

```
DSAParams dsaParams = new DSAParams(p, q, g);
DSAKeyPairGenerator dsaKeyGen = (DSAKeyPairGenerator)keyGen;
dsaKeyGen.initialize(dsaParams, new
    SecureRandom(userSeed));
```

#### Ein *KeyPair* Objekt erzeugen

Ein *KeyPair* (Schlüsselpaar) Objekt wird (unabhängig von der Initialisierungsmethode) immer mit folgender Methode erzeugt:

```
public KeyPair generateKeyPair()
```

Mehrfache Aufrufe erzeugen unterschiedliche Schlüsselpaare.

### 2.3.9 Die *SecureRandom* Klasse

Die *SecureRandom* Klasse ist eine *engine* Klasse, die benutzt wird um plattformunabhängig softwarebasierte Zufallszahlen hoher Qualität zu erzeugen.

#### Ein *SecureRandom* Objekt erzeugen

Es gibt zwei Methoden ein *SecureRandom* Objekt zu erzeugen: Mit oder ohne *seed*. Fehlt das *seed*, wird mit dem Standard-*Seed*mechanismus einen *seed* erzeugt. Mit den folgenden Konstruktoren werden die Objekte erzeugt:

```
public SecureRandom()
public SecureRandom(byte[] seed)
```

Leider befindet sich der Standard-*Seed*mechanismus von JDK 1.1 noch im Experimentierstadium, so daß man einen eigenen *seed* benutzen sollte.

#### Zufallszahlen erzeugen

Um Zufallszahlen zu erzeugen, übergibt man folgender Methode ein beliebig großes Feld von Bytes.

```
public void setSeed(byte[] seed)
```

Anschließend wird das Feld mit Zufallsdaten gefüllt.

#### Re-Seeding a *SecureRandom* Object

Zu jeder Zeit kann man das *SecureRandom* Objekt mit einem neuen *seed* versorgen:

```
public void setSeed(byte[] seed)
```

Zu beachten ist, daß das neue *seed* das alte *seed* nicht ersetzt, sondern ergänzt (d.h. dazugerechnet wird). Mehrere `setSeed` Aufrufe reduzieren die Zufälligkeit der Zufallszahlen also nicht.

## 2.4 Ein kleines Beispiel

Zum Abschluß der Klassenbeschreibungen noch ein kleines Beispiel zur Veranschaulichung:

```
import java.io.*;
import java.security.*;

public class Javatest {

    private static byte data[] = { 'H', 'e', 'l', 'l', 'o' };

    public static void main(String argv[])
    {
        try {
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
            keyGen.initialize(1024, new SecureRandom());
            KeyPair pair = keyGen.generateKeyPair();

            Signature dsa = Signature.getInstance("DSA");
```

```

        PrivateKey priv = pair.getPrivate();
        dsa.initSign(priv);
        dsa.update(data);
        byte[] sig = dsa.sign();

        PublicKey pub = pair.getPublic();
        dsa.initVerify(pub);
        dsa.update(data);
        boolean verifies = dsa.verify(sig);

        System.out.println("signature verifies: " + verifies);
    } catch (NoSuchAlgorithmException except1) {
    }

    catch (InvalidKeyException except2) {
    }

    catch (SignatureException except3) {
    }
}
}
}

```

Das kleine Programm erzeugt zunächst ein Schlüsselpaar. Dann wird eine digitale Unterschrift des Feldes `data` erzeugt. Anschließend wird die digitale Unterschrift auf ihre Authentizität überprüft. Verwendeter Algorithmus: DSA

## 3 Java Sicherheits-Werkzeuge

### 3.1 Das JAR Archiv-Format

#### 3.1.1 Einführung

JAR (**J**ava **A**rchive) ist ein Archiv-Format und kann mehrere Dateien zu einer einzigen Datei zusammenfassen. Ein *Applet*, das aus mehreren `.class`-, Bild- und Audiodateien besteht, kann so mit einer einzigen HTTP-Verbindung übertragen werden, anstatt daß für jede einzelne Datei eine neue HTTP-Verbindung geöffnet werden muß. Dies stellt ein großer Geschwindigkeitsvorteil dar. JAR basiert auf dem verbreiteten ZIP-Format (das von PKWARE entworfen wurde), und unterstützt Dateikomprimierung. Zusätzlich können JAR-Dateien digital unterschrieben (signiert) werden. JAR ist erweiterbar und in Java implementiert.

#### 3.1.2 Das APPLET *tag*

Mit Hilfe von folgendem HTML Code kann man JAR Dateien übers WWW laden. Der `ARCHIVE` Parameter gibt den Pfad und den Namen der JAR Datei relativ zum Pfad der HTML-Seite auf dem Server an. Der `CODE` Parameter gibt den Namen der Klasse an, die zuerst ausgeführt werden soll.

```

<applet code=Animator.class
archive="jars/animator.jar"
width=460 height=160>
<param name=foo value="bar">
</applet>

```

### 3.1.3 Das JAR-Werkzeug

JAR Archive werden mit dem von SUNs JDK mitgelieferten JAR Werkzeug erzeugt. Die JAR Parameter sind an den Parametern des UNIX tar Kommandos angelehnt. Die drei Eingabedateiparameter für das JAR Werkzeug sind:

- *manifest* Datei (optional)
- JAR Zieldatei
- Dateien die archiviert werden sollen

Typischer JAR Aufruf:

```
jar cf myjarfile *.class
```

Dieser JAR Aufruf packt alle `.class` Dateien des gerade benutzten Verzeichnisses in die Datei `myjarfile`. Eine *manifest*-Datei mit dem *default*-Namen `META-INF/MANIFEST.INF` wird automatisch erzeugt. Die *manifest*-Datei enthält alle Metainformationen des Archivs. Soll eine bereits existierende *manifest*-Datei für das neue JAR Archiv benutzt werden, kann man dies mit der `-m` Option angeben:

```
jar cmf myManifestFile myjarfile *.class
```

Wenn eine Datei in ein JAR Archiv hinzugefügt wird, werden die *message digests* der Datei nach den Algorithmen MD5 und SHA ausgerechnet und in die *manifest*-Datei abgespeichert.

## 3.2 Das *jarsigner*-Werkzeug

*Jarsigner* ist das neue JDK 1.2 Werkzeug um JAR Dateien digital zu unterschreiben und um die Authentizität und Integrität digital unterschriebener JAR Dateien zu überprüfen. Zusammen mit dem neuen *keytool*-Werkzeug des JDK 1.2 ersetzt es das alte *javakey*-Werkzeug des JDK 1.1 komplett. Die beiden neuen Werkzeuge bieten mehr Möglichkeiten als *javakey*.

*Keystore* ist eine Datenbank die private Schlüssel, öffentliche Schlüssel und Zertifikate enthält, die von *jarsigner* benutzt werden. Ein Zertifikat ist eine digital signierte Bestätigung einer vertrauenswürdigen Organisation, die aussagt, daß die öffentlichen Schlüssel gültig sind. *Keystore* ist nicht abwärtskompatibel mit der *identity*-Datenbank von *javakey*. Es ist aber möglich mit Hilfe von *keytool -identitydb* Informationen aus *identity* nach *keystore* zu übernehmen. *jarsigner* kann aber auf Informationen aus der *identity*-Datenbank zugreifen und mit *javakey* signierten JAR Dateien umgehen. Alle Schlüssel der *keystore*-Datenbank werden mit Hilfe eines *alias* aufgerufen.

Um eine JAR Datei mit *jarsigner* zu signieren, muß man den *alias* des *keystore*-Eintrages des privaten Schlüssels angeben, der benutzt wird um die Unterschrift zu erzeugen. Folgender Aufruf unterschreibt die JAR Datei `MyJARFile.jar` mit Hilfe des privaten Schlüssels der zum Alias `duke` gehört aus der *keystore*-Datenbank `/working/mystore`. Da keine Ausgabedatei angegeben wurde, wird die `MyJARFile.jar` Datei mit der signierten Datei überschrieben.

```
jarsigner -keystore /working/mystore -storepass myspass  
-keypass dukekeypasswd MyJARFile.jar duke
```

*Keystore*-Datenbanken werden von einem Paßwort geschützt. In diesem Fall heißt das Paßwort `mypass`. Darüber hinaus werden alle Einträge, die einen privaten Schlüssel enthalten, zusätzlich von einem Paßwort geschützt. In diesem Fall

heißt das Paßwort `dukekeypasswd`. Dies stellt einen Fortschritt gegenüber JDK 1.1 und *javakey* dar, wo die Datenbank und die Einträge mit privaten Schlüsseln nicht von einem Paßwort geschützt werden konnten.

Mit der `-keystore` Option kann man mit *jarsigner* auch auf ein *keystore* über eine URL zugreifen.

Mit folgendem Aufruf kann man die Authentizität einer JAR Datei überprüfen:

```
jarsigner -verify MyJARFile.jar
```

### 3.3 Das *keytool*-Werkzeug

*Keytool* ist das neue JDK 1.2 Werkzeug um *keystore*-Datenbanken zu verwalten. Das Format einer *keystore*-Datenbank ist standardmäßig eine Textdatei, kann aber (abhängig von gewählten Provider) in irgendein anderes Format (z.B. binäres Format) abgespeichert werden.

Folgender *keytool* Aufruf erzeugt eine *keystore*-Datenbank mit dem Namen `mykeystore` im Unterverzeichnis `working`. `ab987c` ist das neue Paßwort der *keystore*-Datenbank. Anschließend wird ein neues Schlüsselpaar mit einem öffentlichen und privaten Schlüssel erzeugt und in die Datenbank eingetragen. Der Besitzer des Schlüsselpaares heißt `Mark Jones` der Organisation `Sun` (Unterabteilung `JavaSoft` und Ländercode `US`). Das Paßwort des privaten Schlüssel heißt `kpi135` und der Alias heißt `business`.

```
keytool -genkey -dname cn=Mark Jones, ou=JavaSoft,  
o=Sun, c=US -alias business -keypass kpi135 -keystore  
/working/mykeystore -storepass ab987c -validity 180
```

Die Schlüssel werden standardmäßig mit dem Algorithmus DSA erzeugt und haben eine Länge von 1024 Bit. Es wird ein (selbstunterschriebenes) Zertifikat erzeugt. Dieses Zertifikat hat eine Lebensdauer von 180 Tagen.

## 4 Die Java-*Sandbox*

### 4.1 Einführung

Seit Mitte sechziger Jahren ist es bei ausgewachsenen Betriebssystemen (Multix, Unix) üblich, verschiedene Prozesse in abgetrennten Adressräumen (Speicherschutz) auszuführen. Dabei darf kein Prozeß in den Speicherbereich eines anderen Prozesses hineinschreiben. Diese Funktionalität wird direkt vom Prozessor durch Hardware unterstützt. Darüber hinaus kann man auch die Zugriffsrechte der Prozesse auf Dateien und aufs Netzwerk einschränken.

Trotzdem können unter Unix, vom Anwender gestartete Programme in der Regel alle Dateien die im Besitz dieses Anwenders sind, lesen und ändern, es sei denn, der Anwender richtet sich unter Unix ein Benutzer-Account ein, wo nur dieses eine Programm läuft.

Darüber hinaus bieten Betriebssysteme wie MS-DOS überhaupt keinen Speicherschutz an. Das Betriebssystem Windows (ausgenommen NT) bietet zwar Speicherschutz, aber keinen Schutz vor unbefugtem Zugriff auf Dateien, da es ein Single-User Betriebssystem ist. Unix nützt in diesem Fall auch nicht viel, wenn man regelmäßig unter dem *root*-Account arbeitet.

### 4.2 Die *Sandbox*

Insbesondere Anwendungen die transparent übers Netz geladen werden und gestartet werden können, ohne daß der Anwender was bemerkt, stellen ein großes

Sicherheitsrisiko dar. Deshalb läuft jede Java-Anwendung in einer Art *Sandbox* (Sandkasten), die die Rechte einer Anwendung einschränkt. Diese *Sandbox* arbeitet Betriebssystem- und Prozessorunabhängig und ist in der Java Virtual Machine verankert.

In JDK 1.1 hatte man jedoch relativ wenige Möglichkeiten die Rechte eines Java-Programms genau zu bestimmen. Ein Securitymanager bestimmte auf welche Ressourcen das Programm zugreifen durfte. Ein Securitymanager enthält Methoden (`checkRead`, `checkWrite`, `checkConnect`, `checkPrintJobAccess`), die das Programm aufrufen konnte, um zu prüfen ob es die Zugriffsrechte auf eine bestimmte Ressource (z.B. Schreib- und Leserechte für eine bestimmte Datei oder Verzeichnis, Zugriffsrecht zu einem bestimmten TCP/IP Port) besitzt. Ist dies nicht der Fall, dann wirft die entsprechende Methode eine *Exception*.

Traditionell durften die vom Web heruntergeladene *Applets* nicht auf das lokale Dateisystem zugreifen. Der einzige Server auf den das Applet übers Netzwerk zugreifen durfte, war der Server von dem das Applet geladen wurde. Da diese Einschränkungen manchmal zu eng waren, konnte man den Applets die gleichen Rechte wie lokale Anwendungen geben, indem man sie signieren ließ.

Die führte jedoch dazu, daß es zwei Arten von Java-Programmen gab: Die die fast alles durften und die die fast nichts durften.

Deswegen wurde in JDK 1.2 eine neue *Policy*-basierte Sicherheitsarchitektur eingeführt. Diese Sicherheitsarchitektur bestimmt, welcher Code von welchem Unterschreiber auf welche Ressourcen zugreifen darf. Rechte müssen explizit zugeteilt werden. Werden Rechte nicht explizit zugeteilt, dann darf die entsprechende Applikation auch nicht auf die entsprechende Ressource zugreifen. Dieses neue Konzept erlaubt eine feinkörnige, hochkonfigurierbare, flexible und erweiterbare Zugriffskontrolle.

## 5 Verschiedene kryptographische Algorithmen

### 5.1 Einführung

### 5.2 Symmetrische Verschlüsselungsverfahren

#### 5.2.1 Einführung

Symmetrische Verschlüsselungsverfahren benutzen nur einen einzigen Schlüssel, der den beiden Gesprächspartnern bekannt sein muß. Der Nachteil besteht darin, einen zuverlässigen Weg zu finden, dem anderen Gesprächspartner den Schlüssel mitzuteilen. Dieses Kapitel befaßt sich mit den beiden symmetrischen Verschlüsselungsverfahren DES und IDEA.

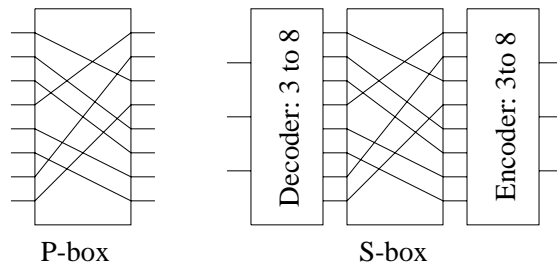
#### 5.2.2 DES

##### Transpositionen und Substitutionen

Eine *Transposition* wird mit Hilfe einer *P-box* (P steht für Permutation) durchgeführt. Mit Hilfe einer *Transposition* werden die einzelnen Leitungen einer Eingabe auf eine andere Position abgebildet. Sind die einzelnen 8 Leitungen der Eingabe durchnummeriert (01234567) dann werden diese in diesem Beispiel in der Ausgabe der *P-Box* folgende Reihenfolge haben: 36071245.

Eine *Substitution* wird mit Hilfe einer *S-box* durchgeführt. Eine *S-box* bildet eine binär kodierte Zahl auf eine beliebig andere Zahl ab. Dabei wird zuerst die binäre Zahl dekodiert: Bei einer 3-bittigen Zahl wird eine der 8 Leitungen ausgewählt und auf 1 gesetzt, alle anderen Leitungen werden auf 0 gesetzt. Die *P-box* bildet diese

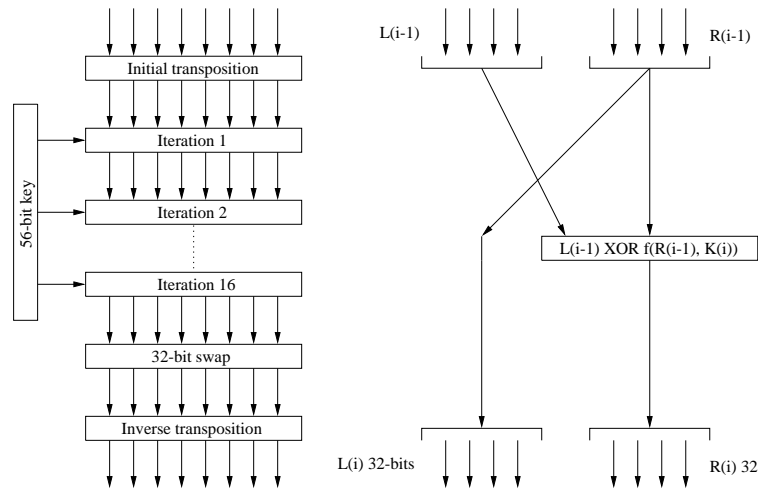
Leitung auf eine andere Position ab. Dann wird diese Position wieder in eine binäre Zahl kodiert.



## DES

Die Eingabe wird in Blöcken von 64 Bit kodiert. Der Algorithmus wird von einem 56-bit Schlüssel parametrisiert und besitzt 19 verschiedene Schritte. Der erste Schritt besteht aus einer schlüsselunabhängigen Transposition. Der letzte Schritt besteht aus der gegenteiligen Transposition. Der zweitletzte Schritt vertauscht die linken 32 Bit mit den rechten 32 Bit. Die anderen 16 Schritte sind eigentlich identisch, werden nur von anderen Funktionen des Schlüssels parametrisiert. Entschlüsselung funktioniert mit dem gleichen Schlüssel, nur in umgekehrter Reihenfolge.

Die Funktion einer der mittleren Schritte wird im Bild unten rechts dargestellt. Jeder Schritt nimmt zwei 32-bit Eingaben und generiert zwei 32-bit Ausgaben. Die linke Ausgabe  $L_i$  ist eine Kopie der rechten Eingabe  $R_{i-1}$ . Die rechte Ausgabe  $R_i$  ist die XOR Verknüpfung der linken Eingabe  $L_{i-1}$  und einer Funktion der rechten Eingabe  $R_{i-1}$  und des Schlüssels für diesen Schritt,  $K_i$ . Die ganze Komplexität liegt in dieser Funktion.



Die Funktion  $f$  besteht aus 4 Schritten, die nacheinander ausgeführt werden. Zuerst wird nach einer fixen Transpositions- und Duplikationsregel die 32-bittige  $R_{i-1}$  zu einer 48-bittigen Zahl  $E$  expandiert. Dann werden  $E$  und  $K_i$  mit XOR verknüpft. Diese Ausgabe wird dann in 8 Gruppen von 6 Bit aufgeteilt, und jede einzelne Gruppe wird in eine verschiedene  $S$ -boxen eingespeist. Jede der 64 verschiedenen Eingaben einer  $S$ -box wird auf eine 4-bittige Ausgabe abgebildet. Schließlich werden diese  $8 \times 4$  bits durch eine  $P$ -box geschickt.

In jedem der 16 Schritte wird ein anderer Schlüssel benutzt. Bevor der Algorithmus startet, wird eine 56-bit Transposition auf den Schlüssel angewendet. Vor jeder Iteration wird der Schlüssel in zwei 28-bit Einheiten aufgeteilt, jede Einheit

wird eine gewisse Anzahl von Bits nach links rotiert, abhängig von der Iterationsnummer.  $K_i$  wird abgeleitet, indem man auf diesen rotierten Schlüssel nochmals eine 56-bit Transposition anwendet, und abhängig von der Iterationsnummer, eine 48-bit Untermenge extrahiert und permutiert.

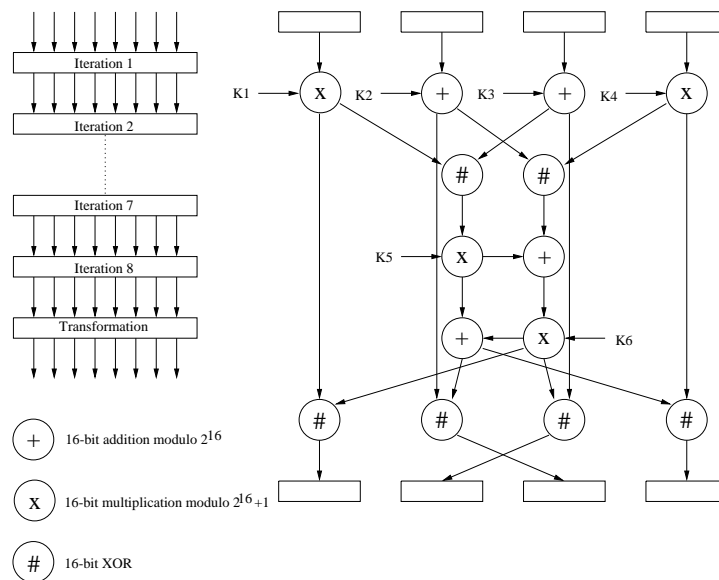
### Ausblick

DES wurde von IBM entwickelt, und wurde im Januar 1977 offiziell von der U.S. Regierung als Standardverschlüsselungsverfahren aufgenommen. Der Originalalgorithmus von IBM (Lucifer) benutzte einen 128-bittigen Schlüssel. Als sich die U.S. Regierung am Algorithmus interessierte, beschloß man mit IBM, den Schlüssel auf 56 Bit zu reduzieren, (höchstwahrscheinlich) damit die NSA (National Security Agency) DES knacken kann. Heutzutage ist die Schlüssellänge von 56 Bit nicht mehr sicher. DES ist immer noch brauchbar, wenn man die Daten dreifach verschlüsselt (mit zwei verschiedenen Schlüssel), so daß man eine effektive Schlüssellänge von 112 Bit hat. Eine zweifache Verschlüsselung (mit zwei verschiedenen Schlüssel) ist nicht viel sicherer als eine einfache Verschlüsselung (*meet-in-the-middle* attack).

Darüberhinaus war DES vom ersten Tag an ein umstrittenes Verschlüsselungsverfahren, da viele Leute den Verdacht hatten, daß sich die NSA eine geheime Hintertür in den Algorithmus eingebaut hatte.

### 5.2.3 IDEA

IDEA (International Data Encryption Algorithm) wurde von zwei schweizer Forschern entwickelt, und enthält wahrscheinlich keine geheimen Hintertüren. Der Algorithmus benutzt eine Schlüssellänge von 128 Bit, was ausreicht um Schutz gegen *brute force* (rohe Gewalt) Attacken für die nächsten Jahrzehnte zu bieten.



## 5.3 Asymmetrische Verschlüsselungsverfahren

### 5.3.1 Einführung

Asymmetrische Verschlüsselungsverfahren benutzen zwei Schlüssel: einen öffentlichen und einen privaten. Der öffentliche Schlüssel kann aus dem privaten Schlüssel abgeleitet werden, aber nicht umgekehrt. Mit dem öffentlichen Schlüssel kann nur verschlüsselt werden, aber nicht entschlüsselt werden. Entschlüsseln kann nur der Besitzer des privaten Schlüssels.

### 5.3.2 RSA

RSA (**R**ivest, **S**hamir, **A**dleman) ist ein sehr populäres asymmetrisches Verschlüsselungsverfahren und basiert auf der Annahme, daß es sehr schwierig ist große Zahlen in ihre Primfaktoren zu zerlegen.

### 5.4 *Message Digests*

Die beiden bekanntesten *Message Digests* Algorithmen heißen MD5 und SHA. Da SHA vom NSA entworfen wurde ist es politisch im Internet nicht so beliebt wie SHA.

### 5.5 Digitale Unterschriften

Digitale Unterschriften basieren sehr oft auf asymmetrische Verschlüsselungsverfahren. Dabei kann der Unterschreiber eine Unterschrift erzeugen, indem er seine Daten mit seinem privaten Schlüssel 'entschlüsselt'. Die Empfänger der Daten können die Authentizität der Daten überprüfen, indem sie die Unterschrift mit dem öffentlichen Schlüssel des Unterschreibers 'verschlüsseln'. Da asymmetrische Verschlüsselungsverfahren für große Datenmengen zu langsam sind, wird normalerweise mit einem *Message Digest* Algorithmus einen Fingerabdruck der Daten erzeugt, und nur der Fingerabdruck wird anschließend unterschrieben. Verbreitete Algorithmen sind: DSS, DSA mit RSA, SHA mit RSA, MD5 mit RSA