Asymmetrische Kryptografie in Java

Sichere verteilte Anwendungen mit Java

Alexander H. W. Lindemann

17. Januar 2014

Inhaltsverzeichnis

1 Theoretische Grundlagen

1.1 Ziele

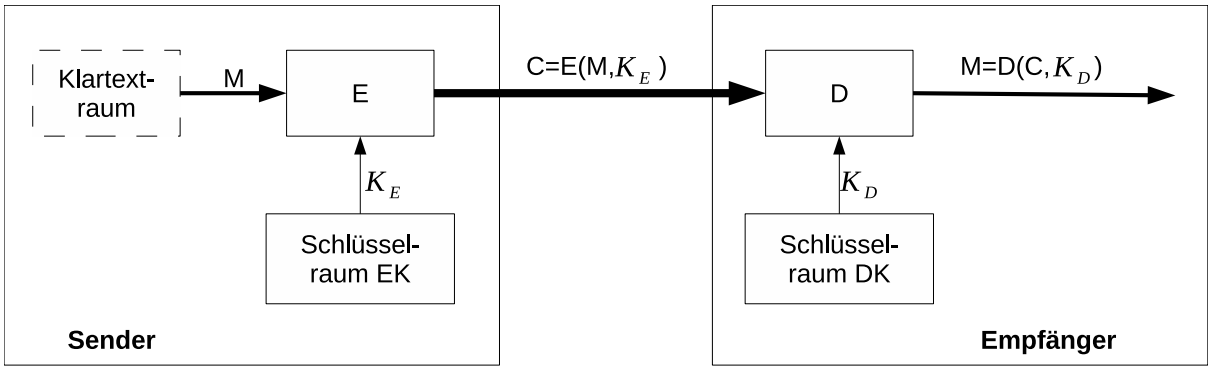
Asymmetrische Kryptografie und Kryptografie im Allgemeinen haben zum Ziel, Nachrichten (Daten, Texte, etc.) so zu manipulieren, dass sie nur vom jeweiligen Empfänger gelesen werden können. Hierbei werden komplexe mathematische Probleme als Grundlage gewählt, um das unberechtigte Lesen der Nachricht zu verhindern.

1.2 Asymmetrische Kryptografie

Im Gegensatz zur symmetrischen Kryptografie, werden bei der asymmetrischen Kryptografie 2 Schlüssel, also ein Schlüsselpaar, benutzt. Einer der beiden wird als öffentlicher Schlüssel, der andere als privater Schlüssel bezeichnet. Mit dem öffentlichen Schlüssel werden die Nachrichten verschlüsselt und mit dem privaten Schlüssel wieder entschlüsselt. Daraus folgt, das der private Schlüssel nur dem Empfänger bekannt sein sollte.

1.3 SYSTEM

Abbildung 1.1: Kryptosysteme nach [Eckert, 2013]



- Tupel = (M, C, EK, DK, E, D)
- 2 endliche Alphabete (A_1, A_2)
- Klartext $(M \subseteq A_1^* \backslash \emptyset)$
- Kryptotext $(C \subseteq A_2^* \backslash \emptyset)$
- Verschlüsselungsschlüsselraum $(EK \backslash \emptyset)$
- Es gilt: $\forall m \in M : D(E(m, K_E), K_D) = m$
- Entschlüsselungsschlüsselraum $(DK/\emptyset$ $mit\ f : EK \rightarrow DK$ $und\ f(K_E) = K_D)$
- Verschlüsselungsverfahren $(E :\ MxEK \rightarrow C)$
- Entschlüsselungsverfahren $(D :\ CxDK \rightarrow M)$

Ein Kryptosystem besteht, laut [Eckert, 2013] aus den in Abbildung 1.1 dargestellten Komponenten.

1.4 Einwegfunktionen

Eine Einwegfunktion ist eine Funktion, welche nach komplexitätstheoretischen Maßstäben leicht zu berechnen, aber schwer umzukehren ist. Meist werden Funktionen so bezeichnet welche sich nicht in angemessener Zeit umkehren lassen. Diese Eigenschaft macht sie ideal zur Verwendung in Kryptografischen Algorithmen, da so sichergestellt ist das die Nachricht nicht unberechtigt entschlüsselt wird. Leider verhindern solche Funktionen das Entschlüsseln vollständig, weshalb eine andere Kategorie von Funktionen zum Einsatz kommt, die so genannten **Einwegfunktionen mit Falltür**.

Einwegfunktionen mit Falltür haben den Vorteil, dass mit gewissem Zusatzwissen die Umkehrung leicht zu berechnen ist. Dieses Zusatzwissen wird in der asymmetrischen Kryptografie im privaten Schlüssel gespeichert, was garantiert das nur der richtige Empfänger die Nachricht entschlüsseln kann.

1.5 Beispiel RSA

Ein Beispiel für ein asymmetrisches Verschlüsselungsverfahren ist der *RSA* Algorithmus. Dieser basiert auf einer Einwegpermutation mit Falltür.

1.5.1 Schlüsselgenerierung

Ein Schlüsselpaar besteht bei RSA aus folgenden Komponenten:

- Öffentlicher Schlüssel - (e, N)
- Privater Schlüssel - (d, N)

Wobei N das *RSA-Modul* genannt wird, e der Verschlüsselungsexponent und d der Entschlüsselungsexponent. Diese Zahlen werden nach folgendem Verfahren bestimmt:

1. Man wähle 2 zufällige und voneinander (stochastisch) unabhängige Primzahlen $p \neq q$
2. Nun wird $N = p \cdot q$ berechnet
3. Danach muss $\varphi(N) = (p-1) \cdot (q-1)$ bestimmt werden
4. Man wähle nun eine Zahl zu $\varphi(N)$ teilerfremde Zahl e für die gilt: $1 < e < \varphi(N)$
5. d wird nun als multiplikativ Inverses zu e bzgl. $\varphi(N)$ bestimmt.
   $(e \cdot d \equiv 1 \ mod \ \varphi(N))$

Nach der Generierung der Schlüsseldaten d und e können und sollten $\varphi(N)$, p und q gelöscht werden, um die unbefugte Rekonstruktion des Privaten Schlüssels zu verhindern.

1.5.2 Ver- und Entschlüsselung

Um mithilfe eines Schlüsselpaares Nachrichten zu ver- und entschlüsseln kommen folgende Formeln zum Einsatz:

- $c \equiv m^e \ mod \ N$
- $m \equiv c^d \ mod \ N$

Wobei m die Nachricht und c den aus der Nachricht abgeleiteten chiffrierten Text darstellt.

Da die Operationen auch invers durchführbar sind, man also mit dem privaten Schlüssel Nachrichten chiffrieren kann welche dann durch den öffentlichen Schlüssel dechiffriert werden können, kann man mit diesem Verfahren eine so genannte digitale Signatur durchführen. Hierbei wird meist auf den Hash einer Nachricht der eigene privaten Schlüssel angewendet und dieser dann an die Nachricht angehängt. Der Empfänger kann nun eindeutig bestimmen ob die Nachricht während des Transports verändert wurde (der Hash würde dann nicht stimmen) und ob die Nachricht wirklich vom angegebenen Versender verfasst wurde (der öffentliche Schlüssel muss zur Signatur passen).

1.6 Hybride Kryptografie

Hybride Kryptografie ist eine Kombination aus symmetrischer und asymmetrischer Kryptografie. Hierbei wird eine Nachricht erst symmetrisch verschlüsselt und der symmetrische Sitzungsschlüssel dann über einen asymmetrisch gesicherten Kanal ausgehandelt oder an die Nachricht angehängt. Der Empfänger kann den Sitzungsschlüssel dann entschlüsseln und so die Nachricht lesen.

Dieses Verfahren hat den Vorteil das die komplexen Berechnungsschritte, welche für asymmetrische Verfahren notwendig sind, nur auf den Sitzungsschlüssel angewendet werden.

Ein weiterer Vorteil besteht darin, dass eine gebrochene Nachricht nicht die Kompromittierung der gesamten Kommunikation bedeutet sondern nur die Offenlegung der jeweiligen Sitzung.

2 Kryptografie mit Java

2.1 Java Cryptography Architecture

2.2 Zufallszahlen

Ein wichtiger Punkt bei der Erzeugung von Schlüsseln und der anschließenden Durchführung von Kryptooperationen ist die Bereitstellung von geeigneten Zufallszahlen, da sonst das Erraten des Schlüssels oder das zurückrechnen von Kryyptooperationen zu einfach werden könnte.

Die JCA bietet hier die Möglichkeit über die Klasse *java.security.SecureRandom* eigene Zufallszahlengeneratoren per Kryptoprovider bereitzustellen. Weiterhin wird mit der SunJCE-Implementierung ein eigener Generator angeboten. Hierbei müssen einigen Richtlinien für Zufallszahlengeneratoren beachtet werden, unter anderem die *FIPS 140-2*-Richtlinie[1] und der *RFC 1750*[2]. Weiterhin ist zu beachten das der Seed mit dem die Generatoren initialisiert werden zufällig sein sollte.

2.3 Schlüsselgenerierung

Abbildung 2.1: Codebeispiel KeyPairGenerator

```
KeyPairGenerator keyGen =
        KeyPairGenerator.getInstance( "RSA" );
keyGen.initialize( int keysize );
KeyPair keys = keyGen.generateKeyPair();
```

(KeyGenerator ist analog zu benutzen)

Schlüssel werden unter Java mit den Klassen *java.security.KeyPairGenerator* (Schlüsselpaar) bzw. *javax.crypto.KeyGenerator* (Symmetrischer Schlüssel) erzeugt. Diese können über die in Abbildung 2.1 dargestellten Codezeilen instanziiert und genutzt werden. Hierbei ist es möglich den gewünschten Algorithmus, die Schlüssellänge und den Zufallszahlengenerator als Parameter zu übergeben.

2.4 Schlüsselspeicherung

Schlüssel werden in Java mithilfe der Klasse *java.security.KeyStore* gespeichert, jedoch ist es auch möglich die Schlüsselpaare als *java.security.KeyPair* in serialisierter Form auf die Festplatte zu speichern. Dabei werden jedoch die Schlüsseldaten nicht gegen unbefugtes öffnen gesichert, deshalb ist in produktiven Umgebungen immer eine Speicherung via Keystore vorzuziehen. Der

[1] http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
[2] http://www.ietf.org/rfc/rfc1750.txt

Keystore erlaubt die Speicherung einer belieben Anzahl von Schlüsseln und Zertifikaten, sowie die Sicherung dieser mithilfe unterschiedlicher Passwörter, sowie weiteren Mechanismen.

Eine weitere Methode zur Schlüsselspeicherung wird über das Interface *java.security.Key* bereitgestellt. Die Methode *getEncoded()* liefert die Repräsentation des Schlüssels in einem, per *getFormat* abzurufenden, Format, Beispielsweise *x.509* oder *PKCS #8*.

2.5 Schlüsseleinigung

Abbildung 2.2: Codebeispiel KeyAgreement

```
KeyAgreement aKeyAgree =
        KeyAgreement.getInstance("DH", "JCE");
KeyAgreement bKeyAgree =
        KeyAgreement.getInstance("DH", "JCE");

KeyPair aPair = keyGen.generateKeyPair();
KeyPair bPair = keyGen.generateKeyPair();

aKeyAgree.init(aPair.getPrivate());
bKeyAgree.init(bPair.getPrivate());

aKeyAgree.doPhase(bPair.getPublic(), true);
bKeyAgree.doPhase(aPair.getPublic(), true);

SecretKey aSecret =
        aKeyAgree.generateSecret("AES");
SecretKey bSecret =
  bKeyAgree.generateSecret("AES");
```

Falls während eines Verbindungsaufbaus eine Schlüsseleinigung notwendig sein sollte, so wird dies durch die Klasse *javax.crypto.KeyAgreement* ermöglicht. Hierbei wird meist[3] der Diffie–Hellman Algortihmus benutzt. In der Abbildung 2.2 wird gezeigt wie diese Klasse zu verwenden ist. Hierbei ist zu beachten, das nach der Intitialisierung mit dem eigenen privaten Schlüssel, die Methode *doPhase()* mit den öffentlichen Schlüsseln der Gesprächspartner ausgeführt wird. Beim Aufruf der Methode mit dem Schlüssel des letzten Partners ist der zweite Parameter auf „true“ zu setzen, um dem System den Abschluss der Operation zu signalisieren. Abschließend kann dann mit der Methode *generateSecret()* ein symmetrischer Sitzungsschlüssel erzeugt werden.

2.6 Ver- und Entschlüsselung

[3]SunJCE und Bouncy Castle unterstützen hier nur Varianten des Diffie–Hellman Algorithmus

Abbildung 2.3: Codebeispiel KeyAgreement

```
Cipher cipher =
        Cipher.getInstance( "RSA", "BC" );
cipher.init( Cipher.ENCRYPT_MODE, publicKey );
cipher.update( message );
byte[] crypt = cipher.doFinal();
```

Literaturverzeichnis

[Eckert, 2013] Eckert, C. (2013). *IT-Sicherheit : Konzepte - Verfahren - Protokolle*. Oldenburg.

[Engelbrecht, 2004] Engelbrecht, M. (2004). *Entwicklung sicherer Software - Modellierung und Implementierung mit Java*. Spektrum.