Asymmetrische Kryptographie in Java

Norman Vetter

Seminar „Sichere verteilte Anwendungen mit Java“

Universität Potsdam

Wintersemester 2012/13

Inhaltsverzeichnis

1 Einleitung

In Zeiten des immer stärker werdenden Aufkommens an digitaler Kommunikation gerät der Schutz von privaten oder wichtigen Nachrichten und Daten immer mehr ins Augenmerk der Menschen. Somit werden Mechanismen zum Bewahren dieser Privatsphäre und Sicherheit zunehmend wichtiger. Eine Methode diese zu erhöhen ist das Verschlüsseln von Daten mittels asymmetrischer Kryptographie, wie dies in der Programmiersprache Java vorgenommen wird und auf welchen Grundlagen diese Form der Verschlüsselung beruht wird in dieser Ausarbeitung erläutert.

2 Theoretische Grundlagen

2.1 Ziele

Ein kryptographisches System dient zum verschlüsseln und entschlüsseln von Texten und anderen Daten, um deren Inhalt vor Dritten geheim zu halten. Genauer gibt ein Kryptosystem an wie ein Klartext in einen von Dritten nicht lesbaren Kryptotext umgewandelt werden kann. Und wie dieser Kryptotext später wieder in einen lesbaren Klartext transformiert wird. Anders als die Steganografie zielt die Kryptographie darauf ab lediglich den Inhalt einer Nachricht zu verschlüsseln, nicht aber deren Existenz zu verbergen. Als oberstes Prinzip ist zu beachten das die Sicherheit allein vom Schlüssel, nicht aber vom Verfahren abhängig ist.(Kerckhoff Prinzip [1]) Die asymmetrische Kryptographie ist eines dieser kryptographischen Systeme.

2.2 Asymmetrische Kryptographie

Die asymmetrische Kryptographie wurde Mitte 1970 von Ralph Merkle sowie von Diffie und Hellmann entwickelt. Sie beruht auf der Idee zur Kommunikation zwischen zwei Instanzen ein Schlüsselpaar zu verwenden. Dieses Schlüsselpaar besteht aus dem privaten und dem öffentlichen Schlüssel. Zur Kommunikation muss im Vorhinein ein gegenseitiger Austausch des öffentlichen Schlüssels erfolgt sein, denn die zu schickende Nachricht ist vom Sender mit dem öffentlichen Schlüssel des Empfängers in einen Kryptotext um zu wandeln. Nach Empfang entschlüsselt der Empfänger nun die Nachricht mit seinem eigenen privaten Schüssel. Im Falle einer Antwort verschlüsselt der ehemalige Empfänger (nun Sender) seine Nachricht wieder mit dem öffentlichen Schlüssels seines Gegenüber. Welcher die Entschlüsselung erneut mit dem eigenen privaten Schlüssel durchführen muss. Somit ist während der Kommunikation kein erneuter Austausch eines sicheren Schlüssels notwendig und auch erneute Kommunikationen können mit den bereits vorhandenen Schlüsseln durchgeführt werden. Wichtig ist hierbei jedoch, dass die Sicherheit des privaten Schlüssels und die Authentizität des öffentlichen Schlüssels gewährleistet ist. Mehr dazu in den folgenden Kapiteln.

2.3 Grundlage asymmetrischer Systeme

Im obigen Szenario hat man gesehen wie ein grober Kommunikationsablauf zwischen zwei Parteien aussieht. Dieses wird in folgender Grafik (nach [1]) veranschaulicht:

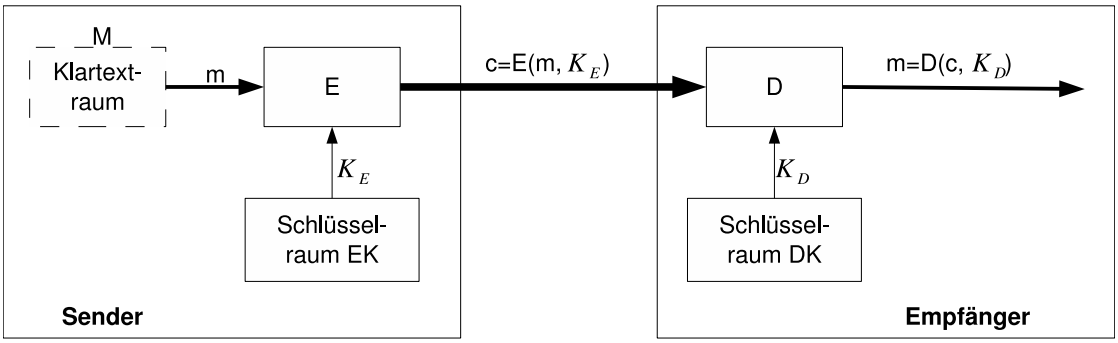

Abbildung 1: Komponenten eines Kryptosystems

1. Tupel
   (M, C, EK, DK, E, D)
2. zwei endliche Alphabete
   (A_1, A_2)
3. Menge von Klartexten
   $(M \subseteq A_1^* \backslash \emptyset)$
4. Klartext
   $(m \in M)$
5. Menge von Kryptotexten
   $(C \subseteq A_2^* \backslash \emptyset)$
6. Kryptotext
   $(c \in C)$
7. Verschlüsselungsschlüsselraum
   $(EK \backslash \emptyset)$
8. Entschlüsselungsschlüsselraum
   $(DK / \emptyset$
   $mit\ f : EK \rightarrow DK$
   $und\ f(K_E) = K_D)$
9. Verschlüsselungsverfahren
   $(E :\ MxEK \rightarrow C)$
10. Entschlüsselungsverfahren
    $(D :\ CxDK \rightarrow M)$
11. Es gilt:
    $\forall m \in M : D(E(m, K_E), K_D) = m$

Die Eigenschaft (11.) der obigen Legende zeigt uns das Zusammenspiel unserer einzelnen Komponenten. Wird ein Klartext m mit einem Schlüssel $K_E \in EK$ durch ein kryptographisches Verfahren E in einen Kryptotext umgewandelt. So ist gegeben, dass dieser Kryptotext unter Verwendung des zu K_E gehörigen Schlüssels $K_D \in DK$, und des kryptographischen Verfahrens D wieder in den ursprünglichen Klartext m umgewandelt werden kann. Damit dies gegeben ist, muss unser asymmetrisches Kryptosystem einige wichtige Punkte erfüllen. Es muss (nach [1]):

1. eine effiziente Möglichkeit zur Erzeugung von Schlüsselpaaren (K_E, K_D) geben.
2. garantiert sein, dass der Private (K_D) nicht effizient aus dem öffentlichen Schlüssel (K_E) gebildet werden kann.
3. möglich sein effizient zu Ver- und Entschlüsseln.
4. optional aber nicht notwendiger Weise die Möglichkeit bestehen mit beiden Schlüsselpaaren zu verschlüsseln und mit dem jeweils anderem das Resultat zu entschlüsseln. Somit ist das System für Certifikate geiegnet.

Um dies zu erreichen nutzt man Einwegfunktionen.

Einwegfunktionen sind besondere Funktionen $(f : X \to Y)$ bei denen das Urbild (X) nicht unter vertretbarem zeitlichem Aufwand aus dem Bild (Y) zu berechnen ist. Mathematische Probleme, welche derartige Funktionen bilden, sind unter Anderem das Faktorisierungsproblem und der diskrete Algorithmus. Es ist nicht bewiesen, dass die Umkehrung nicht möglich ist, jedoch übersteigt die Komplexität der Berechnung unser heutiges Vermögen diese in einer vertretbaren Zeitspanne zu lösen.

Bei der Verwendung von normalen Einwegfunktionen in einem Kryptosystem wäre zwar die Sicherheit der Daten garantiert, jedoch stellt die Umkehrung selbst für autorisierte Personen ein unüberwindbares Hindernis dar. Man benötigt zum Entschlüsseln unserer Daten eine Hintertür - die so genannte Falltür. Einwegfunktionen mit Falltür bieten eine identische Sicherheit wie normale Einwegfunktionen, und die Option verschlüsselte Daten unter Kenntnis eines Geheimnisses (bei und der private oder öffentliche Schlüssel) zu entschlüsseln. Beispiele für mathematische Probleme mit Falltür sind die h-te Potenz modulo(n) und der zusammengesetzte Modul(n) ([1]).

2.4 RSA

Bei RSA handelt es sich um ein 1978 von Ronald Rivest, Adi Shamir und Leonard Adleman entwickeltem Verfahren ([1]), welches alle Vier von uns genannten Bedingungen für ein asymmetrisches Kryptosystem erfüllt. Es beruht auf dem Faktorisierungsproblem.

2.4.1 Schlüsselberechnung

Zur Berechnung unseres Schlüsselpaares mit dem öffentlich Schlüssel (e, n) und dem privaten Schlüssel $(d, \varphi(n))$ geht man folgenden Weg:

1. Wähle zwei Primzahlen $p \neq q$.
2. Ermittle $n = p \cdot q$ und $\varphi(n) = (p-1)(q-1)$.
3. Wähle eine Zahl e für die gilt: $ggt\Big(e, \varphi(n)\Big)$ und $1 < e < \varphi(n)$

4. Als nächstes berechne d mit $ed = 1 mod \varphi(n)$.

5. (n, d) ist privater Schlüssel, (n, e) ist öffentlicher Schlüssel

[1]

2.4.2 Verschlüsselung und Entschlüsselung

Möchte man nun einen Klartext in einen Kryptotext umwandeln hat man zwei Schritte zu befolgen. Als erstes muss der Klatext (m) in eine Folge von Zahlen mit $m_i \in \{0, 1, ..., n-1\}$ umgewandelt werden. Daraufhin wird für alle $m_i \quad c_i = m_i^e mod n$ mit Hilfe des vorher berechneten öffentlichen Schlüssels bestimmt und man erhält den Kryptotext c. Die Umkehrung erfolgt ähnlich. Jedoch nutzt man dieses mal den privaten Schlüssel (d, n). Zuerst wird der Kryptotext in die Zahlenfolge umgewandelt indem man $m_i = c_i^d mod n$ für alle c_i bestimmen. Im Anschluss muss lediglich die Zahlenfolge wieder in unseren Klartext transformiert werden.

Die Möglichkeit dieses Verfahren auch invers an zu wenden, also das Verschlüsseln mit dem privaten und das Entschlüsseln mit dem öffentlichen Schlüssel, wird bei Zertifikaten und E-Mail-Signaturen in Anspruch genommen. Bei letzteren wird aus dem Inhalt der E-Mail ein Hashcode generiert, welcher mit dem privaten Schlüssel verschlüsselt, als Signatur an die Nachricht angehangen und dann verschickt wird. Dabei geht es weniger um das Geheimhalten des Inhaltes der Nachricht. Sondern um das Sicherstellen das sie unverfälscht beim Empfänger eintrifft. Nach Erhalt der E-Mail kann der Empfänger die Signatur mit dem öffentlich Schlüssel des Senders entschlüsseln und den Inhalt der Nachricht mit dem Hashcode abgleichen. Sollte es dabei zu Unstimmigkeiten kommen ist bewiesen das die Nachricht nach anhängen der Signatur verändert wurde.

2.5 Schlüsselmanagement

Bei der asymmetrischen Kryptographie ergeben sich aufgrund ihrer besonderen Grundidee der zwei Schlüssel auch ganz eigene Möglichkeiten und Pflichten zur Erzeugung, Aufbewahrung und möglichen Wiederherstellung dieser.

Die Erzeugung asymmetrischen Schlüsselmaterials kann der Nutzer entweder selbst übernehmen oder einer zentralen Instanz (z.B. CA) überlassen. Sollte der Nutzer den Schlüssel selbst generieren, hat dies zum Vorteil, dass er die volle Kontrolle über den Erstellungsprozess besitzt. Jedoch obliegt ihm somit auch die Verantwortung den Prozess und die fertigen Schlüssel durch nötige Einstellungen im System und nötige Vorsicht zu sichern. Besonderes Augenmerk liegt dabei auf dem privaten Schlüssel, welcher unter keinen Umständen öffentlich werden darf. Da die Sicherheit der verschlüsselten Daten dann nicht mehr gewährleistet ist. Im Falle der Erzeugung durch eine zentrale Instanz gibt der Nutzer diese Verantwortung größtenteils aus der Hand.

Dabei sollte er sich jedoch bewusst sein, dass so auch keine Kontrolle seinerseits besteht. Des Weiteren ist es nun notwendig den privaten Schlüssel sicher dem Nutzer zur Verfügung zu stellen, was unter Umständen kritisch werden kann. Ist der Schlüssel erstmal erzeugt, muss dieser zwingend sicher aufbewahrt werden.

Die Schlüsselspeicherung muss gewährleisten, dass der private Schlüssel keinem Unbefugten zur Verfügung steht. Um ein Fehlverhalten des Nutzers auszuschließen, wäre es also ratsam nicht einmal diesem den Zugang zu dem privaten Schlüssel zu gestatten. Diesen Weg gehen Token (oder SmartCard). Token sind Chipkarten oder USBsticks bei denen der Schlüssel auf einem internen Chip sicher verwahrt wird. Der Zugriff ist ausschließlich über Sicherheitssoftware in Emailprogrammen (PGP) oder anderen (OpenSSH) möglich, welche den Schlüssel selbstständig auslesen und ihn nicht zwischenspeichern. Außerdem ist es natürlich möglich den Schlüssel selbst sicher zu verwahren.

Die Verbreitung des öffentlich Schlüssels ist über beliebige Wege möglich. Zum Einen besteht die Möglichkeit den öffentlichen Schlüssel auf einem KeyServer zu lagern oder ihn per E-Mail zu verschicken. Es gibt noch wesentlich mehr Alternativen zur Verbreitung, welche hier nicht alle genannt werden sollen. Allerdings ist die Authentizität des öffentlich Schlüssels zu berücksichtigen.Denn nicht immer lässt sich garantieren das der Schlüssel den man als öffentlichen Schlüssel von Person A gefunden oder erhalten hat auch wirklich von dieser stammt. Zu diesem Zweck werden Schlüssel von Gesprächspartnern, welche sich bereits gegenseitig überprüft haben, unterschrieben. Somit bestätigt der Signierende die Zugehörigkeit des öffentlichen Schlüssels zu Person A. Es wird eine mit dem privaten Schlüssel des Signierenden verschlüsselte Signatur an den öffentlichen Schlüssel von Person A angehangen. Aufgrund dessen unsere Operationen umkehrbar sind. Lässt sich nun diese Signatur mit dem öffentlichen Schlüssel des Unterzeichners öffnen und überprüfen. Da die Signatur jedoch nicht geändert werden kann, ist sie zudem vor Fälschung sicher. Dieses Verfahren wird ebenso bei Certifikaten im WebOfTrust oder wie bereits erwähnt beim E-Mail Verkehr genutzt.

2.6 Hybride Kryptographie

Die hybride Kryptographie ist die Kombination von asymmetrischer und symmetrischer Kryptographie. Eine symmetrische Verschlüsselung ist bei gleichem Sicherheitsgewinn im Gegensatz zu einer vergleichbaren asymmetrischen Verschlüsselung wesentlich effizienter zu berechnen. Allerdings muss je Kommunikationsbeginn ein erneuter aufwendiger Schlüsselaustausch (z.B. über Diffie-Hellmann) erfolgen, auch wenn mit dem Partner bereits in der Vergangenheit kommuniziert wurde. Die Lösung dafür stellt die hybride Kryptographie dar. Davon ausgehend, es steht uns eine längere Konversation mit Person A bevor. Nun soll nicht jede einzelne Nachricht mit einem asymmetrischen System verschlüsseln, denn je nach Größe der Nachricht kann dies

sehr Rechenintensiv werden. Um einen derartigen Aufwand zu umgehen, legen wir einen symmetrischen Schlüssel fest, verschlüsseln diesen mit dem öffentlichen Schlüssel von Person A und übermitteln ihm diesen sicher verschlüsselt. Person A verfügt nun, nach der Entschlüsselung, ebenso wie wir über den symmetrischen Schlüssel, über den nun die Nachrichten des Gespräches wesentlich effizienter verschlüsselt werden können.

Nicht immer wird bei der hybriden Kryptographie ein symmetrischer Schlüssel von einem Gesprächspartner bestimmt. Es existieren viele unterschiedliche Implementierungen auf die hier aber nicht eingegangen werden soll.

3 Kryptographie mit Java

3.1 Java Cryptography Architecture

Die Java Cryptography Architecture ist neben BouncyCastle ein Provider (Anbieter) für kryptographische Systeme der Programmiersprache Java. Er dient zur Implementierung von:

1. digitalen Signaturen,
2. Hash's,
3. Zertifikaten (und deren Überprüfung),
4. Verschlüsselungen,
5. Schlüsselerzeugung und Management,
6. Zufallszahlen,
7. und vieles mehr [2]

Außerdem beinhaltet das JCA noch andere Provider. In den folgenden Kapiteln werden nun die Punkte vier bis sechs genauer betrachtet.

3.2 Zufallszahlen

Das JCA bietet zur Generierung von Zufallszahlen einen implementierten, nicht quelloffenen Zufallszahlengenerator, welcher in die Klasse *SecureRandom* eingebettet ist. Mittels des in Abbildung 2 gezeigten Aufrufes kann ein solches Zufallszahlen-Objekt generiert und später zur Erzeugung von Schlüsseln Verwendet werden.

Der Generator befindet sich im Provider SUN und trägt die Bezeichnung $SHA1PRNG$. Er funktioniert laut Angaben von Oracle wie folgt:

Abbildung 2: Codebeispiel SecureRandom

```
SecureRandom random =
        SecureRandom.getInstance("SHA1PRNG", "SUN");
```

> "This algorithm uses SHA-1 as the foundation of the PRNG. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used." [3]

Da es sich hierbei um den einzigen implementierten Zufallszahlengenerator im JCA handelt, ist es auch die Standard-Auswahl wenn keine alternative bereitgestellt wurde.

3.3 Schlüsselgenerierung

Wie in Abbildung 3 erkennbar nutzt das JCA die Klassen *KeyPairGenerator* und *KeyPair*, um das für uns wichtige asymmetrischen Schlüsselpaar zu erzeugen. Als erstes wird hierbei ein *KeyPairGenerator* mit der *KeyPairGenerator.getInstance()* Methode erzeugt. Diese Methode erwartet als Parameter das zu verwendende Verfahen. Neben RSA sind ebenfalls DSA,AES und einige weitere möglich. Im nächsten Schritt wird unser Objekt mit der von uns gewünschten Schlüsselgröße initialisiert. Sollte wie hier gezeigt keine Übergabe eines SecureRandom Objektes als zweiter Parameter erfolgen nutzt Java automatisch den Zufallszahlengenerator des Provider mit der höchsten Priorität. Zum Schluss erzeugt man mittels *keyGen.generateKeyPair()* die beiden Schlüssel,

Abbildung 3: Codebeispiel KeyPairGenerator

```
KeyPairGenerator keyGen =
        KeyPairGenerator.getInstance( "RSA" );
keyGen.initialize( int keysize );
KeyPair keys = keyGen.generateKeyPair();
PrivateKey privKey = keys.getPrivate();
PublicKey pubKey = keys.getPublic();
```

welche nun in einem *KeyPair* Objekt gespeichert sind. Möchte man die Schlüssel aus dem *KeyPair* extrahieren, ist dies über *keys.getPrivate()* oder *keys.getPublic()* jederzeit möglich.

3.4 Schlüsselspeicherung

Java bietet zum Speichern und Verwalten von Schlüsseln eine spezielle Klasse namens "*Keystore*" an. Diese repräsentiert eine Datenbank bestehend aus Schlüsseln und Zerti-

fikaten, welche im Speicher gehalten werden. Der Zugriff auf die Einträge im *Keystore* erfolgt aus Sicherheitsgründen über Aliase hinter denen die sensiblen Daten verborgen sind. Erst beim Ausführen der mitgelieferten Methode:
$final void store(OutputStream stream, char[] password)$, wird der Keystore auf die Festplatte geschrieben.
Methoden des Keystore sind unter anderem:

- final void setKeyEntry(String alias,Key key, char[] password, Certificate[] chain)
- final boolean isKeyEntry(String alias)
- final boolean isCertificateEntry(String alias)
- final void deleteEntry(String alias)
- final Key getKey(String alias, char[] password)

Weitere Informationen über den *Keystore* können der JCA Documentation entnommen werden: [2] . Neben dem Keystore ist es ebenfalls möglich die Schlüssel per Serialisierung zu speichern. Dies ist jedoch nicht empfehlenswert, da somit unter Umständen der Zugriff auf die Schlüssel ungeschützt ist.

3.5 Schlüsseleinigung

Im Falle der Implementierung einer Kommunikation über hybride Kryptographie ist nach der Erstellung des Schlüsselpaares eine Schlüsseleinigung für den symmetrischen Schlüssel notwendig. Die folgende Grafik veranschaulicht dieses Vorgehen:

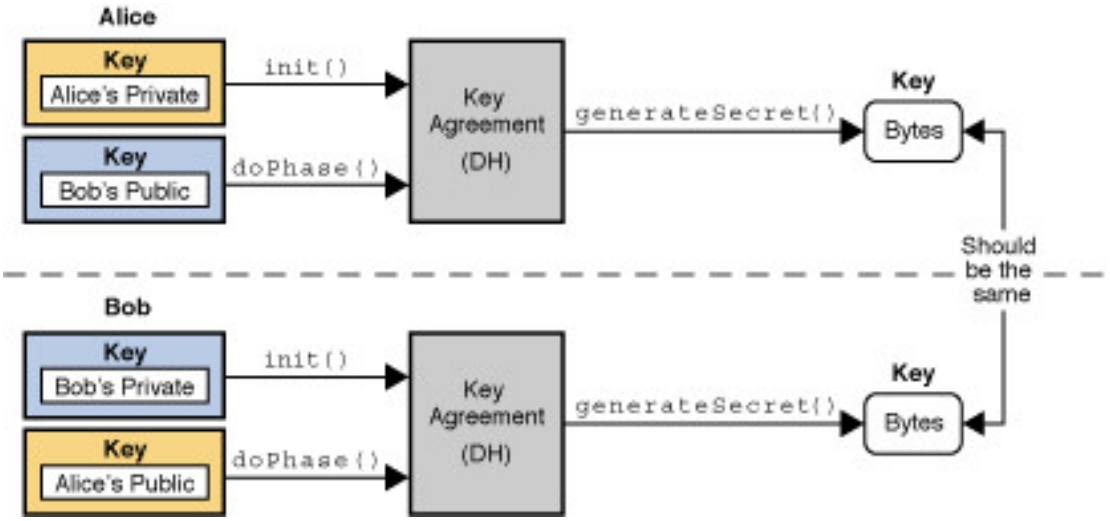

Abbildung 4: KeyAgreement

Wie in Abbildung 4 und 5 zu erkennen, initialisiert jeder Teilnehmer ein *KeyAgreement*

mit seinem privaten Schlüssel, welches von dem Gesprächspartner mit einem *doPhase*() und dem öffentlich Schlüssel des jeweils Anderen bestätigt wird. Im Anschluss generiert sich jeder Teilnehmer den symmetrischen Schlüssel, auf Grundlage der beim *KeyAgreement* gesammelten Daten. Dieses Vorgehen wird bei den Providern BouncyCastle oder JCA fast ausschließlich über Diffie-Hellman unterstützt und ist nicht abhängig von der Anzahl der Teilnehmer. Somit ist ebenfalls ein *KeyAgreement* mit mehr als zwei Parteien möglich.

Abbildung 5: 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");
```

3.6 Ver- und Entschlüsselung

Im Folgenden soll gezeigt werden, wie mit Hilfe der *Cipher* Klasse in Java ein Klartext verschlüsselt oder ein Kryptotext entschlüsselt werden kann. Obiger Program-

Abbildung 6: Codebeispiel Cipher

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

mauszug veranschaulicht die Implementierung einiger Methoden, welche zur Verschlüsselung notwendig sind. Als erstes wird eine Instanz des Cipher mit Hilfe der *Cipher.getInstance*() Methode gebildet. In Abbildung 6 ist die Angabe von RSA als Algorithmus und von BC (BouncyCastle) als CryptoProvider erkenntlich. Sollte nur der gewünschte Algorithmus angegeben werden, wählt Java den Standard-Provider (JCA). Im nächsten Schritt muss der Cipher im gewünschten Modus und mit dem passenden Schlüssel initialisiert werden.
Mögliche Modi sind:

- *ENCRYPT_MODE*
  (zur Verschlüsselung von Klartexten)
- *DECRYPT_MODE*
  (zur Entschlüsselung von Klartexten)
- *WRAP_MODE* (zum Umwandeln eines Schlüssels in Bytecode)
- *UNWRAP_MODE* (Umkehrung von *WRAP_MODE*)

Sollte es der Fall sein, dass der Umfang der Nachricht nicht bekannt oder zu groß für den verfügbaren Arbeitsspeicher ist,empfiehlt es sich wie im obigen Beispiel erst die *cipher.update*() und danach dir *cipher.doFinal*() Methode zu verwenden. Diese Aufteilung ermöglicht es den Klartext (message) nach und nach in Kryptotext umzuwandeln. Bei der alleinigen Verwendung von *cipher.doFinal*() wird die gesamte Nachricht mit einem Mal transformiert, was zum überlaufen des Speichers oder Verlust der Nachricht führen kann. Bei einer Umwandlung von Kryptotext in Klartext geht man dabei ähnlich vor. Lediglich der Modus des Cipher und der verwendete Schlüssel müssen hier angepasst werden.

4 Zusammenfassung

Die hier dargestellte Ausarbeitung thematisiert den Bereich der asymmetrischen Kryptographie in Java. Dem Leser soll ein Überblick verschafft werden wie deren Grundlagen aussehen und man sie später anwendet. Der erste Teil dieser Arbeit spezifiziert die theoretischen Grundlagen und geht im zweiten auf die Implementierung ein. Um dem Leser einen Einstieg in das Thema zu erleichtern werden zu Beginn wichtige Ziele und Eigenschaften der asymmetrischen Kryptographie dargestellt und danach mathematische Grundlagen und Funktionsweisen sowie grundlegende Abläufe näher erläutert. Nach diesem Einstieg wird mit RSA ein Vertreter der asymmetrischen Kryptographie näher erläutert. Des weiteren werden Vorgehensweisen beschrieben, welche den Leser auf den Umgang mit Verschlüsselungen vorbereiten und sensibilisieren sollen. Im letzten Teil der Grundlagenschaffung wird die hybride Kryptographie als ein besonderer Bereich der Kryptographie vorgestellt. Im praktischen Teil wird dargelegt welche Möglichkeiten die Programmiersprache Java bietet um im Bereich der asymmetrischen Kryptographie zu agieren. Hierzu werden zunächst Implementationen wie

der Kryptoprovider JCA in Java Vorgestellt auf dessen Grundlage wichtige Vorgänge wie die Schlüsselerzeugung, das Schlüsselmanagement oder das Ver- und Entschlüsseln von Nachrichten erläutert wird.

Literatur

[1] Claudia Eckert. *IT-Sicherheit : Konzepte - Verfahren - Protokolle*. Oldenburg, 2013.

[2] Oracle. Java Cryptography Architecture (JCA) Reference Guide. `http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`, 2011. [Online; accessed 17-January-2014].

[3] Oracle. Java Cryptography Architecture Standard Algirthm Name Documentation. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#SecureRandom`, 2014. [Online; accessed 17-January-2014].