

# Sichere Programmierung

## Projekt 4

David Pierre Sugar  
(76050)  
Julian Sobott  
(76511)

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Aufgabe 1: Kryptographische Hashfunktion</b>	<b>2</b>
2.1	Hashing . . . . .	2
2.2	Implementierung . . . . .	2
2.3	Beispiel . . . . .	3
<b>3</b>	<b>Aufgabe 2: Symmetrische Verschlüsselung</b>	<b>3</b>
3.1	Cipher . . . . .	3
3.2	Implementierung . . . . .	3
3.2.1	Key erstellen . . . . .	3
3.2.2	Initialisierungs-Vektor . . . . .	4
3.2.3	Cipher . . . . .	4
3.3	Verschlüsselung . . . . .	4
3.3.1	Ausgabe Cipher Text . . . . .	4
3.3.2	Entschlüsselung . . . . .	5
3.3.3	Ausgabe Plain Text . . . . .	5
3.4	Beispielausgabe . . . . .	5
<b>4</b>	<b>Aufgabe 3 Asymmetrische Verschlüsselung</b>	<b>5</b>
4.1	Asymmetrische Verschlüsselung . . . . .	5
4.2	Implementierung . . . . .	6
<b>5</b>	<b>Aufgabe 4: Digitale Signatur</b>	<b>7</b>

## 1 Einleitung

In diesem Praktikum geht es darum Konzepte, die in der Vorlesung *Einführung in die IT-Sicherheit* erklärt wurden, in Java zu implementieren. Hierzu wird die Kryptographie-API von Java verwendet.

## 2 Aufgabe 1: Kryptographische Hashfunktion

### 2.1 Hashing

Eine Hashfunktion wandelt eine beliebige Eingabe in eine Ausgabe fester Länge um. Das Ergebnis wird als *Hash* oder auch engl. *digest* bezeichnet. Jede Hashfunktion, sollte dabei aber folgende Eigenschaften haben:

- **Einwegfunktion:** Das heißt der original Wert soll nicht aus dem Hash zurück gerechnet werden können.
- **Kollisionsresistenz:** Da der Wertebereich der Funktion kleiner als der Definitionsbereich, kann es zu Kollisionen kommen. Das heißt unterschiedliche Eingabe Werte erzeugen den gleichen Ausgabewert. Eine Hashfunktion sollte möglichst solche Kollisionen vermeiden.
- **Große Streuung:** Hashes von ähnlichen Werten sollen weit auseinander liegen.
- **Effiziente Berechnung:** Das Hashen einer Eingabe sollte möglichst effizient sein. Aufgrund der Einwegfunktion, ist das berechnen der Eingabe aufgrund eines Hashes, bei modernen Hashfunktionen aber nicht effizient lösbar.

Hashes können unter anderem dafür verwendet werden, um Prüfsummen (engl. checksums) zu berechnen. Diese Können dann dafür verwendet um die Integrität einer Nachricht zu bewahren. Das heißt es kann kontrolliert werden, dass die Nachricht nicht verändert wurde.

### 2.2 Implementierung

In dieser Aufgabe soll die Erzeugung und Verifikation einer solchen Prüfsumme implementiert werden. Als Hashalgorithmus soll hier SHA-256 verwendet werden. Für die Aufgabe wurden zwei statische Methoden implementiert. `byte[] createChecksum(String message)` und `boolean verifyChecksum(byte[] checksum, String message)`. Die erste erzeugt eine Prüfsumme und die zweite verifiziert ob ein String dieselbe Prüfsumme erzeugt.

Fürs Erzeugen wird eine `MessageDigest` Instanz, welche mit SHA-256 arbeitet, geholt. Mit dieser wird dann, der String in einen entsprechenden Hash Wert umgewandelt.

```
1 MessageDigest md = MessageDigest.getInstance("SHA-256");
2 return md.digest(message.getBytes());
```

Fürs Verifizieren, wird auf dieselbe Weise von dem neuen String die Checksum erstellt. Ob die beiden gleich sind, wird mit der statischen Methode `MessageDigest.isEqual()` überprüft.

```
1 byte[] actual = createChecksum(message);
2 return MessageDigest.isEqual(actual, checksum);
```

## 2.3 Beispiel

Zur demonstrierung der beiden Methoden, wurden in der `main` Methode 2 Strings erstellt. Die Verifikation gibt `true` zurück, wenn beide Strings gleich sind und `false`, wenn sie unterschiedlich sind. Führt man das Programm erhält man die folgende Ausgabe

```
1 Checksums are equal: true
2 Checksums are equal: false
```

Das ganze könnte mithilfe von Verschlüsselung zum Beispiel verwendet werden, um Nachrichten zu versenden deren Integrität gewahrt werden muss.

## 3 Aufgabe 2: Symmetrische Verschlüsselung

Symmetrische Verschlüsselung ist ein Weg, Nachrichten vor unerwünschten Einblicken zu schützen. Symmetrisch heißt dabei, dass der selbe Key zum ver-/ und wieder entschlüsseln verwendet wird. Ein Beispiel hierfür wäre der **AES** (Advanced Encryption Standard) Algorithmus, der auch **Rijndael** Algorithmus genannt wird, in Anlehnung an die Erfinder Vincent Rijmen und Joan Daemen. Symmetrische Verschlüsselungen bieten, einen angemessenen großen Key vorausgesetzt, guten Schutz und eine schnelle Ver-/ Entschlüsselung, jedoch muss zur sicheren Übermittlung auf andere Verschlüsselungsverfahren wie RSA zurückgegriffen werden, da diese sonst leicht von Dritten abgegriffen werden können.

### 3.1 Cipher

Der AES ist ein Block Cipher, d.h. es werden jeweils Blöcke zu je 128 Bit verschlüsselt, dabei werden die **Key-Längen** 128, 192 und 256 Bit unterstützt. Die Länge des Keys gibt dabei die Anzahl an Transformationsrunden vor, die für die Verschlüsselung durchlaufen werden. Um den Text wieder zu entschlüsseln, werden die Runden rückwärts durchlaufen. Die Transformationen finden dabei auf einer 4x4 Byte Matrix statt.

### 3.2 Implementierung

Das Beispiel besteht darin, einen String zu verschlüsseln, den Cipher Text auf Standard Out auszugeben, den Cipher Text wieder zu entschlüsseln und auch den entschlüsselten Text auf der Kommandozeile auszugeben. Sollte schlussendlich der anfangs verschlüsselte Text richtig ausgegeben werden, so kann davon ausgegangen werden, dass die symmetrische Verschlüsselung mittels AES richtig ausgeführt wurde.

#### 3.2.1 Key erstellen

Als erstes wird ein zufälliger Key erstellt, der dann zusammen mit dem AES Cipher verwendet werden soll. Dafür wird die Methode `getAESKey()` aufgerufen, die einen **AES Key Generator** instanziert und danach mittels `init()` diesen Key Generator mit einer Key Größe von 256 initialisiert. Danach wird mit `generateKey()` der zufällige Key erstellt und zurück gegeben.

```
1 SecretKey key = getAESKey();
```

### 3.2.2 Initialisierungs-Vektor

Ein Initialisierungsvektor ist eine Zufallszahl, die als weitere Sicherheit in die Verschlüsselung mit eingebracht wird. Um das Brechen des Ciphers, z.B. mittels Dictionary Attack, zu erschweren wird der vorrausgegangene Cipher Block mit genutzt, um den derzeitigen Plaintext Block zu verschlüsseln. Da für den ersten Plaintext Block noch kein Cipher Block zur Verfügung steht, der verwendet werden kann wird eine Zufallszahl, der IV, zur Verschlüsselung hinzugezogen.

Dieser IV wird von `getIvSpec()` zurückgegeben.

```
1 IvParameterSpec ivSpec = getIvSpec();
```

Dabei nutzt die Methode einen Pseudo-Zufallszahl-Generator um ein Byte Array zu befüllen. Dieses Array wird dann als Argument genutzt, um mit `IvParameterSpec()` ein IV Objekt zu erzeugen.

### 3.2.3 Cipher

Um den Cipher zu initialisieren, wurde die Methode `initCipher()` implementiert. Dies erhält `Key` und `IV`, sowie den Modus (`ENCRYPT`, `DECRYPT`) als Parameter und instanziert ein AES Cipher Objekt, das dann zur Verschlüsselung verwendet werden kann.

```
1 Cipher cipher = initCipher(key, Cipher.ENCRYPT_MODE, ivSpec);
```

## 3.3 Verschlüsselung

Mittels `encrypt()` kann nun der Plain Text verschlüsselt werden. Dazu werden `Cipher` sowie `Text` and die Methode übergeben, zurück kommt der Cipher Text als Byte Array. Intern wird einfach nur `doFinal()` auf dem Cipher Objekt aufgerufen.

```
1 byte cipherText[] = encrypt(cipher, pt);
```

### 3.3.1 Ausgabe Cipher Text

Die Methode `printCipherText()` erhält das Byte Array als Argument und gibt es in Base 64 codiert auf der Kommandozeile aus.

```
1 printCipherText(cipherText);
```

### 3.3.2 Entschlüsselung

Nun muss der Text wieder entschlüsselt werden. Dazu wird das Cipher Objekt nun mit dem DECRYPT Mode initialisiert, dabei bleiben Key und IV die selben. Danach wird `decrypt()` aufgerufen. Die Methode ruft erneut `doFinal()` auf und entschlüsselt damit den übergebenen Cipher Text, zurück in den Ausgangstext.

```
1 cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);
2 String new_pt = decrypt(cipher, cipherText);
```

### 3.3.3 Ausgabe Plain Text

Schlussendlich wird nun auch der entschlüsselte Plain Text auf der Kommandozeile ausgegeben.

```
1 System.out.println("Plain Text: " + new_pt);
```

## 3.4 Beispielausgabe

```
1 [sugar@jellyfish Code]$ java SymEncrypt
2 Cipher Text: 1PSGD5vFeGS7K1TBZjX2RQ==
3 Plain Text: Neo... Wake up!
4 [sugar@jellyfish Code]$ java SymEncrypt
5 Cipher Text: 0G5fVtzaOu6P5RybQLLMGA==
6 Plain Text: Neo... Wake up!
7 [sugar@jellyfish Code]$ java SymEncrypt
8 Cipher Text: +wCZzQ9bfxIA9r1aGFjB8Q==
9 Plain Text: Neo... Wake up!
```

## 4 Aufgabe 3 Asymmetrische Verschlüsselung

### 4.1 Asymmetrische Verschlüsselung

Eine Asymmetrische Verschlüsselung wird dann verwendet, wenn der Schlüsselaustausch nicht geheim erfolgen kann. Es gibt ein Schlüsselpaar, welches aus einem **private key** und einem **public key** besteht. Der public key kann für jeden zugänglich sein. Der private key muss aber privat bleiben und darf nicht von einer zweiten Instanz gekannt werden. Für den Nachrichtenaustausch, erstellt zuerst eine Instanz, ein solches Schlüsselpaar. Der public key wird dann an den Kommunikationspartner weiter geschickt. Dies muss nicht verschlüsselt sein, da der public key kein Geheimnis enthält. Der Kommunikationspartner kann nun eine Nachricht mit dem public key verschlüsseln und zurück senden. Diese Nachricht kann nur mit dem passenden private key entschlüsselt werden. Dieses verfahren kann zum beispiel verwendet werden, um symmetrische Schlüssel auszutauschen. Asymmetrische Verschlüsselung ist im Vergleich zur symmetrischen langsamer und wird deshalb meist nur zu Beginn einer Kommunikation verwendet.

## 4.2 Implementierung

Diese Aufgabe soll die Nutzung des RSA Algorithmus für die asymmetrische Verschlüsselung zeigen. Ein Text soll mit einem public key verschlüsselt werden, und mit dem entsprechenden private key wieder entschlüsselt werden. Hierzu wird als erstes mithilfe eines `KeyPairGenerator` ein Schlüsselpaar für die Verschlüsselung mit RSA erstellt. Auf die beiden Schlüssel kann dann mithilfe von gettern zugegriffen werden.

```
1 // Schlüsselgenerierung
2 KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
3 keyPairGen.initialize(keysize);
4 keyPair rsaKeyPair = keyPairGen.generateKeyPair();
5
6 // private und public key
7 PublicKey publicKey = rsaKeyPair.getPublic();
8 PrivateKey privateKey = rsaKeyPair.getPrivate();
```

Als nächstes wird ein Objekt, welches für die Ver- und Entschlüsselung verantwortlich ist, mithilfe der factory Methode `getInstance` geholt. Dieses kann mehrfach verwendet werden, muss aber vor jeder Verwendung initialisiert werden. Hierbei muss übergeben werden, ob ver- oder entschlüsselt werden soll, und der jeweilige passende key.

```
1 Cipher rsa = Cipher.getInstance("RSA");
2
3 // Verschlüsseln
4 rsa.init(Cipher.ENCRYPT_MODE, publicKey);
5
6 // Entschlüsseln
7 rsa.init(Cipher.DECRYPT_MODE, privateKey);
```

Mit der `doFinal` Methode, kann dann der ver-/entschlüsselungs Prozess abgeschlossen werden. Hierbei wird immer ein `byte[]` zurückgegeben, welches den Text enthält.

```
1 // Verschlüsselter Text
2 byte[] cipherText = rsa.doFinal(plainText.getBytes());
3
4 // Entschlüsselter Text
5 byte[] plainTextAgain = rsa.doFinal(cipherText);
```

Um den verschlüsselten Text schöner auf der Konsole anzuzeigen, wird er noch in Base64 umgewandelt. Außerdem wird das `byte[]` an den Konstruktor der Klasse `String` übergeben, um den Text als zusammenhängenden Text und nicht als Array auszugeben.

```
1 Base64.Encoder encoder = Base64.getEncoder();
2 String message = encoder.encodeToString(cipherText);
3 System.out.println("encrypted message: " + message);

1 System.out.println("decrypted message : " + new String(plainTextAgain));
```

Eine Ausgabe kann dann Beispielsweise wie folgt aussehen:

```
1 \begin{lstlisting}
2 System.out.println("decrypted message : " + new String(plainTextAgain));
```

```
1 Original message: Kryptographie macht immer noch Spass!!!
2
3 encrypted message: GyHYxvw3nZiXt7LbqJAYPgn7idQKFriawHbA7
4 D6Qgq+eD0dHznt7dBc9+bTkqlCUvxvHUTZt4XPa+hYyKCfrdH/UI/NSm
5 9rxjRIjdy30cpBdA+6RqLga5jNIEPd9AJvhfLMUVbsHHY
6 +UQVoV4Fa5FX5TIEAfo4VQUXlvjmHngl5t6nNqdmjBqt4i8
7 gsr00ai2HYYJdowbkzj6lwu7sZPByCGibahXa011GQ/eQUX83Z2K
8 Kacazn/7KeG8HLhsNAUVp4TqJmRBEP0ZwN2/CC0dDvMhXjZdvhI7zYd
9 a0kXOf+hw0xbHZ2ahNdjPfMmYCLS40WfSK0sDiaLbA==
10
11 decrypted message : Kryptographie macht immer noch Spass!!!
```

## 5 Aufgabe 4: Digitale Signatur