Sichere Programmierung
Projekt 4

David Pierre Sugar
(76050)
Julian Sobott
(76511)

Inhaltsverzeichnis

1 Einleitung **2**

2 Aufgabe 1: Kryptographische Hashfunktion **2**
2.1 Hashing . . . 2
2.2 Implementierung . . . 2
2.3 Beispiel . . . 3

3 Aufgabe 2: Symmetrische Verschlüsselung **3**
3.1 Cipher . . . 3
3.2 Implementierung . . . 3
3.2.1 Key erstellen . . . 4
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

4 Aufgabe 3 Asymmetrische Verschlüsselung **5**
4.1 Asymmetrische Verschlüsselung . . . 5
4.2 Implementierung . . . 6

5 Aufgabe 4: Digitale Signatur **7**
5.1 Verfahren . . . 7
5.2 Implementierung . . . 7
5.3 Key-Pair generieren . . . 7
5.4 Text signieren . . . 8
5.5 Signatur verifizieren . . . 8
5.6 Beispielausgabe . . . 8

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.

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

2.3 Beispiel

Zur demosntration 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

```
Checksums are equal: true
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 endschlüsseln 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 angemessen großen Key vorrausgesetzt, 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 ausgeganen 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 erhällt `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()` erhällte 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: lPSGD5vFeGS7KlTBZjX2RQ==
3 Plain Text: Neo... Wake up!
4 [sugar@jellyfish Code]$ java SymEncrypt
5 Cipher Text: OG5fVtza0u6P5RybQLLMGA==
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 entsprechendem private key wieder entschlüssselt 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.

```
// Schlüsselgenerierung
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
keyPairGen.initialize(keysize);
keyPair rsaKeyPair = keyPairGen.generateKeyPair();

// private und public key
PublicKey publicKey = rsaKeyPair.getPublic();
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 mehrmal verwendet werden, muss aber vor jeder Verwendung initialisiert werden. Hierbei muss übergeben werden, ob ver- oder entschlüsselt werden soll, und der jeweilige passende key.

```
Cipher rsa = Cipher.getInstance("RSA");

// Verschlüsseln
rsa.init(Cipher.ENCRYPT_MODE, publicKey);

// Entschlüsseln
rsa.init(Cipher.DECRYPT_MODE, privateKey);
```

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

```
// Verschlüsselter Text
byte[] cipherText = rsa.doFinal(plainText.getBytes());

// Entschlüsselter Text
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.

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

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

Eine Ausgabe kann dann Beispielsweise wie folgt aussehen:

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

```
Original message: Kryptographie macht immer noch Spass!!!

encrypted message: GyHYxvw3nZiXt7LbqJAYPgn7idQKFRiawHbA7
D6Qgq+eDOdHznt7dBc9+bTkqlCUvxvHUTZt4XPa+hYyKCfrdH/UI/NSm
9rxjRIjdy3OcpBdA+6RqLga5jNIEPd9AJvhfLMUVbsHHY
+UQVoV4Fa5FX5TIEAfo4VQUXIvjmHngl5t6nNqdmjBqt4i8
gsrO0ai2HYYJdowbkzj6lwu7sZPByCGibahXaO11GQ/eQUX83Z2K
Kacazn/7KeG8HLhsNAUVp4TqJmRBEpOZwN2/CC0dDvMhXjZdvhI7zYd
aOkXOf+hw0xbHZ2ahNdjPfMmYCLS4OWfSK0sDiaLbA==

decrypted message : Kryptographie macht immer noch Spass!!!
```

5 Aufgabe 4: Digitale Signatur

Digitale Signaturen werden genutzt, um die Authentizität, Integrität, Überprüfbarkeit und Nicht-Abstreitbarkeit von digitalen Dokumenten sicher zu stellen, gewährleisten aber in keinem Fall die Vertraulichkeit eines Dokuments. Zur digitalen Signatur eignen sich dabei die meisten asymmetrischen Verschlüsselungssysteme, wie z.B. RSA. RSA ist dabei aber anfällig für Attacken, sollte der selbe Text mehrmals mit dem Selben Schlüssel signiert werden.

5.1 Verfahren

Anfangs wird eine zufälliges Schlüsselpaar, bestehend aus Public-Key und Private-Key, erstellt. Das gewünschte Dokument wird dann zuerst gehased und der Hash danach mittels RSA unter hinzunahme des Private-Key verschlüsselt. Danach kann der Hash an das Dukument mit angehängt werden. Um die Authentizität des Dokuments nun zu verifizieren kann jeder, der den Public-Key besitzt, den Hash entschlüsseln, selber nocheinmal das Dokument hashen und dann beide Hashes miteinander vergleichen. Stimmen diese überein, so kann davon ausgegangen werden, einen kollisionssicheren Hashing-Algorithmus vorrausgesetzt, dass das Dokument nicht verändert wurde und dass der Author tatsächlich die Person, mit dem zugehörigen Private-Key ist.

5.2 Implementierung

Das Beispiel besteht darin, einen Text mittels RSA digital zu signieren. Die Signatur wird dann auf der Konsole ausgegeben. Danach wird sie verifiziert, um zu überprüfen ob das Dokument authentisch ist und tatsächlich vom Absender stammt. Das Ergebnis wird ebenfalls wieder auf der Kommandozeile ausgegeben. Erwarte wird, dass die Prüfsummen beide gleich sind, d.h. dass verify() `true` zurückliefert.

5.3 Key-Pair generieren

Der Code zur generierung des Schlüsselpaares wurde in der Methode `genKeyPair()` gekapselt. Diese generiert mittels `KeyPairGenerator` einen RSA Private- und Bublic-Key

mit Schlüsselgröße 20000 und gibt das Paar als Rückgabewert zurück.

```
KeyPair kp = genKeyPair();
```

5.4 Text signieren

Das Signieren eines beliebigen Textes geschieht mittels `sign()`. Die Methode erhält den Text als String, sowie den privaten Schlüssel als Argumente und liefert eine Signatur des Textes zurück.

```
byte[] signature = sign(pt, kp.getPrivate());
```

Die Methode selber instanziert als erstes eine SHA256, RSA Instanz von `Signature` (1). Diese wird dann mit dem privaten Schlüssel initialisiert (2). Danach wird mittels `update()` der zu signierende Text, als Byte Array übergeben (3). Schlussendlich wird der Text dann signiert (4).

```
Signature rsa = Signature.getInstance("SHA256withRSA");
rsa.initSign(pk); // initialize this object fo signing
rsa.update(pt.getBytes()); // update the data to be signed
byte signature[] = rsa.sign();  // sign data
return signature;
```

5.5 Signatur verifizieren

Um die Authentizität und Integrität des Dukuments zu überprüfen, kann nun die `verify()` methode zusammen mit der Signatur und dem Public-Key aufgerufen werden.

```
verify(pt, signature, kp.getPublic())
```

Als erstes wird wieder eine SHA256, RSA Instanz von Signature erzeugt (1). Diesmal wird die Instanz jedoch mit dem Public-Key initialisiert (2). Danach wird der Text mittels `update()` an die Instanz zur Verifikation übergeben (3). Schlussendlich wird das Dokument unter Verwendung der Signatur verifiziert (4). Sollte Verifikation erfolgreich sein wird `true` zurück gegeben, andernfalls `false`.

```
Signature sigInst = Signature.getInstance("SHA256withRSA");
sigInst.initVerify(pk); // initialize this object for verification
sigInst.update(pt.getBytes()); // updates the data to be verified
return sigInst.verify(signature);
```

5.6 Beispielausgabe

```
$ java SignatureTest
Text: Follow the white rabbit
Signature: wtTJTF8X9UMrSRsWl44M8t6ui1aHTw+7EGOLgM
+fxjWRdVyLofLyq25VWNGSTlQ98A9Pdg4ywRvfaOjtPgUEC1T
OyUfk4hfzK+S8TmplYwVAlIGdInDvGB1u/lV751RKe/8JcCwv
```

```
kYHwsjBh0JB06OkDyRLip07STl2mPCYTS4UqJ+h7Dk7Cqh239
kr8WyrA7Z9Ni+vy4Y8YPfvgxmWEYwwYaZG77O2DqXnz1TfogN
8ztEMMEx9AoXGESw3k6GoB+Q3DwTpngVOLrBOTaGXWTEzys3k
GrHJEZkhNRfRGirqVzOG9RJfLpLeu6l1jQxjCWwDUekj8qbiB
M6lLGKoBcg==
Verifying document...
Verified: true


Verändere Text...
Text: Follow the white rabbit clown
Verifying document...
Verified: false
```

Das Beispiel führt das oben beschriebene Verfahren durch und signiert den gegebenen Text. Danach wird der Text mittels Signatur verifiziert, siehe `Verified: true`. Nun wird der Text leicht verändert und danach wird versucht ihn ein zweites mal zu verifizieren. Diesmal stimmen die Hashes jedoch nicht überein, wodurch der Text nicht verifiziert werden konnte.