zusammengestellt von: Mark Manulis (mark.manulis@nds.rub.de)
Version 1.0

Grundpraktikum für IT-Sicherheit

Material zum Versuch

„Kryptografie mit Bouncy Castle“

Lehrstuhl für
Netz- und Datensicherheit
ruhr-universität bochum

Der Versuch findet im NDS-Netzlabor (Raum IC 4/58) statt.

1 Kryptografie in der Theorie

Unter dem Begriff Kryptografie ist die Wissenschaft vom geheimen Schreiben zu verstehen. In diesem Zusammenhang ist eine Chiffre eine Methode des Verschlüsselns. Die unverschlüsselte Nachricht wird als Klartext, die verschlüsselte als Chiffretext bezeichnet. Die Überführung eines Klartextes in einen Chiffretext wird als Verschlüsselung (Chiffrierung), die umgekehrte Überführung als Entschlüsselung (Dechiffrierung) bezeichnet. Der Schlüssel kontrolliert die Ver- bzw. Entschlüsselung.

Ein Kryptosystem kann aus verschiedenen Algorithmen und sogar Protokollen bestehen, die für das jeweilige System definierte Sicherheitsanforderungen erfüllen. In diesem Versuch werden folgende Kryptosysteme behandelt: symmetrische und asymmetrische Verschlüsselungs-systeme, Hashfunktionen, Message Authentication Codes und digitale Signaturen.

1.1 Verfahren der symmetrischen Verschlüsselung

Ein symmetrisches Verschlüsselungssystem SVe besteht aus drei Algorithmen:

- Gen(t) – erzeugt den geheimen Schlüssel k nach Eingabe des Sicherheitsparameters t.
- Enc(k, M) – verschlüsselt Nachricht M zu Chiffre C unter Benutzung von k.
- Dec(k, C) – entschlüsselt Chiffre C zu Nachricht M unter Benutzung von k.

Damit SVe korrekt ist, muss folgende Anforderung für alle k und M erfüllt werden:

$$Dec(k, Enc(k, M)) = M$$

Zu den Sicherheitsanforderungen eines symmetrischen Verschlüsselungssystems zählen unter anderem die Geheimhaltung des Schlüssels und Integrität der verschlüsselten Nachrichten.

Einige Verfahren der symmetrischen Verschlüsselung sind sogenannte Blockchiffren. Eine Blockchiffre teilt die zu verschlüsselnde Nachricht M in Klartextblöcke M_1 bis M_n, und verschlüsselt diese einzeln mit Schlüsseln k_1 bis k_n, die aus dem Schlüssel k abgeleitet werden, um Chiffretextblöcke C_1 bis C_n zu bekommen. Diese müssen dann zu einem Chiffretext C zusammengeführt werden. Die Verschlüsselung eines Blocks dauert meistens mehrere Runden und enthält Transposition und Substitution einzelner Bits.

Falls die Blocklänge der Chiffre größer als die Länge des Klartextes (oder eines Klartextblocks) ist, so muss die letztere künstlich erweitert werden. Diese Erweiterung wird als *padding* bezeichnet.

Tabelle 1 zeigt die wesentlichen Merkmale ausgewählter symmetrischer Blockchiffren.

Chiffre	Blocklänge (Bits)	Schlüssellänge (Bits)	Anzahl der Runden
DES	64	56	16
IDEA	64	128	9
AES	128	128, 192, 256	10, 12, 14

Tabelle 1. Symmetrische Blockchiffren

Darüberhinaus gibt es weitere symmetrische Blockchiffren, wie Triple-DES, Blowfish, Skipjack, CAST, RC. Eine Beschreibung einzelner Verfahren finden Sie in [1] und [2].

1.2 Verfahren der asymmetrischen Verschlüsselung

Ein asymmetrisches Verschlüsselungssystem AVe besteht aus drei Algorithmen:

- Gen(t) – erzeugt ein Schlüsselpaar (sk, pk) nach Eingabe des Sicherheitsparameters t. Dabei ist sk der geheime Schlüssel, und pk der öffentliche Schlüssel.
- Enc(pk, M) – verschlüsselt Nachricht M zu Chiffre C unter Benutzung von pk.
- Dec(sk, C) – entschlüsselt Chiffre C zu Nachricht M unter Benutzung von sk aus dem Paar (sk, pk).

Damit AVe korrekt ist, muss folgende Anforderung für alle (sk, pk) und M erfüllt werden:

$$Dec(sk, Enc(pk, M)) = M$$

Im Vergleich zur symmetrischen Verschlüsselung wird bei der asymmetrischen Verschlüsselung eine Trennung des Schlüssels k in einen öffentlichen Teil pk und einen geheimen Teil sk vorgenommen. Solche Verfahren bezeichnet man deshalb auch als Public-Key-Verfahren.

Um eine Nachricht verschlüsselt an einen Empfänger zu versenden, wird sie mit seinem öffentlichen Schlüssel pk und Chiffrierungstransformation Enc verschlüsselt. Der Empfänger verwendet seinen geheimen Schlüssel sk und die Dechiffrierungstransformation Dec, um die Nachricht zu entschlüsseln.

Die Sicherheitsanforderungen für ein AVe sind ähnlich zu denen für ein SVe. Auch hier darf der Angreifer nicht in der Lage sein, den geheimen Schlüssel sk zu bestimmen, oder Integrität von verschlüsselten Nachrichten zu umgehen.

Tabelle 3 zeigt ausgewählte Public-Key-Verschlüsselungsverfahren.

Verschlüsselungsverfahren	Schlüssellänge (Bits)
RSA	512, 1024, 2048, …
ElGamal	512, 1024, 2048, …

Tabelle 3. Public-Key-Verschlüsselungsverfahren

Die Beschreibung von verschiedenen Public-Key Verschlüsselungsverfahren finden Sie in [3].

1.3 Hashfunktionen

Eine Hashfunktion $H:\{0,1\}^* \rightarrow \{0,1\}^l$ bildet eine Nachricht M einer beliebiegen Länge auf einen Funktionswert der festgelegten Länge ab. Der Funktionswert wird oft als Fingerabdruck der Nachricht bezeichnet.

Eine kryptografisch-sichere Hashfunktion muss folgende Anforderungen erfüllen:

- *Schwache Kollisionsfreiheit*: Ein Angreifer muss nicht in der Lage sein, eine Nachricht M' für eine gegebene Nachricht M zu finden, sodass H(M') = H(M) gilt.
- *Starke Kollisionsfreiheit*: Ein Angreifer muss nicht in der Lage sein, zwei verschiedene Nachrichten M und M' zu finden, sodass H(M) = H(M').
- *Ein-Weg-Funktion*: Ein Angreifer muss nicht in der Lage sein, aus einem gegebenen Fingerabdruck H(M) die Nachricht M zu bestimmen.

Hashfunktionen werden in der Kryptografie oft im Zusammenhang mit digitalen Signaturverfahren (siehe Abschnitt 1.5) benutzt. Wenn man zur Herstellung einer Signatur der Nachricht M ein asymmetrisches Signaturverfahren verwendet, ist die Signatur genau so lang, wie die Nachricht. Die Berechnung einer Signature nimmt meistens viel Zeit in Anspruch. Falls also keine gleichzeitige Verschlüsselung der Nachricht erforderlich ist, so kann die Signatur von dem Fingerabdruck der Nachricht (H(M)) erstellt werden. Der Fingerabdruck der Nachricht kann deutlich kürzer als die Nachricht gewählt werden. Damit kann der Zeitaufwand für die Erstellung und Verifizierung der Signatur reduziert werden.

Falls eine der Sicherheitsanforderungen für die Hashfunktion nicht erfüllt ist, so kann ein Angreifer die Integrität der Signatur stören oder sogar selbst gefälschte Signaturen erstellen.

Tabelle 4 zeigt ausgewählte Hashfunktionen und die Länge ihrer Fingerabdrücke.

Verschlüsselungsverfahren	**Länge des Fingerabdrucks (Bits)**
MD5	128
SHA-1, -224, -256, -384, -512	160, 224, 256, 384, 512
RIPEMD-128, -160, -256, -320	128, 160, 256, 320

Tabelle 4. Hashfunktionen

Eine Beschreibung von Hashfunktionen finden Sie in [4].

1.4 Message Authentication Codes

Ein Message Authentication Code MAC besteht aus drei Algorithmen:

- Gen(t) – erzeugt ein Schlüssel k nach Eingabe des Sicherheitsparameters t.
- Tag(k, M) – erstellt den Fingerabdruck T der Nachricht M unter Benutzung von k.
- Ver(k, M, T) – verifiziert den Fingerabdruck T der Nachricht M unter Benutzung von k und liefert 1 oder 0 zurück.

Damit AVe korrekt ist, muss folgende Anforderung für alle k und M erfüllt werden:

$$Ver(k, M, Tag(k, M)) = 1$$

Die Sicherheitsanforderungen an einen MAC umfassen den Schutz gegen einen Angreifer, der versucht ein verifizierbares Paar (M, T) ohne Kenntnis von k zu bestimmen.

Message Authentication Codes (MACs) werden oft mit Hilfe von Hashfunktionen implementiert, die neben der Eingabenachricht zusätzlich den geheimen Schlüssel k als Parameter erhalten. Der Fingerabdruck Tag(k, M) der Nachricht M kann somit nur von einem Empfänger bestimmt werden, der den geheimen Schlüssel k kennt. Damit ist es möglich die

Authentizität ohne Geheimhaltung für die Kommunikation mit dem symmetrischen Schlüssel k zu erreichen. Das bekannteste Beispiel eines MACs ist sogenannter HMac.

Eine Beschreibung verschiedener MACs finden Sie in [4].

1.5 Digitale Signaturverfahren

Ein digitales Signaturverfahren DS besteht aus drei Algorithmen:

- Gen(t) – erzeugt ein Schlüsselpaar (sk, pk) nach Eingabe des Sicherheitsparameters t. Dabei ist sk der geheime Schlüssel, und pk der öffentliche Schlüssel.
- Sig(sk, M) – erzeugt eine digitale Signatur S der Nachricht M unter Benutzung von sk.
- Ver(pk, M, S) – verifiziert die digitale Signatur S auf die Nachricht M unter Benutzung von pk und liefert 1 oder 0 zurück.

Damit DS korrekt ist, muss folgende Anforderung für alle (sk, pk) und M erfüllt werden:

$$Ver(k, M, Sig(k, M)) = 1$$

Ein digitales Signaturverfahren muss die Authentizität und Nicht-Abstreitbarkeit der Nachricht gewährleisten. Ein Angreifer darf nicht in der Lage sein den geheimen Schlüssel sk, oder ein verifizierbares Paar (M, S) selbst zu bestimmen. Ein Signierer darf nicht die Erstellung einer Signatur abstreiten können.

Der Unterschied zu MACs ist, dass die digitalen Signaturverfahren zu den asymmetrischen Public-Key-Verfahren gehören. Damit kann der Absender eine Nachricht signieren, und der Empfänger die Authentizität der Signatur überprüfen ohne dass die beiden einen gemeinsamen geheimen Schlüssel teilen. Die Nicht-Abstreitbarkeit kann prinzipiell nur mit digitalen Signaturen realisiert werden.

Einige Public-Key-Verfahren, wie z.B. RSA, können sowohl als asymmetrische Verschlüsselungssysteme AVe, als auch digitale Signaturverfahren DS verwendet werden. Dabei führen die Algorithmen DS.Sig(sk, M) und AVe.Dec(sk, M) identische Operationen aus und liefern die Signatur S zurück. Für erfolgreiche Verifizierung liefert der Algorithmus Enc(pk, S) die Nachricht M zurück. Bei solchen AVe Verfahren muss demnach gewährleistet werden, dass

$$Enc(pk, Dec(sk, M)) = M$$

Andere Verfahren, wie ElGamal benutzen für Verschlüsselung und Signaturerstellung unterschiedliche Transformationen. Zudem gibt es eine ganze Reihe von Signaturverfahren, wie DSA oder ECDSA, die keine Verschlüsselungstransformationen beinhalten.

Eine Beschreibung verschiedener digitaler Signaturverfahren finden Sie in [5].

2 Kryptografie in der Praxis – Legion of the Bouncy Castle

Legion of the Bouncy Castle (http://www.bouncycastle.org) ist eine Java-Klassenbibliothek der Open-Source-Community, die eine Implementierung (Klassen und Quellcode) von verschiedensten kryptografischen Primitiven zur Verfügung stellt. Jeder, der über die entsprechenden Vorkenntnisse in Java-Programmierung verfügt, kann diese Klassenbibliothek nutzen. Legion of Bouncy Castle ist von SUN als Crypto-Provider anerkannt.

Die aktuelle Javadoc-API ist unter http://www.bouncycastle.org/docs/docs1.4/index.html, sowie lokal unter *C:\BouncyCastle\BouncyCastle-JDK14-JavaDoc* verfügbar. Im Folgenden wird kurz erläutert, wie die einzelnen Mechanismen zu benutzen sind.

2.1 Symmetrische Verschlüsselung

Alle Klassen, die symmetrische Ver-/Entschlüsselungsalgorithmen enthalten, implementieren das Interface BlockCipher aus dem Paket org.bouncycastle.crypto. Dazu zählen unter anderen Klassen, die gleichnamige Algorithmen implementieren, wie z. B. AESEngine, DESEngine, IDEAEngine aus dem Paket org.bouncycastle.crypto.engines. Nachdem eine Instanz der Klasse mit dem jeweiligen Konstruktor (z. B. AESEngine _engine = new AESEngine()) erzeugt wird, muss diese mit der Funkion init(boolean, CipherParameters) initialisiert werden. Dabei gibt die Boolesche Variable an, ob man verschlüsseln (true) oder entschlüsseln (false) möchte. CipherParameters ist ein Interface aus dem Paket org.bouncycastle.crypto, das die notwendigen Schlüsselparameter beschreibt. Für die symmetrischen Chiffren ist die Klasse KeyParameter aus dem Paket org.bouncycastle.crypto.params von Bedeutung, die dieses Interface implementiert und den symmetrischen Schlüssel in Bytes enthält. Auf den Schlüssel kann mit der Funktion getKey() zugegriffen werden.
Um einen symmetrischen Schlüssel einer bestimmten Bitlänge zu erzeugen, muss ein Objekt der Klasse KeyGenerationParameters aus dem Paket org.bouncycastle.crypto erzeugt werden. Dabei wird dem Konstruktor die Schlüssellänge und ein Objekt der Klasse SecureRandom aus dem Paket java.security übergeben. Das Objekt der Klasse SecureRandom wird intern benutzt, um sichere Zufallszahlen zu generieren. Als nächstes wird ein Objekt der Klasse CipherKeyGenerator erzeugt und mit dem vorhandenen Objekt der Klasse KeyGenerationParameters initialisiert. Die Funktion generateKey() von CipherKeyGenerator wird dann benutzt, um die Bytes des Schlüssels zu erzeugen. Diese Bytes werden benutzt um eine Instanz der Klasse KeyParameter zu erzeugen.
Wie in Abschnitt 1.1 erklärt, kann eine Blockhiffre nur Blöcke bestimmter Länge bearbeiten. Damit auch Nachrichten bearbeitet werden können, die länger als die zulässige Blocklänge sind, gibt es in BouncyCastle die Klasse PaddedBufferedBlockCipher aus dem Paket org.bouncycastle.crypto.paddings. Diese Klasse wird bei der Erzeugung mit einer Instanz des Interfaces BlockCipher initialisiert. Anschliessend kann mit der Funktion processBytes() die Nachricht verarbeitet werden. Diese Funktion füllt das übergebene Feld mit den Ausgabebytes und liefert die Anzahl dieser Bytes zurück. Die Verarbeitung der Eingabenachricht muss mit der Funktion doFinal() abgeschlossen werden. Die Funktion füllt das übergebene Feld, das schon einige Ausgabebytes enthält mit den restlichen Ausgabebytes und gibt die Anzahl der hinzugefügten Bytes zurück.

2.2 Asymmetrische Verschlüsselung

Klassen der asymmetrischen Ver-/Entschlüsselungsalgorithmen implementieren das Interface AsymmetricBlockCipher aus dem Paket org.bouncycastle.crypto. Dazu zählen unter anderen Klassen, die gleichnamige Algorithmen implementieren, wie z. B. ElGamalEngine, RSA-Engine aus dem Paket org.bouncycastle.crypto.engines. Genauso wie bei der symmetrischen Verschlüsselung, muss die Instanz dieser Klassen nach der Erzeugung mit dem jeweiligen Konstruktor, mit der Funkion init(boolean, CipherParameters) initialisiert werden. Auch hier gibt die boolesche Variable an, ob man verschlüsseln (true) oder entschlüsseln (false) möchte. Für die asymmetrische Verschlüsselung ist die Klasse AsymmetricKeyParameter aus dem Paket org.bouncycastle.crypto.params von Bedeutung. Sie implementiert das Interface CipherParameters. Jeder asymmetrische Verschlüsselungsalgorithmus hat eine eigene Klasse, die die Klasse AsymmetricKeyParameter beerbt, wie z.B. ElGamalKeyParameters oder RSAKeyParameters aus dem gleichen Paket. Von diesen Klassen werden ihrerseit private und öffentliche Parameterklassen abgeleitet, wie z.B. ElGamalPrivateKeyParameters und ElGamalPublicKeyParameters.

Um ein Schlüsselpaar für den asymmetrischen Verschlüsselungsalgorithmus zu erzeugen, muss zunächst eine Instanz des entsprechenden Generators aus dem Paket org.bouncycastle.crypto.generators erzeugt und initialisiert werden (z.B. ElGamalKeyGenerationParameters oder RSAKeyGenerationParameters). Bei verschiedenen Algorithmen läuft auch die Initialisierung unterschiedlich. Bei ElGamal, z.B., müssen zusätzlich sogenannte ElGamalParameters vorgeneriert werden. Dazu wird ein ElGamalParametersGenerator verwendet. Dagegen bei RSA kann das Objekt der Klasse RSAKeyGenerationParameters direkt mit der Schlüssellänge, SecureRandom und dem öffentlichen Exponenten als BigInteger initialisiert werden. Die initialisierte (RSA/ElGamal)KeyGenerationParameters dienen zur Initialisierung des Objekts der Klasse (RSA/ElGamal)KeyPairGenerator, der das Schlüsselpaar als Objekt der Klasse AsymmetricCipherKeyPair aus dem Paket org.bouncycastle.crypto mit dem Aufruf der Funktion generateKeyPair() erzeugt. Das erzeugte Objekt verfügt über zwei Funktionen: getPrivate() und getPublic() mit denen auf den privaten bzw. öffentlichen Schlüssel (Instanz des Interfaces CipherParameters) zugegriffen werden kann. Man muss darauf achten, dass bei der Verschlüsselung die öffentlichen und bei der Entschlüsselung die privaten Schlüsselparameter dem AsymmetricBlockCipher-Objekt zusammen mit der entsprechenden Booleschen Variable übergeben werden.

Auch asymmetrische Chiffren sind Blockchiffren, die Nachrichten bestimmter Länge verarbeiten können. Um eine kurze Nachricht zu verarbeiten kann die Funktion processBlock() des jeweiligen Objekts der Klasse AsymmetricBlockCipher verwendet werden. Ist die Nachricht länger als die Blockgröße, so ist die Klasse BufferedAsymmetricBlockCipher hilfreich. Das Objekt dieser Klasse wird mit dem jeweiligen AsymmetricBlockCipher und seinen Parametern initialisiert. Um die gesamte Nachricht blockweise zu verarbeiten, werden die Funktionen processBytes() und doFinal() nacheinander benutzt. Nach jedem Block muss BufferedAsymmetricBlockCipher mit der Funktion reset() „geleert" werden, damit keine Bytes aus vorherigen Berechnungen die Verarbeitung des nächsten Blocks stören. Die Ausgaben aus der Verarbeitung einzelner Blöcke müssen dann künstlich in einem Feld zusammengefasst werden.

2.3 Hashfunktionen

Alle Hashfunktionenklassen implementieren das Interface Digest aus dem Paket org.bouncycastle.crypto und sind im Paket org.bouncycastle.crypto.digests untergebracht, wie z. B. SHA1Digest, MD5Digest, RIPEMD128Digest usw. Um die Hashfunktion zu benutzen,

muss zunächst das jeweilige Objekt erzeugt werden (z.B. SHA1Digest digest = new SHA1Digest()). Die Funktion getDigestSize() liefert die Größe des produzierten Fingerabdrucks. Damit kann das Ausgabefeld initialisiert werden. Mit der Funktion update() können dem Digest neue Eingabebytes hinzugefügt werden, und das solange bis die Funktion doFinal() aufgerufen wird, die das Ausgabefeld mit den Bytes des Fingerabdrucks füllt.

2.4 Message Authentication Codes

Alle MACs befinden sich im Paket org.bouncycastle.crypto.macs, wie z.B. HMac und implementieren das Interface Mac aus dem Paket org.bouncycastle.crypto. Je nach verwendetem MAC ändern sich die Parameter des jeweiligen Konstruktors. Z. B. beim HMac wird eine Hashfunktion, also eine Instanz des Interfaces Digest übergeben. Anders als bei Hashfunktionen muss die Instanz von Mac nach der Erzeugung mit einem Schlüsselparameter (Instanz der Klasse CipherParameters) initialisiert werden. Der Schlüssel dafür kann genauso wie in Abschnitt 2.1 beschrieben erzeugt werden. Die Funktion getMacSize() liefert die Größe des produzierten Fingerabdrucks (MACs). Damit kann, wie bei Hashfunktionen das Ausgabefeld initialisiert werden. Ähnlich sind die Funktionen update() und doFinal() zu benutzen.

2.5 Digitale Signaturen

Bouncy Castle enthält verschiedene Verfahren zur Erzeugung von digitalen Signaturen. Zum einen gibt es Klassen die das Interface Signer aus dem Paket org.bouncycastle.crypto implementieren, wie z.B. PSSSigner. Das sind Signaturverfahren, die auf verschiedenen standardisierten Public-Key-Verfahren basieren, wie z.B. RSA-Signaturverfahren. Zum anderen gibt es Klassen die das Interface DSA aus demselben Paket implementieren, wie z. B. DSASigner oder ECDSASigner. Diese Implementierungen befolgen den standardisierten Digital Signature Algorithm.

2.5.1 Digitale Signatur mit Signer

Die Konstruktoren der Klassen, die das Interface Signer implementieren, enthalten unter anderem ein Public-Key-Verfahren (Instanz der Klasse AsymmetricBlockSigner) sowie eine Hashfunktion (Instanz der Klasse Digest). Nach der Erzeugung wird Signer mit init(boolean, CipherParameters) initialisiert. Die Boolesche Variable gibt an, ob das Verfahren zum Signieren (true) oder zum Verifizieren (false) benutzt wird. Die zugehörigen Schlüsselparameter CipherParameters müssen entsprechend dem gewählten AsymmetricBlockCipher generiert werden, wie im Abschnitt 2.2 erläutert. Die Funktion update() wird verwendet um die Eingabebytes der zu signierenden oder verifizierenden Nachricht dem Signer hinzuzufügen. Beim Signierungsprozess werden mit der Funktion generateSignature() die Bytes der Signature erstellt. Beim Verifizierungsprozess wird die eingegebene Signatur überprüft und eine boolesche Variable zurückgegeben, die das Verifizierungsergebnis enthält.

2.5.2 Digitale Signatur mit DSA

Im Gegensatz zu Signer-Klassen werden den Konstruktoren der DSA-Klassen keine Public-Key-Verfahren und Hashfunktionen übergeben. Das ist so, weil der DSA-Algorithmus ein standardisierter Mechanismus zum Signieren und Verifizieren ist. Genauso wie die Signer-Klassen müssen die DSA-Klasse mit der Funktion init(boolean, CipherParameters) initialisiert werden. Die CipherParameters müssen gemäß der verwendeten DSA-Klasse (DSASigner oder ECDSASigner) erstellt werden. Die Erzeugung der CipherParameters ist ähnlich zu der Erzeugung von asymmetrischen Schlüsselpaaren aus Abschnitt 2.2. Auch hier werden

(DSA/EC)KeyGenerationParameters definiert, die einen (DSA/EC)KeyPairGenerator initialisieren, um damit die entsprechende Instanz der AsymmetricCipherKeyPair zu erhalten. Nach der Initialisierung kann die Signatur mit der Funktion generateSignature() aus der übergebenen Eingabebytes erzeugt werden. Die Signatur ist ein BigInteger-Feld mit zwei Elementen gemäß dem DSA-Standard. Die Verifikation der Signatur kann mit der Funktion verifySignature() durchgeführt werden. Diese Funktion bekommt als Eingabe die Bytes der Nachricht, sowie beide BigInteger-Werte der Signatur, und liefert eine boolesche Variable zurück, die das Ergebnis der Verifikation enthält.

3 Versuch

3.1 Voraussetzungen für die Teilnahme

- Dieses Dokument muss vorher gelesen und verstanden werden
- Sie müssen in der Lage sein, die Kontrollfragen zu beantworten
- Vorkenntnisse in Java-Programmierung
- Sie müssen sich mit den Basics der Java-Entwicklungsumgebung Eclipse vertraut machen. (Diese kann kostenlos unter http://www.eclipse.org/downloads/ runtergeladen werden. Kostenlose Tutorials zu Eclipse finden Sie unter http://www.3plus4software.de/eclipse/index.html)
- Sie müssen sich die Beschreibung (http://www.bouncycastle.org) und die Javadoc-API (http://www.bouncycastle.org/docs/docs1.4/index.html) von Bouncy Castle anschauen.
- Zum besseren Verständnis des Versuchs sollten Sie sich die in Abschnitt Links ausgewiesenen Kapiteln des Buches „Handbook of Applied Cryptography" anschauen.

3.2 Versuchsaufbau

Nach den einleitenden Informationen erhalten Sie nun eine konkrete Versuchsbeschreibung.

Auf jedem Rechner ist die Java-Entwicklungsumgebung von Eclipse (Version 3.0.0) und die Java Developement Kit (Version 1.5.0_01) installiert, sowie das Eclipse-Projekt cryptobc im Workspace *C:\BouncyCastle\cryptobc* angelegt. Das Projekt besteht aus sieben Java-Klassen: AsymmetricCipher.java, DigitalSignatures1.java, DigitalSignatures2.java, HashFunctions.java, MACs.java, SymmetricCiphers.java und Utility.java. An das Projekt ist die Klassenbibliothek von Bouncy Castle *bcprov-jdk15-125.jar* angebunden.

Die Gruppenanordnung wird im Versuch festgelegt.

3.3 Versuchsdurchführung

Schritt 1:

- Starten Sie die Java-Entwicklungsumgebung von Eclipse und laden sie das Projekt cryptobc im Workspace *C:\BouncyCastle\cryptobc*. Die nächsten Schritte beziehen sich auf die Arbeit in dieser Entwicklungsumgebung.
- Öffnen Sie die Klasse Utility.java. Versuchen Sie anhand der Kommentare in der Datei zu verstehen, was die einzelnen Methoden machen. Schreiben Sie es auf.

Schritt 2:

- Öffnen Sie die Klasse HashFunctions.java. Versuchen Sie anhand der obigen Beschreibung und den Kommentaren in der Datei zu verstehen, was diese Klasse und ihre einzelnen Methoden machen (dazu kann die Klasse auch gestartet werden). Schreiben Sie es auf.
- Ergänzen Sie die Klasse um die zusätzlichen Hashfunktionen SHA512 und RIPEMD320 und testen Sie diese.
- Durch die entsprechende Änderung der Parameter in der Methode main() rufen Sie die Klasse mehrmals auf und vervollständigen Sie die folgende Tabelle (jeder Aufruf muss mindestens dreimal durchgeführt werden, damit die durchschnittliche Zeit möglichst genau ist):

Hashfunktion	Gemessene Zeit (ms)
MD5 (128 Bits)	
RIPEMD128	
RIPEMD160	
SHA1 (160 Bits)	
SHA224	
RIPEMD256	
SHA256	
RIPEMD320	
SHA384	
SHA512	

- Vergleichen Sie die gemessenen Zeiten der Hashfunktionen mit gleicher Fingerabdrucksgröße.

Schritt 3:

- Öffnen Sie die Klasse MACs.java. Versuchen Sie anhand der obigen Beschreibung und den Kommentaren in der Datei zu verstehen, was diese Klasse und ihre einzelnen Methoden machen (dazu kann die Klasse auch gestartet werden). Schreiben Sie es auf.
- Durch die entsprechende Änderung der Parameter in der Methode main() rufen Sie die Klasse mehrmals auf und vervollständigen Sie die folgende Tabelle (jeder Aufruf muss mindestens dreimal durchgeführt werden, damit die durchschnittliche Zeit möglichst genau ist):

Hashfunktion	Schlüssellänge (Bits)	Gemessene Zeit (ms)
MD5	128	
RIPEMD128	128	
RIPEMD160	160	
SHA1	160	
RIPEMD256	256	
SHA256	256	
RIPEMD320	320	
SHA512	512	

- Vergleichen Sie die gemessenen Zeiten der MACs mit gleicher Schlüssellänge.
- Vergleichen Sie die gemessenen Zeiten der MACs mit den Zeiten der gleichnamigen Hashfunktionen aus Schritt 2.

Schritt 4:

- Öffnen Sie die Klasse SymmetricCipher.java. Versuchen Sie anhand der obigen Beschreibung und den Kommentaren in der Datei zu verstehen, was diese Klasse und ihre einzelnen Methoden machen (dazu kann die Klasse auch gestartet werden). Schreiben Sie es auf.
- Ergänzen Sie die Klasse um die zusätzlichen symmetrischen Chiffren IDEA und DESede (Triple-DES) und testen Sie diese.
- Durch die entsprechende Änderung der Parameter in der Methode main() rufen Sie die Klasse mehrmals auf und vervollständigen Sie die folgende Tabelle (jeder Aufruf muss mindestens dreimal durchgeführt werden, damit die durchschnittliche Zeit möglichst genau ist):

Chiffre	Schlüssellänge (Bits)	Schlüsselerzeugungs-zeit (ms)	Verschlüsselungs-zeit (ms)	Entschlüsselungs-zeit (ms)
DES	64			
IDEA	128			
AES	128			
DESede	192			
AES	192			
AES	256			

- Vergleichen Sie die Zeiten für die Verschlüsselung und die Zeiten für die Entschlüsselung einzelner Verfahren.
- Vergleichen Sie die gemessenen Zeiten für Chiffren mit der gleichen Schlüssellänge.

Schritt 5:

- Öffnen Sie die Klasse AsymmetricCipher.java. Versuchen Sie anhand der obigen Beschreibung und den Kommentaren in der Datei zu verstehen, was diese Klasse und ihre einzelnen Methoden machen (dazu kann die Klasse auch gestartet werden). Schreiben Sie es auf.
- Durch die entsprechende Änderung der Parameter in der Methode main() rufen Sie die Klasse mehrmals auf und vervollständigen Sie die folgende Tabelle (jeder Aufruf muss mindestens dreimal durchgeführt werden, damit die durchschnittliche Zeit möglichst genau ist):

Chiffre	Schlüssellänge (Bits)	Schlüsselerzeugungs-zeit (ms)	Verschlüsselungs-zeit (ms)	Entschlüsselungs-zeit (ms)
RSA	512			
ElGamal	512			
RSA	768			
ElGamal	768			
RSA	1024			
RSA	2048			

- Vergleichen Sie die Zeiten für die Verschlüsselung und die Zeiten für die Entschlüsselung einzelner Verfahren.
- Vergleichen Sie die gemessenen Zeiten für Chiffren mit der gleichen Schlüssellänge.

Schritt 6:

- Öffnen Sie die Klasse DigitalSignatures1.java. Versuchen Sie anhand der obigen Beschreibung und den Kommentaren in der Datei zu verstehen, was diese Klasse und ihre einzelnen Methoden machen (dazu kann die Klasse auch gestartet werden). Schreiben Sie es auf.
- Durch die entsprechende Änderung der Parameter in der Methode main() rufen Sie die Klasse mehrmals auf und vervollständigen Sie die folgende Tabelle (jeder Aufruf muss mindestens dreimal durchgeführt werden, damit die durchschnittliche Zeit möglichst genau ist):

Hashfunktion	Schlüssellänge (Bits)	Signierungszeit (ms)	Verifizierungszeit (ms)
SHA1 (160 Bits)	768		
RIPEMD160	768		
SHA1 (160 Bits)	1024		
RIPEMD160	1024		
SHA1 (160 Bits)	2048		
RIPEMD160	2048		
SHA256	768		
RIPEMD256	768		
SHA256	1024		
RIPEMD256	1024		
SHA256	2048		
RIPEMD256	2048		

- Vergleichen Sie die Signierungs- und Verifizierungszeiten einzelner Verfahren.

Schritt 7:

- Öffnen Sie die Klasse DigitalSignatures2.java. Versuchen Sie anhand der obigen Beschreibung und den Kommentaren in der Datei zu verstehen, was diese Klasse und ihre einzelnen Methoden machen (dazu kann die Klasse auch gestartet werden). Schreiben Sie es auf.
- Durch die entsprechende Änderung der Parameter in der Methode main() rufen Sie die Klasse mehrmals auf und vervollständigen Sie die folgende Tabelle (jeder Aufruf muss mindestens dreimal durchgeführt werden, damit die durchschnittliche Zeit möglichst genau ist):

DSA Scheme	Schlüssellänge (Bits)	Signierungszeit (ms)	Verifizierungszeit (ms)
DSA	512		
DSA	768		
DSA	1024		
ECDSA	192 (fest)		

- Vergleichen Sie die Signierungs- und Verifizierungszeiten einzelner Verfahren.
- Vergleichen Sie die Signierungs- und Verifizierungszeiten von DAS und PSS-RSA (aus Schritt 6) für die gleichen Schlüssellängen.

3.4 Schriftliche Ausarbeitung

Jede Gruppe muss eine eigene schriftliche Ausarbeitung anfertigen. Die Ausarbeitung muss alle in den Versuchsschritten aufgeführten Anweisungen dokumentieren. Dazu gehören die Beschreibung einzelner Klassen und Methoden, eigens implementierte Codezeilen, sowie alle Tabellen, ausgefüllt mit gemessenen Zeiten. Ferner muss eine Analyse der Zeitkomplexität, gemäß den Vergleichsanweisungen der Versuchsdurchführung erstellt werden.

Die Versuchsauswertung bitte innerhalb von 7 Tagen per eMail an `mark.manulis@rub.de` senden oder ausgedruckt zum nächsten Termin mitbringen.

3.5 Kontrollfragen

Sie sollten folgende Fragen beantworten können (bevor der Versuch beginnt):

- Zählen Sie die Komponenten der beschriebenen Kryptosysteme auf!
- Was sind die wichtigsten Anforderungen an die beschriebene Kryptosysteme?
- Was ist der Unterschied zwischen symmetrischen und asymmetrischen Verfahren?
- Wie funktionieren die Verfahren der symmetrischen Verschlüsselung?
- Zählen Sie einige symmetrische Verschlüsselungsverfahren auf!
- Wie kann man eine Nachricht mit dem asymmetrischen Verfahren verschlüsselt versenden, und wie kann sie von dem Empfänger dechiffriert werden?
- Wie kann man eine Nachricht mit dem asymmetrischen Verfahre authentisch versenden, und wie kann der Empfänger die Authentizität verifizieren?
- Zählen Sie einige asymmetrische Verschlüsselungsverfahren auf!
- Was sind Hashfunktionen und wozu werden Sie meistens benutzt?
- Zählen Sie einige Hashfunktionen auf und nennen Sie die Größe des jeweiligen Fingerabdrucks!
- Was ist der Unterschied zwischen Hashfunktionen und Message Authentication Codes (MACs)?
- Wozu können MACs verwendet werden?
- Was sind digitale Signaturverfahren?
- Welcher Unterschied gibt es zwischen MACs und digitalen Signaturen?
- Was ist der "Legion of the Bouncy Castle"?
- Wie ist die Bouncy Castle Klassenbibliothek aufgebaut?
- Welche Möglichkeiten bietet die Bouncy Castle Klassenbibliothek, um eine Nachricht digital zu signieren?

Links

[1] Block Ciphers, Handbook of Applied Cryptography (Chapter 7), http://www.cacr.math.uwaterloo.ca/hac/about/chap7.pdf

[2] Noch eine Beschreibung von symmetrischen Verschlüsselungsverfahren, http://www.finecrypt.net/references.html

[3] Public-Key Ciphers, Handbook of Applied Cryptography (Chapter 8), http://www.cacr.math.uwaterloo.ca/hac/about/chap8.pdf

[4] Hashfunktionen und MACs, Handbook of Applied Cryptography (Chapter 9), http://www.cacr.math.uwaterloo.ca/hac/about/chap9.pdf

[5] Digitale Signaturen, Handbook of Applied Cryptography (Chapter 11), http://www.cacr.math.uwaterloo.ca/hac/about/chap11.pdf