Aufgaben zum Praktikum

Kryptologie

Dr. Jürgen Koslowski

Institut für Theoretische Informatik

Technische Universität Braunschweig

im Sommersemester 2011

Inhaltsverzeichnis

Hallo und willkommen beim Praktikum „Kryptologie“!

Die nachfolgend gestellten Aufgaben sind in der Reihenfolge ihrer Abgabetermine aufgeführt und sollten auch in dieser Reihenfolge bearbeitet werden.

Die Abnahme der erstellten Programme erfolgt **ausschließlich** an den jeweiligen Abgabeterminen in den betreuten Übungszeiten. Ein Programm, das an diesen Terminen nicht mit Erfolg abgenommen wird, kann nach **vorheriger Absprache mit den Betreuenden** nur noch in der **darauffolgenden** Woche abgegeben werden. Eine Verschiebung der Abgabe ist in seltenen, begründeten Ausnahmefällen ebenfalls nach vorheriger Absprache mit den Betreuenden möglich.

Eine Bescheinigung über die erfolgreiche Teilnahme am Praktikum „Kryptologie“ erhalten alle diejenigen TeilnehmerInnen, deren Gruppe **alle** zu erstellenden Programme erfolgreich abgenommen bekommen hat und die zu ca. 50% an der Erstellung dieser Programme beteiligt gewesen sind.

Viel Spaß & Erfolg!

Teil I
Chiffren

Richten Sie sich zur Bearbeitung der folgenden Aufgaben zunächst die Programmierumgebung ein. Diese finden Sie auf der Web-Seite des Praktikums zum Herunterladen:

`http://www.tu-braunschweig.de/iti/teaching/ss2010/kryptoprakt`

Entpacken Sie das entsprechende Archiv und benutzen Sie zur Einrichtung der Programmierumgebung die Anleitung unter

`doc/README.html`

Dort ist auch die Programmierumgebung dokumentiert.

Beachten Sie bitte, dass Ihre Routinen zum **Brechen** der jeweiligen Chiffre **nicht** vollautomatisch arbeiten können, da Ihr Programm im Allgemeinen nicht feststellen kann, wann ein gültiger Klartext vorliegt. Vielmehr haben sie die Aufgabe, dem Benutzer eine nach Wahrscheinlichkeit geordnete Liste von Lösungen vorzuschlagen, aus denen dieser dann **auswählen** kann. Dabei soll für den Nutzer auch die Möglichkeit bestehen, andere als die angebotenen Lösungen zu wählen.

1 Vigenère-Chiffre

Abgabe: 3. Wo

Implementieren Sie für *Vigenère*-Chiffren $(f_1, \ldots, f_t)$, mit $f_i(a) = (a + k_i) \bmod n$, $k_i \in \mathcal{A}$, $n = |\mathcal{A}|$ für $i = 1, \ldots, t$ und mit dem **gleichen** Klar- und Chiffretextalphabet $\mathcal{A}$, die Prozeduren zum Erzeugen eines Schlüssels, Chiffrieren, Dechiffrieren und Brechen, indem Sie die Klasse `Cipher` erweitern.

Der Funktionenvektor $(f_1, \ldots, f_t)$ bildet in Form von t Arrays den Schlüssel. Dieser soll erzeugt werden, indem die Parameter n, k_1, $\ldots$, k_t eingelesen und in einer Datei im Format $n \sqcup k_1 \sqcup k_2 \sqcup \ldots \sqcup k_t$ abgelegt werden.

Das Brechen soll unterstützt werden, indem das Programm zunächst nach der *Kasiski*-Methode einige wahrscheinliche Periodenlängen d berechnet. Anschließend werden diese Werte wie in Beispiel 2.13[1] getestet, indem für jeden der d Teiltexte mit Hilfe des approximierten Koinzidenzindexes (siehe Hinweis (2)) eine Periodenlänge des Teiltextes ermittelt wird. Diese sollte in allen d Fällen nahe bei 1 liegen. Periodenlängen, die diesen Test bestehen, werden dem Benutzer zur Auswahl vorgeschlagen. Ausreißer deuten auf eine unwahrscheinliche Periode hin.

Nach Auswahl einer solchen Periodenlängen d werden die resultierenden d Teiltexte jeweilsals *Cäsar*-Chiffretexte behandelt (siehe Beispiel 1.1). Das heißt, es muß lediglich **eine** Zuordnung von Chiffre- zu Klartextzeichen gefunden werden. Verschiedene Vorschläge hierfür sollen anhand der Unigrammtabelle gemacht werden. Dabei sollen mindestens die beiden häufigsten Buchstaben jedes Teiltextes (außer $*$ bei den Moduli 27 und 31) als Kandidaten für “e” und “n” in Betracht gezogen werden. Der ermittelte Schlüssel ist im o.g. Format in einer Datei zu speichern.

[1] Sämtliche Verweise beziehen sich auf folgende Quelle: Wätjen, D.: Kryptographie, Grundlagen, Algorithmen, Protokolle, 2. Aufl., Spektrum Akademischer Verlag, 2008.

Hinweis: (1) Die Programmierumgebung beinhaltet Unigrammtabellen für verschiedene Alphabete und Texte. Für Informationen über Alphabete und Häufigkeitstabellen lesen sie die Dokumentation der Klassen `CharacterMapping` bzw. `FrequencyTables`. Als Beispiel zur Benutzung betrachten Sie die Methode `breakCipher` der Klasse `Caesar`.

(2) Es sei N die Anzahl der Zeichen in einem gegebenen Chiffretext. Dann kann unter Verwendung der Formel für den approximierten Erwartungswert des Koinzidenzindexes

$$E(IC) \approx \frac{1}{d} \cdot \frac{N-d}{N-1} \cdot \sum_{i=0}^{n-1} p_i^2 + \frac{d-1}{d} \cdot \frac{N}{N-1} \cdot \frac{1}{n}$$

eine Formel für die approximierte Periodenlänge

$$d \approx \frac{\left(\sum_{i=0}^{n-1} p_i^2 - \frac{1}{n}\right) \cdot N}{(N-1) \cdot IC - \frac{1}{n} \cdot N + \sum_{i=0}^{n-1} p_i^2} \qquad \text{mit} \qquad IC = \frac{\sum_{i=0}^{n-1} F_i(F_i-1)}{N(N-1)} \approx E(IC)$$

hergeleitet werden, wobei F_i die Häufigkeit des i-ten Zeichens im Chiffretext und p_i die relative Häufigkeit des i-ten Zeichens in einem beliebigen zufälligen Chiffretext bezeichnen. Die p_i können also einer Unigrammtabelle entnommen werden. Für die 4 einfachsten Alphabete gilt somit

n	26	27	30	31
$\sum_{i=0}^{n-1} p_i^2$	0,0773428514	0,0895147618	0,0728648066	0,0875340844

Für weitere Alphabete soll $\sum_{i=0}^{n-1} p_i^2$ durch Ihr Programm berechnet werden.

2 Chiffre mit laufendem Schlüssel

Abgabe: 5. Wo

Implementieren Sie für Chiffren mit *laufendem Schlüssel* die Prozeduren zum Erzeugen eines Schlüssels, Chiffrieren, Dechiffrieren und Brechen, indem Sie die Klasse `Cipher` erweitern (vergl. Buch, S. 28f)

Der Schlüssel besteht aus der Größe n des verwendeten Alphabets und dem Namen einer Datei, die den Schlüsseltext enthält, der mindestens ebenso lang sein muss wie der zu chiffrierende Klartext. Der Schlüssel wird erzeugt, indem diese Parameter eingelesen und in einer Datei im Format $n \sqcup$ *Name* abgelegt werden.

Gehen Sie davon aus, dass es sich bei den Klar- und Chiffretexten um Texte in deutscher Sprache handelt, und implementieren Sie das Brechverfahren von *Friedman* (siehe Beispiel 2.14). Der ermittelte Schlüssel ist im o.g. Format in zu speichern.

Hinweis: Zum Brechen eines Chiffretextes soll der Nutzer nacheinander bestimmte Chiffretextausschnitte, die beliebig im Chiffretext positionierbar sind, betrachten können. Bewerten Sie für einen solchen Ausschnitt die Wahrscheinlichkeit jedes möglichen zugehörigen Paares von Klar- und Schlüsseltextausschnitt anhand der Uni-, Di- und Trigrammtabellen. Da ein bestimmtes Trigramm seltener vorkommt als ein bestimmtes Di- oder gar Unigramm, empfiehlt es sich, die Wahrscheinlichkeit eines Trigramms höher zu bewerten als die eines Digramms und diese höher als die eines Unigramms. Wird beispielsweise ein Chiffretextausschnitt der Länge

4 betrachtet, dann ist in einem (nicht berliner-) deutschen Text das Auftreten der Kombination `eine`, die das häufigste Trigramm `ein` und ein weiteres häufiges Trigramm `ine` enthält, wahrscheinlicher als die Kombination `ener`, die zwar wahrscheinlichere Digramme enthält, dafür aber bei den Trigrammen schlechter abschneidet. Eine brauchbare Bewertungsfunktion, die vereinfachend das Vorkommen von Uni-, Di- und Trigrammen im Klar- und Schlüsseltext als stochastisch unabhängig voraussetzt, lautet

$$\left(g_1 \cdot \sum_{i=1}^{4} k_{1,i} + g_2 \cdot \sum_{i=1}^{3} k_{2,i} + g_3 \cdot \sum_{i=1}^{2} k_{3,i}\right) \cdot \left(g_1 \cdot \sum_{i=1}^{4} s_{1,i} + g_2 \cdot \sum_{i=1}^{3} s_{2,i} + g_3 \cdot \sum_{i=1}^{2} s_{3,i}\right),$$

wobei $k_{j,i}$ bzw. $s_{j,i}$ die den Tabellen entnommenen relativen Häufigkeiten des i-ten j-gramms im Klar- bzw. Schlüsseltextausschnitt sind, und g_j für $j \in \{1, 2, 3\}$ die jeweiligen Gewichtungen angeben. Letztere soll der Benutzer frei wählen können, z.B. gemäß der Heuristik, dass ein Trigramm von mittlerer Wahrscheinlichkeit höher zu bewerten ist als drei Digramme von höchster Wahrscheinlichkeit plus vier Unigramme von höchster Wahrscheinlichkeit und ebenso ein Digramm von mittlerer Wahrscheinlichkeit höher zu bewerten ist als vier Unigramme von höchster Wahrscheinlichkeit. Zur Verringerung der Wahlmöglichkeiten wollen wir die g_j auf natürliche Zahlen < 1000 beschränken.

3 Chiffren mit dem IDEA-Kryptosystem **Abgabe: 7. Wo**

Implementieren Sie für Chiffren gemäß dem *International Data Encryption Algorithm* (IDEA) jeweils die Methoden zur Erzeugen eines Schlüssels, Chiffrieren und Dechiffrieren, indem Sie die Klasse `BlockCipher` erweitern.

Der Schlüssel besteht aus einem String K von 128 Bits. Er soll erzeugt werden, indem der Nutzer entweder eine Zeichenkette aus 16 ASCII-Zeichen eingibt oder indem per Zufallsgenerator eine solche Zeichenkette generiert wird und diese dann in einer Datei abgelegt wird. Die 52 Teilschlüssel werden erst während der (De-)Chiffrierung erzeugt.

Die Klar- und Chiffretexte werden Byteweise gelesen bzw. geschrieben und Blockweise verarbeitet.

Die Chiffrierung und Dechiffrierung soll im *CBC-Modus* erfolgen, wobei für jeden Klartext der Initialisierungsvektor IV per Zufallsgenerator bestimmt werden soll. Dieser wird dann als 0. Chiffretextblock in die Chiffretextdatei geschrieben. Da sich nicht jeder Klartext genau in 64-Bit-Blöcke zerlegen lässt, sollen gegebenenfalls einige Leerzeichen automatisch verschlüsselt werden, um den Klartext passend aufzufüllen. Achten Sie bei Art und Weise Ihrer Implementierung auf Wiederverwendbarkeit in späteren Aufgaben.

Hinweise: Beachten Sie bitte, dass Sie Ihre IDEA-Implementation im 2. Teil des Praktikums in Aufgabe 6 verwenden werden.

Klartext	abcdefghijklmnopqrstuvwxyz
Schlüssel	abcdefghijklmnop
IV	ddc3a8f6c66286d2
Chiffretext	ca106c5b47af041e46947aa28244440d75f20cbff040253cb16778bcc8132786
Klartext	Ein kurzer deutscher Satz.
Schlüssel	`!@#$%Krypto^&*()`
IV	5c7119dd40913232
Chiffretext	ff0cfc50bf4a49d6cb76c77f3de2ae52f6e17d1ff878facfe555b05bab4ee49b

Beachten Sie, dass die ersten 64 Bits der Chiffretexte jeweils die Initialisierungsvektoren des CBC-Modus sind (hier als Zeile IV) und der Rest des Chiffretextes von diesen zufällig gewählten Werten abhängt.

Einen weiteren Testvektor finden Sie im Beispiel 7.106 im Abschnitt 7.6 des Handbook of Applied Cryptography[2].

Weiterhin können die Klassen

`java.math.BigInteger` und

`de.tubs.cs.iti.jcrypt.chiffre.BigIntegerUtils`

verwendet werden.

4 ElGamal-Public-Key-Chiffren **Abgabe: 8. Wo**

Implementieren Sie für Chiffren gemäß dem *ElGamal-Public-Key-Verschlüsselungsverfahren* und gemäß dem *ElGamal-Public-Key-Signaturverfahren* jeweils die Methoden zum Erzeugen eines Schlüssels, Chiffrieren und Dechiffrieren, indem Sie die Klasse `BlockCipher` bzw. `Signature` erweitern.

Beiden Verfahren benötigen einen privaten Schlüssel (p, g, x) und einem öffentlichen Schlüssel (p, g, g^x). Die Schlüsseldatei enthält die Pfade zu den Dateien mit dem öffentlichen und privaten Schlüssel.

Sofern die Schlüssel noch nicht existieren, sollen sie an dieser Stelle gemäß Algorithmus 7.4 erzeugt werden können. Lesen Sie hierzu die Dokumentationen zu den Klassen `java.math.BigInteger` und `de.tubs.cs.iti.jcrypt.chiffre.BigIntegerUtils`. Die *sichere* Primzahl p soll hierbei eine Bitlänge von mindestens 512 Bits aufweisen. Legen Sie den öffentlichen Schlüssel für die Geheimhaltung unter

```
$KRYPTOUMGEBUNG/ElGamal/schluessel/<Ihr Accountname>.secr.public
```

und den für die Authentifizierung unter

```
$KRYPTOUMGEBUNG/ElGamal/schluessel/<Ihr Accountname>.auth.public
```

ab, und zwar jeweils im Format

1. Zeile: p
2. Zeile: g
3. Zeile: $g^x \bmod p$

Machen sie diese mit `chmod a+r` für alle lesbar. Ihre privaten Schlüssel dürfen Sie in einem von Ihnen gewählten Format speichern und mit `chmod go-rw` gegen fremden Zugriff sichern.

Um die Verfahren überhaupt anwenden zu können, muss jeweils die zu chiffrierende Datei blockweise in eine numerische Darstellung konvertiert und umgekehrt die numerische Darstellung einer dechiffrierten Datei in Textblöcke zurückübersetzt werden. Damit alle Gruppen miteinander kommunizieren können, muss diese Konvertierung einheitlich geschehen. Verwenden Sie daher die Methoden `readClear`, `writeClear`, `readCipher` und `writeCipher` der Klasse `BlockCipherUtil`. Die Blocklänge L muss hierbei so gewählt werden, dass die durch

[2]Menezes, A. J.; van Oorschot, P. C.; Vanstone, S. A.: Handbook of Applied Cryptography, CRC Press, 1996, verfügbar unter `http://www.cacr.math.uwaterloo.ca/hac/`.

die Konvertierung errechnete Zahl stets kleiner ist als der Modulus p (je größer L desto geringer ist der Rechenaufwand). Wenn L_p die Bitlänge von p ist, wäre also

$$L = \left\lfloor \frac{L_p - 1}{8} \right\rfloor$$

optimal. Die maximale von der Programmierumgebung unterstützte Bitlänge von p ist 2048.

Zum ElGamal-Public-Key-Verschlüsselungsverfahren: Das Chiffrieren und Dechiffrieren erfolgt gemäß einer Variante des Algorithmus 7.5, bei der der berechnete Chiffretext $C = (a, b)$ zu einer einzigen Zahl $C' = a + b \cdot p$ weiterverarbeitet wird, was die Handhabung des Chiffretextes erheblich vereinfacht. Entsprechend müssen beim Dechiffrieren zunächst $a = C' \bmod p$ und $b = C' \operatorname{div} p$ berechnet werden.

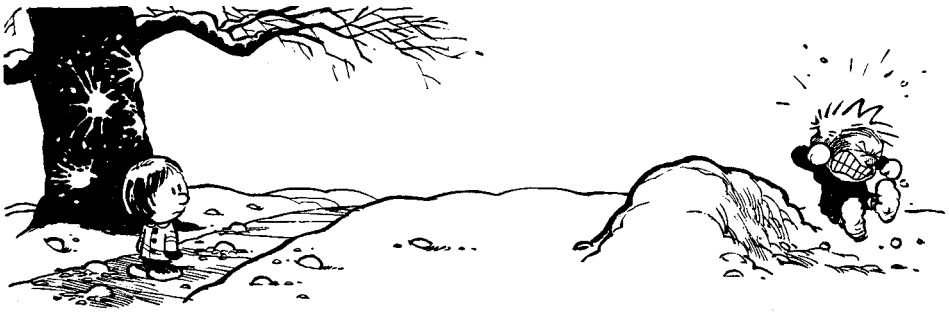
Zum ElGamal-Public-Key-Signaturverfahren: Das Chiffrieren und Dechiffrieren erfolgt gemäß einer Variante des Algorithmus 7.8, bei der ebenfalls der berechnete Chiffretext $C = (r, s)$ zu einer einzigen Zahl $C' = r + s \cdot p$ weiterverarbeitet wird. Beim Dechiffrieren wird geprüft, ob die angegebene Chiffretextdatei die Signatur der angegebenen Klartextdatei enthält, und das Ergebnis auf dem Bildschirm ausgegeben.

Testen Sie Ihre Programme, indem Sie in

`$KRYPTOUMGEBUNG/ElGamal/nachrichten/`

verschlüsselte und authentifizierte Nachrichten für andere Gruppen ablegen und an Sie gerichtete Nachrichten dechiffrieren, wobei die Signatur einer Datei `text` den Namen `text.auth` erhalten soll.

Hinweis: Beachten Sie bitte, dass Sie Ihre Implementierung des ElGamal-Verfahrens im 2. Teil des Praktikums in den Aufgaben 7 und 9 weiterverwenden werden.



5 Fingerabdruck mit Chaum, van Heijst und Pfitzmann **Abgabe: 9. Wo**

Implementieren Sie eine stark kollisionsfreie Hashfunktion, die eine Bitfolge beliebiger Länge auf eine Bitfolge fester Länge abbildet, indem Sie die *Hashfunktion von Chaum, van Heijst und Pfitzmann* aus Abschnitt 7.6 implementieren und diese dann gemäß Algorithmus 6.1 erweitern.

Ihre Implementation soll die Methoden `hash`, `verify` und `makeParam` der Klasse `HashFunction` erweitern. Die Methode `makeParam` soll die Parameter (p, g_1, g_2), wobei p sichere Primzahl mit einer Bitlänge von mindestens 512 Bits und g_i, $i = 1, 2$, verschiedene primitive Wurzeln modulo p sind, erzeugen und in einer Datei im Format

1. Zeile: p
2. Zeile: g_1
3. Zeile: g_2

speichern. Die Methode `hash` berechnet aus gegebenem Klartext und Parametern (p, g_1, g_2) den Fingerabdruck und speichert diesen als Chiffretext, wobei der Fingerabdruck als Hexadezimalzahl ähnlich wie in Aufgabe 3 zu speichern ist. Die Methode `verify` prüft, ob Klar- und Chiffretext zueinander passen.

Hinweise: (1) Beachten Sie, dass Sie die Implementation dieser Aufgabe in der Aufgabe 6 weiterverwenden werden.

(2) Die Hashfunktion von Chaum, van Heijst und Pfitzmann ist eine Kompressionsfunktion $h : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{Z}_p^*$. Um diese mit Algorithmus 6.1 zu verwenden, muss sie in die Form $h' : \mathbb{Z}_2^m \to \mathbb{Z}_2^t$ gebracht werden. Eingabebitfolgen der Länge m werden also in zwei Hälften m_1 und m_2 zerlegt, und dann in Zahlen x_1 und x_2 des Typs `BigInteger` mit $0 \leq x_1, x_2 < q$ umgewandelt. Der Hashwert $h(x_1, x_2)$ muss anschließend wieder in eine Bitfolge umgewandelt werden.

Damit hierbei nicht die geforderten Wertebereiche verletzt werden, muss $m = 2 \cdot (L_q - 1)$ und $t = L_p$ gelten, wobei L_q und L_p die Bitlängen von q und p sind. Da $L_p = L_q + 1$ gilt, ist offenbar für $L_q \geq 5$ immer $m \geq t + 2$, was für Algorithmus 6.1 verlangt war.

Es werden im Laufe des Praktikums noch Testwerte in dem Verzeichnis

`$KRYPTOUMGEBUNG/Chaum_vanHeijst_Pfitzmann/`

zur Verfügung gestellt werden.

Teil II
Protokolle

Richten Sie sich Ihre Programmierumgebung für die Bearbeitung der folgenden Aufgaben jeweils mit dem Befehl

```
$KRYPTOUMGEBUNG/protokoll/make_local.sh
```

bzw. auf analoge Weise auf Ihrem lokalen System ein. Achten Sie darauf, dass Ihre Programme ausschließlich die in den Protokollen genannten Informationen senden bzw. empfangen. Implementieren Sie jeweils die im Buch beschriebenen Betrugsmöglichkeiten und -aufdeckungsmaßnahmen, und ergänzen Sie diese, so dass jede Art von Betrug sowohl von Ihnen unternommen als auch vom Programm aufgedeckt werden kann. Aus Praktikabilitätsgründen empfiehlt es sich hierbei, die implementierten Betrugsoptionen komplett abschalten zu können.

Der bereitgestellte Server hat Probleme mit Zeilenumbrüchen, bitte vermeiden Sie diese.

6 Station-to-Station-Protokoll **Abgabe: 10. Wo**

Implementieren Sie die Methoden `sendfirst` und `receivefirst` gemäß dem Station-to-Station-Protokoll 8.6 für den Schüsselaustausch. Als symmetrisches Kryptosystem E benutzen Sie dabei Ihre Implementation von IDEA aus Aufgabe 3, und als Hashfunktion h wird die Hashfunktion aus Aufgabe 5 verwendet.

Um ein gültiges Zertifikat zu erhalten, stehen Ihnen die Klassen `Certificate` und `TrustedAuthority` bereits zur Verfügung, deren Dokumentation auch die Beschreibung der Signaturtransformationen E_T und D_T zu entnehmen ist. Wir modifizieren Protokoll 8.6 für unsere Zwecke folgendermaßen:

(0) Alice und Bob lesen zunächst die Parameter (p_1, g_1, g_2) für die Hashfunktion aus einer vom Nutzer anzugebenden Datei. Danach wählt Alice die Parameter p und g für den Schlüsselaustauch und sendet diese an Bob. Zuletzt sendet Alice ihren öffentlichen RSA-Schlüssel (e_A, n_A) an Bob, worauf Bob seinen öffentlichen Schlüssel (e_B, n_B) an Alice schickt.

(2),(5) Um Hashwerte der Form $h(u, v)$ zu berechnen, formen Sie $m = u \cdot p + v$, wandeln dies in eine Bitfolge um und berechnen $h(m)$, was wiederum in eine Zahl vom Typ `BigInteger` umgewandelt wird.

(2),(6) Als Schlüssel für das IDEA-Kryptosystem werden die niederwertigsten 128 Bits der Zahl k verwendet.

(7) Alice und Bob starten eine verschlüsselte Kommunikation (ähnlich wie „chat“) beginnend mit Alice: Alice gibt eine Nachricht ein, schickt sie an Bob und wartet dann auf Antwort von Bob. Nachdem sie die Antwort erhalten hat, gibt sie wieder eine Nachricht ein, schickt sie an Bob und wartet auf Antwort usw. Bob verhält sich komplementär.

Hinweis: Ziel dieser Aufgabe ist es, zwischen den Programmen *verschiedener* Gruppen eine verschlüsselte Kommunikation durchzuführen. Damit die Kommunikation reibungslos

vonstatten geht, müssen daher alle Daten genau in der im Protokoll angegebenen Reihenfolge gesendet werden.

Die Parameter (p_1, g_1, g_2) für die Hashfunktion dürfen nicht von den Teilnehmern des Schlüsselaustausches gewählt werden. Falls zum Beispiel Alice diese Parameter wählte, könnte sie g_1 und g_2 derart bestimmen, dass sie $\log_{g_1} g_2$ kennt, was aber laut Definition 7.2 nicht der Fall sein soll. Aus diesem Grunde werden die Parameter für die Hashfunktion (Formatierung siehe Aufgabe 5 im Verzeichnis

`$KRYPTOUMGEBUNG/Station-to-Station/`

bereitgestellt.

Die Zahl p soll eine sichere Primzahl von mindestens 512 Bits Länge sein. Die RSA-Moduli von Alice und Bob sollen mindestens 1024 Bits lang sein.

7 1-von-2-Oblivious-Transfer-Protokoll **Abgabe: 11. Wo**

Implementieren Sie die Methoden `sendFirst` und `receiveFirst` gemäß dem 1-*von*-2-*Oblivious-Transfer*-Protokoll 10.5, wobei das RSA- durch das ElGamal-Public-Key-Kryptosystem zu ersetzen ist. Dies hat zur Folge, dass im *Klar*textraum der RSA-Modulus n durch den ElGamal-Modulus p, jedoch im *Chiffre*textraum der Modulus n durch p^2 ersetzt werden muss. Bei der Übergabe des öffentlichen Schlüssels (p_A, g_A, y_A) von Alice wird zuerst p_A, dann g_A und zuletzt y_A gesendet.

Hinweis: Aus der Verwendung des ElGamal-Verfahrens ergeben sich im Vergleich zum RSA-Verfahren Schwierigkeiten beim Feststellen eines Betrugsversuches. In den Überlegungen im Anschluss an Protokoll 10.5 wird die Äquivalenz von $k'_{r\oplus 1} = D_A((q - m_{r\oplus 1}) \bmod n)$ mit $E_A(k'_{r\oplus 1}) = q - m_{r\oplus 1} \bmod n$ benutzt. Diese gilt *nicht* beim ElGamal-Verfahren, da hier $E_A(D_A(C))$ im Allgemeinen nicht mit C übereinstimmt.

Wir modifizieren das Protokoll folgendermaßen: In Schritt (3) berechnet Alice zusätzlich mit dem ElGamal-Signaturverfahren aus Aufgabe 4 die Signaturen S_i von k'_i, $i = 0, 1$, und sendet diese an Bob. In Schritt (4) kann Bob einen Betrugsversuch feststellen, wenn er die Signatur $S_{r\oplus 1}$ gegen den Wert $\bar{k}_{r\oplus 1}$ testet. Fällt der Test positiv aus, gilt $\bar{k}_{r\oplus 1} = k'_{r\oplus 1}$, d. h. es hat ein Betrugsversuch stattgefunden.

Das bedeutet, dass Alice im Falle eines Betrugsversuches in Schritt (3) nicht zwei gültige Signaturen schicken darf. Sie muss versuchen, das Bit r richtig zu raten und schickt Bob eine echte Signatur S_r und eine gefälschte Signatur $S_{r\oplus 1}$. Da Alice das Bit r nicht kennt, kann Bob in Schritt (4) mit Wahrscheinlichkeit $\frac{1}{2}$ den Betrugsversuch aufdecken.

Zur Implementierung: Der Benutzer soll Alice anweisen, ob und an welcher Stelle sie betrügen soll.

8 Geheimnisaustausch-Protokoll **Abgabe: 13. Wo**

Implementieren Sie das Protokoll 10.7, indem Sie Ihre Prozeduren aus Aufgabe 7 erweitern. Es werden maximal $n = 10$ Geheimnispaare ausgetauscht und Bob soll einen maximalen Berechnungsvorteil von $(2^k + 1)$ zu 2^k für $k \in \{0, \ldots, 7\}$ erhalten. Die auszutauschenden Geheimnisse seien hierbei Wörter über dem Alphabet $\{0, \ldots, 9, a, \ldots, z\}$ der Länge ≤ 10, wobei

führende Nullen ignoriert werden. Zur einfacheren Bearbeitung soll ein solches Wort w mittels des Konstruktors `BigInteger(`w`,36)` in eine Zahl `I` vom Typ `BigInteger` umgewandelt werden, die wegen

$$\log_2 36^{10} = 10 \cdot \frac{\log 36}{\log 2} \approx 51,7$$

durch $m = 52$ Bits kodiert werden könnte. Die Rückwandlung erfolgt mittels `I.toString(36)`. Um den Datenaustausch exakt festzulegen, wird Protokoll 10.7 wie folgt konkretisiert:

- Das Durchlaufen von Indexmengen erfolgt stets in der üblichen aufsteigenden Reihenfolge. Indexpaarmengen werden in aufsteigender lexikografischer Ordnung durchlaufen.
- Schritt (1) wird ersetzt durch: Alice sendet ein $n \in \{1, \ldots, 10\}$ und ein $k \in \{0, \ldots, 7\}$ an Bob. Anschließend senden beide je 1 von 2 Geheimnissen eines jeden Geheimnispaares gemäß dem Protokoll aus Aufgabe 7.

Betrugsversuche durch Alice sollen wieder vom Benutzer spezifiziert werden.

9 Vertragsunterzeichnungsprotokoll **Abgabe: 14. Wo**

Implementieren Sie, indem Sie Ihre Prozeduren aus Aufgabe 8 modifizieren, die folgende Version des *Vertragsunterzeichnungsprotokolls* 10.8:

(0) Wie in Aufgabe 8 wählen Alice und Bob zunächst ihre privaten und öffentlichen Schlüssel und tauschen die öffentlichen Schlüssel aus. Außerdem soll der Name der Datei, die den Vertragstext C enthält, von beiden Parteien eingelesen werden.
Dann bestimmt Alice die Anzahl $n \in \{1, \ldots, 10\}$ der M-Puzzles und wählt zufällig eine große Primzahl $p_A < 2^{52}$ sowie eine Zahl $M \ll p_A$. Sie sendet n, p_A und M in dieser Reihenfolge an Bob und empfängt p_B von Bob.
Bob empfängt n, p_A und M von Alice, bestimmt ebenfalls zufällig eine Primzahl p_B mit $M \ll p_B < 2^{52}$ und sendet diese an Alice.
Hierbei ist darauf zu achten, dass die Zahlen p_A und p_B mit hoher Wahrscheinlichkeit Primzahlen sein müssen.

(1) Alice berechnet zufällig n Paare von beliebigen Schlüsseln $(A_{i,1}, A_{i,2})$, $i = 1, \ldots, n$ für das **Pohlig-Hellmann-Schema** mit dem Modulus p_A. Das heißt, für alle Schlüssel muss lediglich $\mathrm{ggT}(A_{i,j}, p_A - 1) = 1$ erfüllt sein. Damit berechnet sie die M-Puzzles

$$C_{A_{i,j}} = M^{A_{i,j}} \bmod p_A$$

und sendet diese in der Reihenfolge $C_{A_{1,1}}, C_{A_{1,2}}, \ldots, C_{A_{n,1}}, C_{A_{n,2}}$, wobei sie nach jedem Senden eines M-Puzzles $C_{A_{i,j}}$ ein entsprechendes M-Puzzle $C_{B_{i,j}}$ von Bob empfängt. Anschließend erzeugt Alice die Erklärung gemäß Protokoll 10.8 (1), wobei die Phrase „Vertrag C“ durch „untenstehende Vertrag“ ersetzt wird und der Vertragstext aus der zuvor angegebenen Datei an die Erklärung angehängt wird. Alice berechnet mittels der SHA-Hashfunktion (siehe Dokumentation der Klasse `java.security.MessageDigest`) den Fingerabdruck H des gesamten Textes und chiffriert diesen mit ihrem privaten Schlüssel gemäß der Methode `sign` des ElGamal-Signaturverfahrens aus Aufgabe 4 zu $E_A(H)$. Danach sendet sie den Erklärungs- und Vertragstext und anschließend $E_A(H)$ an Bob und erwartet analoge Nachrichten von Bob.
Alice berechnet den SHA-Hashwert der empfangenen Erklärung und prüft ihre Authentizität mit Bobs öffentlichem Schlüssel. Sie sendet **0** und wartet auf **0** oder **1** von

Bob, falls das Protokoll fortgesetzt werden soll, oder sendet **1** und terminiert, falls das Protokoll abgebrochen werden soll.
Bob verhält sich analog, jedoch komplementär bzgl. des Sendens und Empfangens.

(2) Alice und Bob führen das Protokoll 10.7 zum gleichzeitigen Austausch von Geheimnissen gemäß Aufgabe 8 (jedoch mit $k \leq 4$) aus, wobei die $2n$ Schlüsselpaare die auszutauschenden Geheimnispaare darstellen.
Dabei wird mit Hilfe der in (1) empfangenen M-Puzzles $C_{A_{i,j}}$ bzw. $C_{B_{i,j}}$ zusätzlich getestet, ob das jeweils empfangene Geheimnis eine Lösung eines der beiden zugehörigen M-Puzzles bzw. des verbliebenen ungelösten M-Puzzles ist.
Implementieren Sie ebenfalls die entsprechende Betrugsmöglichkeit.