Lehrstuhl Technik der Rechnernetze
Institut für Experimentelle Mathematik &
Institut für Informatik und Wirtschaftsinformatik
Web: http://www.uni-due.de/tdr/

UNIVERSITÄT DUISBURG ESSEN

Network and Information Security

Programmierprojekt

1. Einleitung

Das Programmierprojekt soll Ihnen ermöglichen, Praxiserfahrung im Umgang mit Verschlüsselungen zu erlangen. Dazu sollen Sie verschiedene mathematische Funktionen und Verschlüsselungsalgorithmen in Java implementieren. Die Parameter für die verschiedenen Funktionen und Algorithmen erhalten Sie dabei von einem bereitgestellten Server, welcher außerdem Ihre Lösung überprüft und Ihnen darüber Rückmeldung gibt, ob Sie eine Aufgabe korrekt gelöst haben. Die Kommunikation mit dem Server wird Ihnen in Form eines Client-Frameworks vorgegeben, in welches Sie bitte Ihre Lösungen implementieren.
Im Folgenden finden Sie eine Beschreibung des Client-Frameworks und der einzelnen Aufgaben sowie Beispiele zu diesen.

2. Vorbereitung

Bitte laden Sie sich, nach dem Einloggen ins TMT (Teaching Management Tool; https://tmt.tdr.iem.uni-due.de/), unter dem Navigationspunkt „Programmierprojekt“ den Quelltext des Client-Frameworks herunter. Auf derselben Seite können Sie auch einsehen, welche Aufgaben Sie bearbeiten können.
Zur Authentisierung am Server werden Ihre Matrikelnummer und Ihr TMT-Passwort benötigt. Bitte tragen Sie diese in der Klasse Client in die entsprechenden Variablen ein. Ihr Passwort wird nur in Form eines Hashes übertragen. Sie sollten aber darauf achten, dass Ihr Quelltext nicht für andere Personen einsehbar ist und dass Sie Ihr Passwort entfernen, bevor Sie Ihren Quelltext per TMT abgeben.

3. Client-Framework

Das vorgegebene Client-Framework steuert die Kommunikation mit dem Server. Sie müssen lediglich das Abrufen, Bearbeiten und Abschicken der Aufgaben bzw. der Lösungen implementieren. Sie können dabei beispielsweise so vorgehen, dass Sie in einer Schleife alle zu bearbeitenden Aufgaben durchlaufen und je nach Aufgabentyp die erhaltenen Parameter an verschiedene Methoden übergeben. Dieser Ansatz ist in der Klasse *Client* exemplarisch implementiert.

Das Framework besteht aus drei verschiedenen Klassen:

1. Client
   Die „Hauptklasse“ des Frameworks. In dieser ist bisher nur der Verbindungsaufbau zum Server implementiert. In diese Klasse können Sie Ihre Methoden zum Lösen der Aufgaben implementieren. Zur einfacheren Handhabung der Aufgabenidentifikation

implementiert diese Klasse das Interface *TaskDefs*, welche Ihnen zu jeder Aufgabe eine Konstante definiert.
Diese Klasse ist die einzige Klasse, an der Sie Änderungen vornehmen sollten. Sie können allerdings neue Klassen definieren, wenn Sie dies wollen.

2. Connection
   Die Klasse *Connection* steuert die Verbindung zum Server. Sie bietet Methoden an, um Aufgaben vom Server abzurufen, die Lösung zum Server zu schicken und für einige Aufgaben erweiterte Parameter zu übermitteln. Diese drei für Sie relevanten Methoden werden später genauer erläutert.
   Diese Klasse ist so implementiert, dass falls der Server einen Fehler zurückgibt, dieser ausgegeben und das Programm beendet wird. Sie müssen sich also nicht selbstständig um Fehlerbehandlung kümmern.

3. TaskObject
   Diese Klasse bildet die Datenstruktur für eine einzelne Aufgabe. Wenn Sie eine Aufgabe über die von der Connection-Klasse bereitgestellten Methoden abrufen, erhalten Sie die Aufgabe als Objekt dieser Klasse.
   Die Klasse besteht aus einer Variable für den Aufgabentyp, sowie jeweils einem Array für String, int und double, welche je nach Aufgabe mit den benötigten Aufgabenparametern gefüllt sind. Der Zugriff auf diese Variablen erfolgt über Getter-Methoden (siehe Quelltext und/oder JavaDoc).

3.1. Abrufen einer Aufgabe

Zum Abrufen einer Aufgabe vom Server stellt die Connection-Klasse die Methode `TaskObject getTask(int taskId)` zur Verfügung. Bei einigen Aufgaben müssen weitere Parameter übertragen werden. Dazu wird die Methode `TaskObject getTask(int taskId, String[] params)` bereitgestellt.
Das zurückgegebene Objekt enthält jeweils die für den Aufgabentyp relevanten Parameter in den drei schon beschriebenen Arrays.

3.2. Senden einer Lösung

Zum Senden einer Lösung an den Server stellt die Connection-Klasse die Methode `boolean sendSolution(String solution)` zur Verfügung. Dabei wird die Lösung immer als String übertragen. Ggf. müssen sie also einen Typecast vornehmen. Die Methode gibt *true* zurück, wenn Ihre Lösung korrekt war, andernfalls *false.*

3.3. Senden erweiterter Informationen

Da einige Aufgaben erst nach dem Übertragen zusätzlicher Informationen (z.B. eines öffentlichen Schlüssels bei asymmetrischer Verschlüsselung) generiert werden kann, stellt die Connection-Klasse die Methode `void sendMoreParams(TaskObject task, String[] params)` zur Verfügung. Die vom Server erhaltene Aufgabe wird dabei in das übergebene Objekt der Klasse TaskObject geschrieben. Die Parameter werden immer als String-Array übergeben. Sollte nur ein Parameter übertragen werden müssen, so muss dieser erst in ein Array kopiert werden.

Weitere Informationen entnehmen Sie bitte dem Quelltext und den dazugehörigen JavaDocs.

4. Bearbeitungshinweise

Die Lösung des Programmierprojekts darf nur mit Hilfe des vorgegebenen Frameworks gelöst werden. Eine Lösung in anderen Programmiersprachen oder unter Zuhilfenahme zusätzlicher Frameworks ist nicht erlaubt.

Grundsätzlich gilt, dass das Programmierprojekt unter "klausurähnlichen" Bedingungen gelöst werden soll. Wie bereits in den Folien erwähnt, sollen keine Java-eigenen Routinen, die die Aufgabenstellung lösen können, zum Einsatz kommen. Dies betrifft z.B. auch die Umrechnung von Hexadezimal- in Binärschreibweise, da Sie dies in der Klausur per Hand erledigen müssten (und keinen Taschenrechner dafür benutzen dürften).
Bei einigen Aufgaben ist die Umwandlung von Buchstaben in ihre ASCII-Werte erforderlich. Falls eine solche Aufgabe in der Klausur gestellt würde, würden wir Ihnen eine ASCII-Tabelle zur Verfügung stellen; die Umwandlung müsste jedoch von Ihnen durchgeführt werden. Das bedeutet, dass Sie eine ASCII-Tabelle "hardcoden" dürfen, die Umwandlung der Buchstaben in diese Werte jedoch selbst programmieren müssen und nicht die eingebauten Java-Routinen benutzen dürfen.

Zur Erstellung von Zufallszahlen dürfen Sie eingebaute Methoden nutzen - Ihr Gehirn kann dies schließlich auch.

Folgende Methoden sind z.B. erlaubt:

- Array<T>.length
- String.charAt(int)
- String.length()
- String.substring(int, int)
- StringBuilder.StringBuilder()
- StringBuilder.StringBuilder(String)
- StringBuilder.insert(int, char)
- StringBuilder.length()
- StringBuilder.append(char)
- StringBuilder.append(int)
- StringBuilder.toString()
- Typecasting

Folgende Funktionen sind verboten:

- %
- ^
- Integer.valueOf()
- Integer.parseInt()
    - Zur Umrechnung zwischen Zahlensystemen verboten, als reiner Typecast jedoch erlaubt
- Character.forDigit()
- Character.getNumericValue()

5. Aufgaben

Im Folgenden sind alle Aufgaben inkl. der benötigten Parameter und dem geforderten Ausgabeformat beschrieben. Die Arrays, die bei der Beschreibung des Inputs aufgeführt sind, sind dabei die des Aufgaben-Objektes TaskObject: sa: String-Array, ia: int-Array, da: double-Array. Die Lösungen werden immer als String im angegebenen Format übertragen.

Einige Aufgaben erfordern, dass der Client zusätzliche Informationen an den Server schickt (z.B. ein öffentlicher Schlüssel, mit dem der Server einen Klartext für den Client verschlüsseln kann). Diese Informationen werden mithilfe der *moreParams*-Methode der Connection-Klasse übermittelt oder direkt bei der Aufgabenanforderung übertragen. Wie es bei den einzelnen Aufgaben zu handhaben ist, entnehmen Sie bitte den unten stehenden Erläuterungen.

1. Klartext
   Klartext soll unverschlüsselt zurückgegeben werden.
   Input: sa[0]: Klartext als String
   Output: Klartext als String (Groß-/Kleinschreibung wird nicht berücksichtigt.)

2. XOR
   Binäre XOR-Verknüpfung zweier HEX-Strings.
   Input: sa[0], sa[1] (jeweils ein zufälliger HEX-String)
   Output: Binäre Zahl als ein String (z.B. „101001010…“).

3. Modulo
   Modulo-Berechnung zweier Integer.
   Input: ia[0]: Integerzahl1, ia[1]: Integerzahl2
   Output: ia[0] mod ia[1]

4. Faktorisierung
   Zufallszahl in Primfaktoren zerlegen.
   Input: ia[0]: Zufallszahl als Integer
   Output: Primfaktoren, aufsteigend sortiert und mit Sternchen(*) getrennt als String. (z.B.: „2*2*5*7“)

5. Vigenère
   Entschlüsselung eines Chiffretexts per Vigenère-Verfahren mit gegebenem Schlüssel.
   Input: sa[0]: Chiffretext als String, sa[1]: Key als String
   Output: Klartext als ein String (Groß-/Kleinschreibung wird nicht berücksichtigt)

6. DES: Rundenschlüssel-Berechnung
   Berechnung des Rundenschlüssels für eine gegebene Runde und gegebenen Schlüssel.
   Input: sa[0]: Key als Binär-String, ia[0]: geforderte Runde (1 - 16) als Integer
   Output: Roundkey als ein Binär-String (48Bit) (z.B. „1001111011…“)

7. DES: R-Block-Berechnung
   Berechnung des R-Blocks für eine gegebene Runde und gegebenen Input. Der Schlüssel wird hierbei als 0 angenommen.
   Input: sa[0]: Binär-String (64Bit), ia[0]: geforderte Runde (1 - 16) als Integer
   Output: R-Block als Binär-String (z.B. „1001111011…“)

8. DES: Feistel-Funktion
   Einmalige Anwendung der f-Funktion mit vorgegebenem Rundenschlüssel. Die ersten 32 Bit des Inputs bilden den L-Block, die zweiten 32 Bit den R-Block.

Input: sa[0]: Input als Binär-String(64Bit), sa[1]: Rundenschlüssel als Binär-String (48Bit)
Output: L-Block XOR R-Block als Binär-String

9. DES: Berechnung einer Runde
   Durchführung einer kompletten Runde inkl. Rundenschlüssel-Berechnung.
   Input: sa[0]: L-Block der vorherigen Runde (Binär-String, 32Bit), sa[1]: R-Block der vorherigen Runde (Binär-String, 32Bit), sa[2]: Key (64Bit), ia[0]: geforderte Runde als Integer
   Output: Binär-String (64Bit, zuerst L-Block 32Bit dann R-Block 32Bit) (z.B. „1001111011…“)

10. AES: Multiplikation im Raum GF8
    Multiplikation zweier HEX-Zahlen in GF(28)
    Input: sa[0]: HEX-Zahl1 als String, sa[1]: HEX-Zahl2 als String
    Output: Ergebnis der GF8-Multiplikation als HEX-String (Groß-/Kleinschreibung wird nicht berücksichtigt.) (z.B. „2f“) Führende Nullen sind mit anzugeben.

11. AES: Schlüssel-Generierung
    Generierung von 3 Rundenschlüsseln. Als Output sollen alle drei Schlüssel der Reihenfolge entsprechend jeweils durch einen Unterstrich („_“) getrennt und in Hexadezimal-Darstellung an den Server gesendet werden.
    Input: sa[0]: Key als HEX-String (128 Bit)
    Output: HEX-String aller drei Rundenschlüssel (jeweils 128 Bit) jeweils durch einen Unterstrich („_“) getrennt (z.B. „a56e…_35c…_72a…“)

12. AES: MixColumns()
    Durchführung der MixColumns-Funktion. Der Input ist spaltenweise angegeben. D.h. die ersten vier Byte des Inputs entsprechen der ersten Spalte.
    Input: sa[0]: HEX-String (128Bit)
    Output: Gemischte Spalten als ein HEX-String (128Bit) (z.B. „21ae5….“)

13. AES: SubBytes(), ShiftRows() und MixColumns()
    Berechnung in der Reihenfolge von SubBytes(), ShiftRows() und MixColums() für einen gesamten Datenblock.
    Input: sa[0]: HEX-String (128Bit)
    Output: HEX-String(128Bit) (z.B. „21ae5….“)

14. AES: Initiale & zwei weitere Runden
    Berechnung von „Initial Round“ und zwei weiterer „Standard Rounds“;.
    Input: sa[0]: Datenblock als HEX-String (128Bit), sa[1]: Key als HEX-String (128Bit), sa[2]: Keyroom als Zahlen-String (z.B. „128“)
    Output: Ausgabe (des verschlüsselten Textes) aller drei Runden als ein HEX-String (128Bit), wobei die verschlüsselten Texte der Reihenfolge nach sortiert sind und jeweils durch einen Unterstrich („_“) getrennt werden (z.B. 34e2…_e7c…_a45b…)

15. RC4: Generation Loop
    Generieren von pseudo-zufälligen Bytes mithilfe des State-Tables.
    Input: sa[0]: State-Table (Integer-Werte durch Unterstrich („_“) getrennt. z.B.: 2_1_3_0 wären die State-Table-Werte an den Positionen null bis drei), sa[1]: Klartext als String
    Output: Inhalt des State-Tables an der Stelle t des jeweiligen Loops. Alle Werte sollen in einem String (ohne Trennzeichen) hintereinander ausgegeben werden. (z.B. „2130“)

16. RC4: Keyscheduling
    RC4-Schlüsselgenerierung.
    Input: sa[0]: Key als Zahlen-String (Integer-Werte durch Unterstrich („_“) getrennt. z.B.: 1_7_1_7 wären die Schlüssel-Werte an den Positionen null bis drei)
    Output: State-Table (Integer-Werte durch Unterstrich („_“) getrennt. z.B.: 2_1_3_0 wären die State-Table-Werte an den Positionen null bis drei)

17. RC4: Verschlüsselung
    Verschlüsselung unter Verwendung des Generation Loops und des Keyschedulings.
    Input: sa[0]: Key als Zahlen-String (Integer-Werte durch Unterstrich („_“) getrennt. z.B.: 1_7_1_7 wären die Schlüssel-Werte an den Positionen null bis drei), sa[1]: Klartext als String
    Output: Chiffretext als ein Binär-String (z.B. „1000111101011…“)

18. Diffie-Hellman
    Die Lösung von Teil 1 wird mit der Methode *moreParams* des Connection-Objekts an den Server geschickt. Die Lösung von Teil 2 wie gewohnt mit *sendSolution*.

    Teil 1 (Public Key-Berechnung):
    Berechnung des Public Keys.
    Input: ia[0]: p , ia[1]: g, da[0]: B (Variablenbezeichnungen aus den Vorlesungsunterlagen)
    Output: A als String

    Teil 2 (Entschlüsselung eines Chiffretextes):
    Entschlüsseln einer Nachricht mithilfe des in Teil 1 berechneten Schlüssels. Klartext und Schlüssel wurden XOR-verknüpft.
    Input: sa[0]: Chiffretext beliebiger Länge als String. (ASCII-Werte des Chiffretextes durch Unterstrich („_“) getrennt)
    Output: Klartext als String (Zeichenfolge, keine ASCII-Werte, Groß-/Kleinschreibung wird nicht berücksichtigt.)

19. RSA: Verschlüsselung
    Verschlüsselung einer Nachricht per RSA.
    Input: ia[0]: n, ia[1]: e (Variablenbezeichnungen aus den Vorlesungsunterlagen), sa[0]: zu verschlüsselnder Klartext
    Output: Chiffretext als String (ASCII-Werte der einzelnen Zeichen durch Unterstrich („_“) getrennt)

20. RSA: Entschlüsselung

Die Lösung von Teil 1 wird mit der Methode `getTask(int taskId, String[] params)` des Connection-Objekts an den Server geschickt. D.h. der öffentliche Schlüssel wird schon bei der Aufgabenanforderung mitgeschickt.
Die Lösung von Teil 2 wie gewohnt mit `sendSolution`.

Teil 1 (Public Key-Berechnung):

Client generiert zufälliges Schlüsselpaar und sendet den Public Key an den Server.
Output: Public Key als String-Array (String[] array = {n, e})

Teil 2 (Entschlüsselung):

Zufallstext, der mit dem Public Key des Clients (aus Teil 1) verschlüsselt ist, soll entschlüsselt werden.
Input: sa[0]: Chiffretext als String
Output: Klartext als String (Groß-/Kleinschreibung wird nicht berücksichtigt.)

21. ElGamal: Verschlüsselung

Verschlüsselung einer Nachricht per Verfahren von ElGamal.
Input: ia[0]: p, ia[1]: α, ia[2]: β (jeweils als Integer; zusammen: Public Key, mit dem verschlüsselt werden soll), sa[0] zu verschlüsselnder Klartext (Variablenbezeichnungen aus den Vorlesungsunterlagen)
Output: Chiffretext als String (ASCII-Werte der einzelnen Zeichen im Hexadezimalformat durch Unterstrich („_“) getrennt). Dabei wird y1 (siehe Vorlesungsunterlagen) dem Chiffretext im gleichen Format vorangestellt. (also y1_hex1_hex2_...)

22. ElGamal: Entschlüsselung

Die Lösung von Teil 1 wird mit der Methode `getTask(int taskId, String[] params)` des Connection-Objekts an den Server geschickt. D.h. der öffentliche Schlüssel wird schon bei der Aufgabenanforderung mitgeschickt.
Die Lösung von Teil 2 wie gewohnt mit `sendSolution`.

Teil 1 (Public Key-Berechnung):

Client generiert zufälliges Schlüsselpaar und sendet den Public Key an den Server.
Output: Public Key als String-Array (String[] array = {p, α, β})

Teil 2 (Entschlüsselung):

Zufallstext, der mit dem Public Key des Clients (aus Teil 1) verschlüsselt ist, soll entschlüsselt werden.
Input: sa[0]: Chiffretext als String (Format wie Output bei ElGamal: Verschlüsselung, also y1_hex1_hex2_...)
Output: Klartext als String (Groß-/Kleinschreibung wird nicht berücksichtigt.)

6. Aufgaben-Beispiele

Hier finden Sie zu jeder Aufgabe ein Beispiel, mit dem Sie Ihre Methoden testen können.

1. Klartext
   Input: steganography
   Output: steganography

2. XOR
   Input: sa[0] = 457e76, sa[1]: 55db1
   Output: 100000000100011110001111

3. Modulo
   Input: ia[0] = 5021129, ia[1] = 257
   Output: 120

4. Faktorisierung
   Input: ia[0] = 26572
   Output: 2*2*7*13*73

5. Vigenère
   Input: sa[0] = EUOLWQKWRXBL, sa[1] = eavesdropping
   Output: authenticity

6. DES: Rundenschlüssel-Berechnung
   Input:
   sa[0] = 11000000000101000101010100010001110001010100011101010111111000101,
   ia[0] = 5
   Output: 000110110110000101000001001010101100000010101011

7. DES: R-Block-Berechnung
   Input:
   sa[0] = 100100100101111111000100101001111000001011010001100011101110010 0,
   ia[0] = 4
   Output: 11100001011111101110110101111010

8. DES: Feistel-Funktion
   Input:
   sa[0] = 000001110011100110100101001010011101011011011001101111100101 1000,
   sa[1] = 10101000000100010011110111100000011111000110111 0
   Output: 00100001111000110110011101111011

9. DES: Berechnung einer Runde
   Input:
   sa[0] = 011010010000011010111000000110111,
   sa[1] = 010011111001110101001111000000011,
   sa[2] = 111001110101000100101101001011011001011011001101111000110101101 0,
   ia[0] = 12
   Output: 010011111001110101001111000000110001100111101101111001111001 1001

10. AES: Multiplikation im Raum GF8
    Input: sa[0] = 02, sa[1] = 0c
    Output: 18

11. AES: Schlüssel-Generierung
    Input: sa[0] = cb628baeeeba913288654ea330413248
    Output:
    cb628baeeeba913288654ea330413248_4941d9aaa7fb48982f9e063b1fdf3473_d559566a7
    2a21ef25d3c18c942e32cba

12. AES: MixColumns()
    Input: sa[0] = 60866a0a1a4624d36ccdb602aa73a762
    Output: 31c32c5809297af1202ed0cb1fdc2af5

13. AES: SubBytes(), ShiftRows() und MixColumns()
    Input: sa[0] = 3085b849d1547370864a964a9d05998a
    Output: 86919d40c89b620c087704694737ad4d

14. AES: Initiale & zwei weitere Runden
    Input:
    sa[0] = 8eac3d33c3bebc832228e55d5d988d64,
    sa[1] = 63d272d563c88719290287c3a59c493c,
    sa[2] = 176
    Output:
    ed7e4fe6a0763b9a0b2a629ef804c458_9e43056aa2baaa0f91d880693635a0f1_cb8c89435
    c79973f3ae6d374bd031498

15. RC4: Generation Loop
    Input: sa[0] = 1_0_11_9_12_6_3_8_5_10_7_4_2, sa[1] = Datenschutz
    Output: 1412457212947

16. RC4: Keyscheduling
    Input: sa[0] = 5_5_2_4_3_3_1
    Output: 5_3_6_2_0_4_1

17. RC4: Verschlüsselung
    Input:
    sa[0] = 7_11_13_14_20_22_4_1_15_1_24_2_20_23_13_2_13_20_4_21_23_3_9_7,
    sa[1] = encryption
    Output:
    01100110011000100111000001100100011110110111101001111000011001110110110 0
    01100100

18. Diffie-Hellman
    Input: ia[0] = 29, ia[1] = 19, da[0] = 27
    MoreParams (Public Key Client): 22
    Input: sa[0] = 99_104_101_116_127_118_114_111_105_104
    Output: encryption

19. RSA: Verschlüsselung
    Input: ia[0] = 377, ia[1] = 253, sa[0] = wiretapping
    Output: 119_105_114_101_116_97_112_112_105_110_103

20. RSA: Entschlüsselung
    Output (bei `taskRequest`): array{235, 13}
    Input: 195_31_166_168_177_205_136_168_109_177_2_179_126
    Output: steganography

21. ElGamal: Verschlüsselung
    Input: ia[0]: 1931, ia[1] = 2, ia[2] = 1327, sa[0] = nonrepudiation
    Output: 658_191_24_191_368_6db_642_6ac_bd_127_504_8e_127_24_191
    (Vom Client errechneter privater Schlüssel: 876)

22. ElGamal: Entschlüsselung
    Output (bei `taskRequest`): array{6217, 5, 404}
    Input: 5ed_a86_9e9_14f5_6c1_d11_52d_71_52d_10d6_14f5
    Output: cryptology