**Computersystemsicherheit**

# Lösungsblatt Nr. 1

Hendrik Beckmann, Ibrahim Osman,
Patrick Elsen, Tiancheng Chen

Wintersemester 2018-2019
Technische Universität Darmstadt

## Gruppenübung

Die Übungsaufgaben in diesem Bereich sind Gegenstand der Übungen in der Woche vom 29.10.2018–02.11.2018.

### Aufgabe 1 (Verständnisaufgaben).

*In dieser Aufgabe prüfen wir unser Verständnis über den Inhalt der Vorlesung.*

*Benennen Sie die drei Schutzziele aus der Vorlesung.*

Das erste Schutzziel ist *Vertraulichkeit*: die Eigentschaft, dass Informationen nicht Unberechtigten zur Verfügung gestellt wird). *Integrität* bedeutet, dass Informationen akkurat, komplett und unversehrt ist. *Verfügbarkeit* bedeutet, dass auf Informationen zugegriffen werden kann.

*Was besagt Kerckhoffs-Prinzip?*

Das Kerckhoffs-Prinzip sagt, dass die Sicherheit eines kryptographischen Systems nicht auf der Geheimhaltung des Systems, sondern auf der des Schlüssels beruht. Will man ein sicheres System bauen, muss man davon ausgehen, dass der Angreifer das System kennt. Die Sicherheit darf also nicht darauf beruhen, dass der Angreifer das System nicht versteht. Ein sicheres System muss auch dann sicher sein, wenn es bekannt ist.

*Was besagt die funktionale Korrektheit eines symmetrischen Verschlüsselungsverfahren?*

Die funktionale Korrektheit (auch *Vollständigkeit*) besagt, dass man mit allen Schlüsseln jede Nachricht erfolgreich ver- und entschlüsseln kann. Man schreibt also:

$$\forall m \in M, k \in K : \mathrm{dec}(k, \mathrm{enc}(k, m)) = m$$

Wobei $M$ die Menge aller möglichen Nachrichten und $K$ die Menge aller möglicher Schlüsselt ist.

## Aufgabe 2: Schutzziele.

Wie Sie in der Vorlesung gelernt haben, bietet die deutsche Sprache keine eigenen Worte für *Safety* und *Security*. Beide Worte werden in der Regel mit Sicherheit übersetzt. *Safety* bezieht sich auf die Verlässlichkeit von IT-Systemen in Bezug auf Ablauf- und Ausfallsicherheit. Häufig übersetzt mit Betriebssicherheit. *Security* wird oft mit Angriffssicherheit übersetzt und spaltet sich dabei in die Teilaspekte *Authenzität*, *Integrität*, *Vertraulichkeit*, *Verfügbarkeit* und *Verbindlichkeit*.

*Im Folgenden sind einige Szenarien gegeben, die eine Verletzung bestimmter Eigenschaften aufzeigen. Bitte nennen Sie die verletze Schutzeigenschaft und ggf. den Teilaspekt.*

| SZENARIO | SCHUTZEIGENSCHAFT |
|---|---|
| Durch Abhören eines Firmennetzwerkes ist es möglich, Passwörter von Mitarbeitern abzufangen. | Security (Vertraulichkeit) |
| Durch einen Programmierfehler in der Software des U-Bahn-Betriebs kommt es in letzter Zeit häufiger zu Abstürzen und einzelne Bahnen bleiben dabei auch unvorhergesehen in Tunneln stehen. | Safety |
| Ein Vorgesetzter beauftragt seinen Mitarbeiter eine Reise für ihn im internen Buchungssystem zu buchen. Der Mitarbeiter möchte aber nicht mit der Buchung in Verbindung stehen und beauftragt den Datenbankbeauftragten seinen Namen zu entfernen. | Security (Verbindlichkeit) |
| Durch eine Schwachstelle in einer Web Applikation ist es beliebigen (auch nicht registrierten) Nutzern möglich Gästebucheinträge zu verändern. | Security (Integrität) |

## Aufgabe 3: Teilertheorie und Modulare Arithmetik

*Berechnen Sie:*

i) $7 \equiv 1 \mod 2$ ii) $3^5 = 243 \equiv 5 \mod 7$

iii) $4^3 \equiv 1^3 = 1 \mod 3$ iv) $12 \equiv 0 \mod 4$

*Zeigen Sie, dass 429 und 595 teilerfremd sind.*

Das geht am einfachsten mit dem Euklidischen Algorithmus, der den größten gemeinsamen Teiler von zwei beliebigen Zahlen bestimmen kann. Hier kann man ein kleines Ruby-Programm schreiben, welches den größten gemeinsamen Teiler berechnet.

```
1 def euclid(a, b)
2   res = if b == 0 then a else euclid(b, a % b) end
3   puts "euclid(#{a},␣#{b})␣=␣#{res}"
4   res
5 end
6
7 euclid(595, 429)
```

Wenn man dieses Programm laufen lässt, dann sieht man, dass der größte gemeinsame Teiler 1 ist. Dass die Zahlen keine größeren Teiler haben, zeigt, dass sie teilerfremd sind.

```
euclid(1, 0) = 1
euclid(2, 1) = 1
euclid(13, 2) = 1
euclid(28, 13) = 1
euclid(69, 28) = 1
euclid(97, 69) = 1
euclid(166, 97) = 1
euclid(429, 166) = 1
euclid(595, 429) = 1
```

*Sei $n \in \mathbb{N}$ und sei $p \in \mathbb{N}$ ein Teiler von $n$, d. h. $p \mid n$. Zeigen Sie:*

$$a \equiv b \mod n \implies a \equiv b \mod p \quad \text{für alle } a, b \in \mathbb{Z}$$

TODO.

## Aufgabe 4: One-Time-Pad (Vernam Chiffre)

*In der Vorlesung haben Sie das One-Time-Pad (Vernam Chiffre) kennengelernt. Erinnern Sie sich an die Definition des OTP und notieren Sie wie die Verschlüsselung einer Nachricht $m$ und Entschlüsselung des resultierenden Chiffrats $c$ funktioniert.*

Bei einer One-Time-Pad-Verschlüsselung benutzt man die bitweise Exclusive-OR (XOR) Funktion, um die Nachricht zu verschlüsseln und um das Chiffrat zu entschlüsseln.

$$c = m \oplus k \qquad m = c \oplus k \qquad k = m \oplus c$$

*Gegeben sind folgende Nachricht $m = 1010\,1111$ und der Schlüssel $k = 1111\,0000$. Berechnen Sie das OTP Chiffrat.*

Um das Chiffrat zu bestimmen, wird die XOR-Operation auf die Bits der Nachricht und die des Schlüssels angewendet.

$$\begin{array}{r} 1010\,1111 \\ \oplus\ 1111\,0000 \\ \hline 0101\,1111 \end{array}$$

*Was ist perfekte Sicherheit? Und warum erfüllt das OTP diese?*

Perfekte Sicherheit ist die Eigenschaft einer Verschlüsselung, dass das Chiffrat keine Informationen über die Nachricht besitzt. Es können aus diesem also keine Schlüsse über die Nachricht gezogen werden. OTP besitzt perfekte Sicherheit, wenn der Schlüssel geheim ist und zufällig generiert wurde.

Da der Schlüssel zufällig gewählt wurde, ist jedes Bit zu 50% $1$ und zu 50% $0$. Durch den XOR-Operator wird damit jedes Bit der Nachricht zu 50% gewechselt. Damit besitzt das Chriffrat keine Informationen mehr über die Nachricht: für jedes Bit ist die Wahrscheinlichkeit gleich, dass es ein Bit der Nachricht ist, und dass es ein gewechseltes Bit ist.

*Diskutieren Sie warum das OTP unsicher wird sobald man den Schlüssel mehrmals verwendet. Unter welchen Umständen wird das OTP auch unsicher?*

Wenn man eine andere Nachricht, sagen wir $m' = 0011\,0011$ mit dem OTP mit dem selben Schlüssel $k$ verschlüsselt, dann bekommt man das Chiffrat $c'$.

$$\begin{array}{r} 0011\,0011 \\ \oplus\ 1111\,0000 \\ \hline 1100\,0011 \end{array}$$

Idealerweise sollten diese beiden Chiffrate $c$ und $c'$ *keine* Informationen über die Nachrichten $m$ und $m'$ enthalten. Da hier der selbe Schlüssel verwendet wurde, ist dem aber nicht der Fall. Wegen der Eigenschaften der XOR-Operation gilt: $m \oplus m' = c \oplus c'$.

$$\begin{array}{r} 1010\,1111 \\ \oplus\ 0011\,0011 \\ \hline 1001\,1100 \end{array} \qquad \begin{array}{r} 0101\,1111 \\ \oplus\ 1100\,0011 \\ \hline 1001\,1100 \end{array}$$

OTP wird also unsicher, wenn man einen Schlüssel mehrfach verwendet, weil dann Informationen über die Nachrichten preisgegeben werden.

*Diskutieren Sie was passiert wenn Sie XOR ($\oplus$) durch die Operation ODER ($\vee$) im OTP ersetzen.*

Mit der Operation ODER kann der OTP nicht funktionieren. Die Funktionsweise des OTP hängt von der Eigenschaft der XOR-Operation, dass eine wiederholte Anwendung zurück zum Klartext führt. Mathematisch braucht man also einen Operator $\circ$, so dass gilt:

$$m = (m \circ k) \circ k$$

Diese Eigenschaft ist mit dem ODER-Operator nicht gegeben, was an folgenden Beispiel illustriert werden kann:

$$0 \neq (0 \vee 1) \vee 1$$

## Aufgabe 5: Angriff auf eine Chiffre (Known-Plaintext-Attack)

*Im Folgenden wollen wir einen Angriff auf eine Chiffre betrachten. Die zu verschlüsselende Nachricht $m$ und das resultierende Chiffrat $c$ sind Bitstrings der Länge $n$ (d.h. $m, c \in \{0,1\}^n$). Der für die Verschlüsselung notwendige Schlüssel ist folgendermaßen definiert:*

- *einem Bitstring $k \in \{0,1\}^n$.*
- *einer quadratischen invertierbaren Matrix $M \in \{0,1\}^{n \times n}$.*
- *Funktionen $f_i : \{0,1\}^n \to \{0,1\}$ für $i = 1, \ldots, n$.*

*Die Verschlüsselungsoperation für eine beliebige Nachricht $m \in \{0,1\}^n$ und Schlüssel ist wie folgt definiert:*

1. *Setze $v = M * m$, wobei $v = (v_1, \ldots, v_n)$.*
2. *Berechne die $i$-te Chriffratkomponente $c_i = v_i \oplus f_i(k)$ für $i = 1, \ldots, n$.*

*Dadurch erhalten Sie das Chiffrat $c = (c_1, \ldots, c_n)$.*

| M | C |
|---|---|
| 0000 | 1010 |
| 0001 | 1111 |
| 0011 | 0011 |
| 0111 | 0100 |
| 1111 | 1110 |

*Sie haben jetzt diese Klartext und Chiffrat Paare abgefangen. Diese Informationen genügen, um jedes beliebige Chiffrat effizient zu entschlüsseln, obwohl die Funktionen $f_i$ und die Matrix $M$ unbekannt sind. Brechen Sie die obige Chiffre und berechnen Sie den Klartext hinter dem Chiffrat $c = (0010)$.*

Weil hier der Keyspace klein ist, ist es möglich, alle möglichen Schlüssel auszuprobieren. Es ist außerdem möglich, diese Verschlüsselung vereinfacht zu modellieren. So müssen wir zum Beispiel nicht den Bitstring $k$ und alle $f_i$ kennen, es reicht $f_i(k)$ für $i = 1, \ldots, n$ zu kennen (in diesem Fall ist $n = 4$. Das sind vier Bits an Informationen, dazu kommt, dass man die Matrix $M$ kennen muss, die 16 Bits an Informationen enthält.

Wir können einmal alle möglichen Matrizen generieren und herausfinden, wie viele überhaupt invertierbar sind, was den Keyspace noch weiter einschränkt. Dafür kann man ein kleines Ruby-Programm verwenden.

```
require "matrix"

matrices = (2**16)
  .times
```

```
  .map{|n| n.to_s(2).rjust(16,'0')}
  .map{|n| n.chars.map{|c| c.to_i}}
  .map{|n| Matrix[n[0,4], n[4,4], n[8,4], n[12,4]].det}
  .count{|n| n != 0}

p matrices.size
```

Wenn man dieses Programm ausführt, bekommt man als Ergebnis 22560, was nur 34% der theoretisch möglichen $2^{16}$ Matrizen ist. Also schreiben wir ein weiteres Programm, welches alle möglichen Matrizen und Bitstrings $t \in \{0,1\}^4$ durchgeht, und testet, ob diese möglich sind. TODO.

# 1 Hausübung

Dieser Bereich ist dazu gedacht das Gelernte weiter zu vertiefen. Dazu werden je nach Themen weitere Übungsaufgaben, ergänzende Beweise oder ähnliche Aufgaben gestellt. Die Aufgaben sind freiwillig, können aber, bei erfolgreicher Bearbeitung, zu Bonuspunkten in der Klausur führen. Die Abgabe dieser Übungen erfolgt über *Moodle* und kann in Gruppen mit bis zu vier Studenten (*aus Ihrer eigenen Übungsgruppe*) eingereicht werden. Abgaben werden nur als PDF-Dateien akzeptiert. Denken Sie bitte daran, dass Ihre Lösungen nachvollziehbar und entsprechend ausführlich dargestellt werden sollen.

### Hausübung 1: PGP (2 Punkte – Einzelabgabe).

*Sie haben in der Vorlesung das Schutzziele Dreieck (C.I.A.) kennengelernt. Ein Beispiel Angriff auf die Confidentiality ist das Mitlesen der Internetkommunikation, wie z.B. ihrer Emails. Dieses Problem kann man leicht beheben indem man die Emailkommunikation verschlüsselt. PGP (Pretty Good Privacy) ist ein Programm zum Verschlüsseln (und signieren) von Daten und basiert auf einem Public-key (asymetrisches) Verfahren. Führen Sie folgende Schritte durch:*

- *Installieren Sie PGP kostenlos für Ihr bevorzugtes Betriebssystem.*
- *Generieren Sie für Ihre Emailadresse ein Schlüsselpaar und besorgen Sie sich aus Moodle den öffentlichen Schlüssel Ihres Tutors, sowie die jeweilige Emailadresse.*
- *Schicken Sie dann eine verschlüsselte Email an ihren Tutor mit dem Betreff* ComSySec - Hausübung 1 *und mit folgenden Informationen: Ihr Name, Name des Tutors, Übungstermin (inkl. Gruppennummer) und Raumnummer.*

### Hausübung 2: Permutationschiffre und Operationsmodi

*In der Vorlesung haben Sie das Konzept der Blockchiffre und verschiedene Operationsmodi (u.a. ECB und CBC) kennengelernt. Wir definieren eine Blockchiffre E, die die Eingabebits permutiert, wobei die*

*Permutation $\pi$ als Schlüssel fungiert. Die Ver- und Entschlüsselung sei gegeben durch*

$$\text{Enc}((b_1, \ldots, b_n), \pi) := (b_{\pi(1)}, \ldots, b_{\pi(n)})$$
$$\text{Dec}((d_1, \ldots, d_n), \pi) := (d_{\pi^{-1}(1)}, \ldots, d_{\pi^{-1}(n)})$$

*Zeigen Sie, dass dies ein Chiffriersystem definiert. Hinweis: Zeigen Sie, dass Ver- und Entschlüsselung kommutieren.*

Dieses System definiert genau dann ein Chiffriersystem, wenn jede mögliche Nachricht ver- und entschlüsselt werden kann, und die originale Nachricht daraus resultiert.

$$\text{Dec}(\text{Enc}((m_1, \ldots, m_n), \pi), \pi) = (m_1, \ldots, m_n)$$

Das ist genau dann wahr, wenn die Applikation der Permutation und die darauffolgende Applikation der Inversen der Permutation die originalen Bits zurückgibt.

$$\forall n \in \mathbb{N} : n = \pi^{-1}(\pi(n))$$

Da eine Permutation eben so definiert ist, ist dieses ein Chiffriersystem.

*Betrachten Sie nun speziell im Fall $n = 3$ den Schlüssel $\pi = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$ und verschlüsseln Sie den String `101010101010` im ECB-Mode.*

Hier kann man ein kleines Ruby-Programm schreiben, um das Ganze zu vereinfachen.

```ruby
def encrypt_ecb(permutations, data)
  data.scan(/.{3}/).map do |block|
    permutations
      .sort_by{|_, v|}
      .map{|_, p| block[p-1]}
      .join
  end.join
end

puts encrypt_ecb({1 => 2, 2 => 1, 3 => 3}, "101010101010")
```

Dieses Progremm gibt als ECB-Verschlüsslung den String `011100011100` aus.

*Gegeben sei eine Sequenz $m_1, m_2, \ldots, m_n$ von Klartexten, die mittels einer beliebigen Blockchiffre im ECB oder CBC-Modus verschlüsselt wird, um eine Sequenz von Chiffraten $c_1, c_2, \ldots, c_n$ zu erhalten. Nehmen Sie an, dass bei der Übertragung von $c_1$ ein Fehler passiert (d.h. einige Bits von $c_1$ werden falsch übermittelt). Wieviele Blöcke der Sequenz $m_1, m_2, \ldots, m_n$ werden durch den Empfänger falsch rekonstruiert, wenn der ECB oder CBC-Modus verwendet wird? Begründen Sie Ihre Antwort.*

Wenn der ECB-Modus verwendet wird, dann wird nur $c_1$ falsch entschlüsselt. Die Blocks im ECB-Modus hängen nicht voneinander ab, also werden Übertragungsfehler auch nicht auf sukzessive Blocks weitergegeben. Nur $m_1$ wird also fehlerhaft sein.

Anders sieht das beim CBC-Modus aus. Hier wird ebenso $c_1$ falsch entschlüsselt, gleichzeitig gibt es aber auch einen Bitfehler in $m_2$, da dieser von $c_1$ abhängt (welches falsch ist). Also sind hier $m_1$ falsch sein und $m_2$ wird einige falsche Bits haben (an genau den Stellen, an den $c_1$ falsche Bits hat). Um das zu demonstrieren, kann man ein kleines Ruby-Programm schreiben, was das ganze austestet.

```ruby
require "openssl"

# data (four blocks big)
data = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ;"

# encrypt data
cipher = OpenSSL::Cipher::AES.new(128, :CBC)
cipher.encrypt
key = cipher.random_key
iv = cipher.random_iv
encrypted = cipher.update(data) + cipher.final
orig_enc = encrypted.dup

# mutate first block of cipher
12.times do
  char = rand(16)
  bit  = rand(8)
  encrypted[char] = (encrypted[char].bytes.first ^ (1 << bit)).chr
end

# decrypt data
decipher = OpenSSL::Cipher::AES.new(128, :CBC)
decipher.decrypt
decipher.key = key
decipher.iv = iv
plain = decipher.update(encrypted) + decipher.final

# analyze resulting plaintext
plain.bytes.zip(data.bytes).each_slice(16).each_with_index do |block, count|
  diffs = 0
  block.each do |bytepair|
    diffs += (0..8)
      .map{|bit| bit = (1 << bit); (bytepair[0] & bit) ^ (bytepair[1] & bit) != 0}
      .inject(0){|a,b| if b then a + 1 else a end}
  end
```

```
36    puts "block␣#{count}:␣#{diffs}"
37  end
```

Dieses Programm verschlüsselt ein paar Daten mit AES128. Von den resultierenden Chiffraten werden 12 zufällig ausgewählte Bits verändert. Dann werden diese Entschlüsselt. Die entschlüsselten Daten werden nun, Block-für-Block, mit dem originalen Daten verglichen, und es wird ausgegeben, wie viele Bitfehler entstanden sind. Der Output von diesem Programm sieht so aus:

```
block 0: 64
block 1: 12
block 2: 0
block 3: 0
```

Hier sieht man, dass im ersten Block 64 Bitfehler entstanden sind (man erwartet, dass 50% der Bits fehlerhaft sind, das ist also exakt wie erwartet). Im zweiten Block sind nur noch 12 Bitfehler (das macht ebenso Sinn, da hier nur die Bits falsch sind, die im ersten Chiffratenblock falsch waren). Das Experiment bestätigt also diese Antwort.

## Hausübung 3: Kasiski-Test

*Sie haben einen Ciphertext abgefangen und wissen, dass es sich bei der Verschlüsselung um eine Vigenère-Verschlüsselung handelt. Der Ciphertext hat die folgende Form:*

```
EckOstgloaUclxatrxmfUclxrvkuikugfqwobxvKdfeywtqxpdbcgkw
CcbomsxxrKdfeywtqxpfhcaxvjclkmubtwafqbgzpdmaafbxvzpjxr
```

*Wenden Sie den Kasiski-Test an, um die benutzte Schlüssellänge zu ermitteln. Bearbeiten Sie dafür folgende Schritte.*

*Suchen Sie alle eindeutig doppelt vorkommende N-Gramme (für $N \leq 4$) im Ciphertext und berechnen Sie den Abstand (Position des ersten Auftretens minus die Position des zweiten Auftretens) zwischen beiden gleichen N-Grammen.*

Hier kann man ein kleines Ruby-Programm schreiben, um die N-Gramme und deren Abstand zu finden.

```
1  data = "EckOstgloaUclxatrxmfUclxrvkuikugfqwobxvKdfeywtqxpdbcgkw
2  CcbomsxxrKdfeywtqxpfhcaxvjclkmubtwafqbgzpdmaafbxvzpjxr".split.join
3
4  # find n-grams (4 or longer) and their distance.
5  def ngrams(data, min=4)
6    (1..data.size).map do |offset|
7      (0..data.size-offset)
8        .map{|pos| data[pos] == data[pos+offset]}
9        .chunk(&:itself)
```

```
      .map{|c| [c[0], c[1].length]}
      .inject([[false, 0, 0]]){|c,o| c << [o[0], c[-1][1] + o[1], o[1]]}
      .select{|c| c[0] && c[2] >= min}
      .map{|c| [c[1]-c[2], offset, c[2], data[c[1]-c[2], c[2]]]}
  end.flatten(1)
end

ngrams(data).each do |pos, offset, len, word|
  puts "word␣'#{word}'␣repeated␣with␣offset␣#{offset}␣at␣position␣#{pos}."
end
```

Wenn man dieses Programm ausführt, erhält man zwei N-Gramme.

```
word 'Uclx' repeated with offset 10 at position 10.
word 'Kdfeywtqxp' repeated with offset 25 at position 39.
```

*Bestimmen Sie die Primfaktorzerlegung für alle gefundenen Differenzen.*

Die N-Gramme haben einen Abstand von $10 = 2 \cdot 5$ und $25 = 5 \cdot 5$ Stellen.

*Für den Fall, dass die Wiederholung des N-Gramms nicht zufällig, sondern aufgrund einer Wiederholung eines N-Gramms im Klartext aufgetreten ist, enthält die Primfaktorzerlegung dieser Differenz einen Teiler der Schlüssellänge oder die Schlüssellänge selbst. Vermuten Sie die Länge des verwendeten Schlüssels.*

Wenn man davon ausgeht, dass beiden N-Gramme nicht zufällig auftritt, dann sollte der Schlüssel eine Länge von 5 Buchstaben haben, weil dieser Primfaktor in beiden Zerlegungen vorkommt.

*Nachdem Sie die Schlüssellänge in Aufgabenteil c) bestimmt haben, versuchen Sie nun den Text zu entschlüsseln. Sie konnten außerdem in Erfahrung bringen, dass der zweite Buchstabe im Schlüssel ein `Y` und der Vierte ein `E` ist. Benutzen Sie diese Information um den Schlüssel zu ermitteln und dann den Klartext zu berechnen.*

Der Schlüssel ist `BYTES`. Mit einem simplen Ruby-Funktion und dem Schlüssel kann man die Nachricht entschlüsseln.

```
def decrypt(key, data)
  key = key.chars.map{|c| c.ord - "A".ord}
  data
    .chars
    .map{|c| [c.upcase.ord - "A".ord, c.upcase == c]}
    .zip(key.cycle)
    .map{|d, k| if d[1] then "A".ord else "a".ord end + (d[0] - k) % 26}
    .map(&:chr)
```

```
    .join
end
```

Die entschlüsselte Nachricht lautet: *Der Kasiski Test ist ein Test zum bestimmen der Schluessellaenge. Bei kurzen Schluesseln geht dies gut da sie sich oft wiederholen.*