Praktikum 2 zur Vorlesung IT-Sicherheit

Thema Kryptographie

Für die Durchführung von Krypto-Experimenten nutzen wir OpenSSL. OpenSSL ist eine freie Software, die Implementierungen von Kryptoalgorithmen und Netzwerkprotokollen umfasst sowie ein Programm openssl für die Kommandozeile, mit dem kryptographische Funktionen ausgeführt werden können. Eine ausführliche Anleitung finden Sie unter

https://www.feistyduck.com/library/openssl-cookbook/

In einem späteren Praktikum werden wir OpenSSL noch einsetzen, um Public Key Zertifikate zu erzeugen.

1. Vorbereitung und Einstieg in OpenSSL

Arbeitsverzeichnis und Dateien

Die Experimente führen Sie in einem Arbeitsverzeichnis aus. Legen Sie das Verzeichnis Krypto auf dem Desktop an.

Kopieren Sie die Dateien `brief.txt`, `ueberweisung.txt` in das Arbeitsverzeichnis. Sie finden die Dateien im Verzeichnis `\ITS\ITS_P2\` auf dem Desktop oder im Lernraum der Vorlesung. Die Dateien sind außerdem im Anhang abgedruckt.

Als Arbeitsumgebung halten Sie am Besten mehrere Fenster gleichzeitig geöffnet:

- Eine Kommandozeilensitzung im Arbeitsverzeichnis Krypto, in der Sie OpenSSL-Befehle durchführen.
- Ein Datei-Explorer Fenster, mit dem geöffneten Arbeitsverzeichnis Krypto.
- Ein Editor-Fenster, in dem Sie die jeweils zu bearbeitende Datei geöffnet haben.
- Den Hex-Editor HxD zum Vergleich und zum Ändern von Dateien in Binärform.

OpenSSL

Starten Sie eine Kommandozeilensitzung (cmd), wechseln Sie ins Arbeitsverzeichnis. Im Labor starten Sie OpenSSL durch Aufruf von:

```
"C:\OpenSSL-Win32\bin\openssl.exe"
```

Beim Arbeiten zu Hause den Pfad entsprechend anpassen. Häufig ist openssl.exe in

```
"C:\Programme (x86)\OpenSSL-Win32\bin\openssl.exe"
```

Jetzt sind Sie in der interaktiven Umgebung von OpenSSL. Eine Übersicht über Kommandos und Kryptoverfahren erhalten Sie in OpenSSL durch Eingabe von

```
help
```

Für jedes OpenSSL-Kommando erhalten Sie mit der Option `-help` Informationen zu den Optionen des Kommandos. Probieren Sie

```
prime -help
```

Die `Number of checks` gibt an, wie oft die erzeugte Primzahl mittels des Miller-Rabin Algorithmus auf Primalität geprüft wird.

Wie lautet das Kommando zur Erzeugung einer Primzahl mit 1000 Bit und 20 Miller-Rabin Prüfungen?

_____

Passwort für Praktikum 2

OpenSSL schützt viele Schlüssel und Verschlüsselungen durch Passwörter.

Sie müssen bei den folgenden Übungen ein Passwort sehr oft eintippen.

Um nicht jedes Mal lange zu Suchen und Tippfehlern vorzubeugen, wählen und notieren Sie für dieses Praktikum ein unkompliziertes Passwort für alle Aufgaben.

Folgendes unkomplizierte Passwort wird verwendet: _____

2. Hashwerte und Diffusion

Öffnen Sie die Datei brief.txt im Editor.

Berechnen Sie den Hashwert der Datei brief.txt mit dem Kommando: `dgst brief.txt`

Welcher Hashalgorithmus wird verwendet? _____

Ändern Sie genau ein Zeichen irgendwo im Text der Datei brief.txt und speichern Sie die geänderte Datei.

Berechnen Sie den Hashwert der geänderten Datei und vergleichen Sie die Hashwerte. Was fällt Ihnen auf?

_____

Dieser Effekt der gleichmäßigen Verteilung einer *lokalen* Änderung auf das *ganze* Ergebnis nennt sich ***Diffusion***. Eine gute Diffusion ist eine notwendige Eigenschaft kryptographischer Hashfunktionen.

Überlegen Sie, mit welcher Wahrscheinlichkeit sich im Optimalfall jedes Output**bit** ändern sollte, bei der Änderung eines beliebigen Inputbits?

_____

Machen Sie die Änderung in der Datei brief.txt wieder rückgängig und speichern Sie die Datei.

3. Verschlüsselungen und Manipulationen

OpenSSL leitet bei passwort-basierten Verschlüsselungen aus dem eingegebenen Passwort jeweils mit Hilfe einer ***Key Derivation Function (KDF)*** einen Schüssel für das zu verwendende Verschlüsselungsverfahren (z.B. AES) ab. Damit sich die eingesetzten Schlüssel unterscheiden, geht in die KDF ein zufällig generierter SALT-Wert mit ein. Dieses SALT wird am Anfang der Chiffretext-Datei gespeichert.

Bei Betriebsarten, die einen Initialisierungs-Vektor (IV) erfordern, wird zudem der IV von OpenSSL zufällig generiert.

Wir wollen im Folgenden untersuchen, welche Auswirkungen Änderungen einzelner Zeichen im Chiffretext bewirken. Dazu zwingen wir OpenSSL bei Verschlüsselungen, den gleichen Schlüssel und IV und keinen SALT zu verwenden.

Der SALT wird mit der Option `-nosalt` deaktiviert. Die Option `-iv 0` setzt den IV auf Null und die Option `-p` zeigt den verwendeten Schlüssel an.

3.1 AES Verschlüsselungen im ECB Mode

Öffnen Sie die Datei brief.txt im Editor.

Verschlüsseln Sie die Datei ohne SALT, mit Schlüsselanzeige unter Nutzung Ihres oben gewählten Passworts zur Datei ecb1.txt. (Die IV-Option können Sie weglassen, da der ECB keinen IV erfordert.)

```
aes-128-ecb -p -nosalt -in brief.txt -out ecb1.txt
```

Ändern Sie das „hier“ in der Datei brief.txt auf „Bier“ und speichern Sie die geänderte Datei. Verschlüsseln Sie die geänderte Datei in gleicher Weise zu ecb2.txt. Kontrollieren Sie, dass derselbe Schlüssel verwendet wurde!

Öffnen Sie ecb1.txt und ecb2.txt im Hex-Editor HxD und vergleichen Sie die Dateien manuell (Fenster => Übereinander).

In welchen und wie vielen Bytes unterscheiden sich die die Chiffretexte?

Erklären Sie das Ergebnis auf Grundlage der Funktionsweise des ECB Modes.

Machen Sie die Änderung in der Datei brief.txt wieder rückgängig und speichern Sie die Datei.

3.2 AES Verschlüsselungen im CBC Mode

Jetzt das Gleiche im CBC-Mode.

Verschlüsseln Sie die Originaldatei brief.txt ohne SALT, mit dem IV 0 (Null), mit Schlüsselanzeige unter Nutzung Ihres oben gewählten Passworts zur Datei cbc1.txt.

```
aes-128-cbc –p –nosalt –iv 0 –in brief.txt –out cbc1.txt
```

Ändern Sie das „hier“ in der Datei brief.txt auf „Bier“ und speichern Sie die geänderte Datei. Verschlüsseln Sie die geänderte Datei in gleicher Weise zu cbc2.txt. Kontrollieren Sie, dass derselbe Schlüssel verwendet wurde!

Öffnen Sie cbc1.txt und cbc2.text im Hex-Editor HxD und vergleichen Sie die Dateien.

In welchen und wie vielen Bytes unterscheiden sich die die Chiffretexte?

_____

Erklären Sie das Ergebnis auf Grundlage der Funktionsweise des CBC Modes.

_____

_____

_____

_____

Machen Sie die Änderung in der Datei brief.txt wieder rückgängig und speichern Sie die Datei.

3.3 AES Entschlüsselungen im CBC Mode

Im ECB-Mode werden die Klartextblöcke unabhängig voneinander ver- und entschlüsselt. Eine Änderung eines Zeichens im Chiffretext wirkt sich daher genau auf den betreffenden Klartextblock aus.

Wie ist das im CBC Mode?

Die Datei brief.txt wurde zu cbc1.txt verschlüsselt. Öffnen Sie cbc1.txt im HexEditor und ändern Sie mitten im vierten Block ein Byte. Speichern Sie die geänderte Datei cbc1.txt.

Entschlüsseln Sie cbc1.txt im CBC-Mode (Option –d) zu brief1.txt.

```
aes-128-cbc –d –p –nosalt –iv 0 –in cbc1.txt –out b2.txt
```

Öffnen sie b2.txt mit dem HexEditor. In wie vielen Bytes unterscheidet sich die Datei von brief.txt?

_____ (Hinweis: Die Antwort 16 ist falsch! Schauen Sie genau hin!)

Erklären Sie das Ergebnis auf Grundlage der Funktionsweise der CBC Mode Entschlüsselung.

_____

_____

_____

3.4 Verschlüsselung mit einer Stromchiffre

Öffnen Sie die Datei brief.txt im Editor.

Verschlüsseln Sie die Originaldatei mit der Stromchiffre chacha20 ohne SALT und dem IV 0 (Null) in die Datei cha1.txt.

```
chacha20 -p -nosalt -iv 0 -in brief.txt -out cha1.txt
```

Ändern Sie das „hier“ in brief.txt auf „Bier“, speichern Sie die geänderte Datei und verschlüsseln Sie diese in gleicher Weise zu cha2.txt.

Vergleichen Sie cha1.txt im HexEditor mit cha2.txt.

Worin unterscheiden sich die Dateien und wieso ist das so?

_____

_____

An welcher Eigenschaft (siehe Aufgabe 2 oben) mangelt es also Stromchiffren (im Gegensatz zu Blockchiffren)?

_____

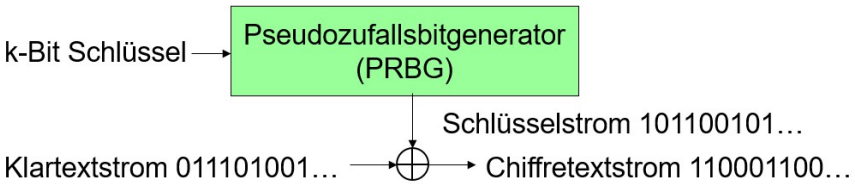
Welche Auswirkung mag es haben, wenn Sie genau ein Bit im verwendeten Schlüssel ändern? Probieren Sie es aus (Option –K, z.B. einmal mit 0 und einmal mit 1 als Schlüssel)

Die mangelnde _____ bezieht sich also nur auf den _____,

nicht auf den _____.

3.5 Fälschung einer Überweisung

ChaCha20 verschlüsselt den Klartext Bit für Bit durch ein XOR mit der Bitfolge des erzeugten Schlüsselstroms.



Wenn ein Bit im Klartext geändert wird, ist daher auch das entsprechende Bit im Chiffretext geändert. Das gilt auch umgekehrt.

Nutzen Sie dieses Wissen, um eine Überweisung zu fälschen.

In der Datei ueberweisung.txt sind Daten einer Überweisung von 100,- EUR auf das Zielkonto mit der IBAN DE88 8888 8888 5555 5555 55 angegeben.

Verschlüsseln Sie die Überweisung mit der Stromchiffre chacha20 wie bei 3.4 zur Datei uw.txt.

```
chacha20 -p -nosalt -iv 0 -in ueberweisung.txt -out uw.txt
```

Stellen Sie sich vor, Sie haben die verschlüsselte Datei uw.txt auf dem Weg zur Bank ***als Man-in-the-Middle*** abgefangen.

Schaffen Sie es, die *verschlüsselte Datei uw.txt* so zu manipulieren, dass nach der Entschlüsselung durch die Bank

- der Überweisungsbetrag von 100,- auf 100100,- EUR erhöht ist und

- die letzten 2 Ziffern im Zielkonto von 55 auf 44 geändert sind?

Welche Bits im Chiffretext müssen Sie hierzu in welcher Weise ändern?

Dabei gehen wir davon aus, dass Sie den Klartext zwar kennen, jedoch nicht manipulieren können. Das ist im Internet häufig der Fall. Z.B. können Sie Webseiten abrufen und wissen bitgenau, wie diese aufgebaut sind. Auch bei http Post-Kommandos, bei denen Parameter an den Server übertragen werden, können Sie das vorher selbst aufzeichnen und kennen daher die Struktur, wenn auch nicht den Parameterwert.

Nachdem Sie es geschafft haben, notieren Sie hier kurz Ihren Lösungsweg:

_____

_____

_____

_____

Der Einsatz von Stromchiffren ist heute im Internet weit verbreitet (z.B. TLS1.3). Wenn nur verschlüsselt würde, wären solche gezielten Manipulationen möglich.

Durch welche zusätzliche Maßnahme werden solche Manipulationen verhindert bzw. erkannt?

---

4. RSA Schlüssel erzeugen

In Aufgabe 5 sollen Sie eine Datei hybrid verschlüsselt und signiert an Ihren Nachbarn übertragen. Dazu benötigen Sie zunächst ein RSA Schlüsselpaar.

4.1 Privaten RSA Schlüssel erzeugen

Lassen Sie sich mit `genrsa –help` die Optionen des Kommandos zum Generieren von RSA-Schlüsseln anzeigen.

Anschließend generieren Sie ein RSA Schlüsselpaar mit einer RSA Schlüssellänge (Modul) von 2048 Bit, dass in der Datei rsapriv.key gespeichert wird.

Der Private Key soll dabei AES 128-Bit verschlüsselt und passwortgeschützt abgelegt werden.

```
genrsa -aes128 -out rsapriv.key 2048
```

Welche Informationen umfasst ein privater RSA-Schlüssel gemäß Vorlesung?

Lassen Sie sich mit folgendem Kommando die Inhalte des privaten Schlüssels anzeigen:

```
rsa -in rsapriv.key -text
```

Hinweis: Die Zahlen exponent1, exponent2 und coefficient sind vorberechnete Zahlen, die zur Beschleunigung der Entschlüsselung bzw. Signaturberechnung dienen.

Welcher öffentliche Exponent wird verwendet? _____

4.2 Öffentlichen RSA-Schlüssel extrahieren

Der Public Key wird mit folgendem Kommando aus dem Private Key extrahiert und in der Datei rsapub.key gespeichert:

```
rsa -in rsapriv.key -pubout -out rsapub.key
```

Welche Informationen umfasst ein öffentlicher RSA-Schlüssel?

---

5. Hybride Verschlüsselung mittels RSA und AES sowie RSA-Signierung

Jetzt sollen Sie eine Datei hybrid verschlüsseln und digital signieren, so dass Ihr Nachbar die Daten entschlüsseln und die Signatur verifizieren kann.

Mit welchem RSA-Schlüssel welcher Person verschlüsseln Sie? Bitte ankreuzen:

O Private Key O Public Key O Sender (Sie) O Empfänger (Nachbar)

Mit welchem RSA-Schlüssel welcher Person erfolgt Ihre Signierung? Bitte ankreuzen:

O Private Key O Public Key O Sender (Sie) O Empfänger (Nachbar)

Damit Sie und Ihr Nachbar Schlüssel und Dateien auseinanderhalten können, werden Dateinamen ab sofort mit einem Kürzel (2 Großbuchstaben) ergänzt.

Mein Kürzel zur Identifikation meiner Dateien ist: ____ ____ (2 Großbuchstaben)

Achten Sie darauf, dass Sie und Ihr Nachbar **unterschiedliche Kürzel** haben!!

Im Folgenden werden bei Dateinamen allgemein die Kürzel XX und YY angegeben. Diese sollten Sie durch Ihr bzw. dem Kürzel Ihres Nachbarn ersetzen.

Zur besseren Übersicht sind die verwendeten Dateinamen nachstehend angegeben:

testXX.pdf	Ihre zu verschlüsselnde und zu signierende Originaldatei
testencXX.pdf	Die von Ihnen verschlüsselte Originaldatei
skeyXX.key	Der von Ihnen verwendete Session Key
sencXX.key	Ihr RSA-verschlüsselter Session Key
rsapubYY.key	Der öffentliche RSA-Schlüssel Ihres Nachbarn
sigXX.txt	Die RSA/SHA256 Signatur Ihrer Originaldatei

5.1 Session Key für die AES Verschlüsselung erzeugen

Zunächst erzeugen Sie sich einen Session Key für die AES Verschlüsselung. Für einen max. 256 Bit langen AES-Schlüssel benötigen wir 32 Byte:

```
rand -hex -out skeyXX.key 32
```

(Für jede Verschlüsselung soll ein neuer SessionKey gewählt werden.)

5.2 AES-Verschlüsselung einer pdf-Datei mit dem Session Key

Wählen Sie eine nicht zu große pdf-Datei (einige kb), die Sie verschlüsseln und signieren möchten. Die Datei wird im Folgenden `testXX.pdf` genannt.

Nutzen Sie den SesseionKey um die Datei testXX.pdf zu verschlüsseln.

```
enc -aes256 -in testXX.pdf -out testencXX.pdf -pass file:./skeyXX.key
```

(Sie verschlüsseln IHRE Datei mit IHREM Session Key.)

5.3 Übertragung Ihres öffentlichen RSA-Schlüssels an Ihren Nachbarn

Benennen Sie Ihren Public Key rsapub.key unter Nutzung Ihres Kürzels um in rsapubXX.key.

Anschließend geben Sie die Datei rsapubXX.key an Ihren Nachbarn weiter. Per E-Mail oder USB-Stick oder …

5.4 RSA-Verschlüsseln des Session Keys

Jetzt verschlüsseln Sie Ihren SessionKey mit dem öffentlichen RSA-Schlüssel, den Sie von Ihrem Nachbarn erhalten haben:

```
rsautl -encrypt -inkey rsapubYY.key -pubin -in skeyXX.key -out sencXX.key
```

5.5 Signieren der Originaldatei

Signieren Sie die Datei testXX.pdf mit Ihrem Private Key. Ihre Signatur speichern Sie in der Datei sigXX.txt. Die Signatur erfolgt dabei durch Bildung eines Hashwerts (SHA256) über die Daten und Anwendung des RSA-Verfahrens mit dem Privaten Schlüssel auf den Hashwert.

```
dgst -sha256 -sign rsapriv.key -out sigXX.txt testXX.pdf
```

5.6 Übertragung der notwendigen Dateien an Ihren Nachbarn

Überlegen Sie, welche Dateien Ihr Nachbar VON IHNEN benötigt, damit dieser die hybride Verschlüsselung rückgängig machen kann UND die Signatur prüfen kann.

Übertragen Sie die Dateien an Ihren Nachbarn.

5.7 Entschlüsselung des verschlüsselten Session Keys

Sie haben den verschlüsselten Session Key in der Datei sencYY.key von Ihrem Nachbarn erhalten. Entschlüsseln Sie die Datei mit Ihrem privaten RSA Schlüssel:

```
rsautl -decrypt -inkey rsapriv.key -in sencYY.key -out skeyYY.key
```

5.8 Entschlüsselung der pdf-Datei

Mit dem SessionKey können Sie dann die von Ihrem Nachbarn erhaltene verschlüsselte pdf-Datei `testencYY.pdf` entschlüsseln:

```
enc -d -aes256 -in testencYY.pdf -out testYY.pdf -pass file:./skeyYY.key
```

Prüfen Sie, ob die Entschlüsselung geklappt hat, in dem Sie sich die pdf-Datei anzeigen lassen.

Falls die entschlüsselte Datei nicht der Originaldatei entspricht, suchen Sie den Fehler und wiederholen Sie die Schritte.

5.9 Prüfung der Signatur

Mit der entschlüsselten Originaldatei `testYY.pdf` und dem öffentlichen Schlüssel Ihres Nachbarn können Sie jetzt die Signatur prüfen.

```
dgst -sha256 -verify rsapubYY.key -signature sigYY.txt  testYY.pdf
```

Falls die Signaturprüfung fehlschlägt, wiederholen Sie die Signierung.

Optional und nur im Labor

6. RSA einmal selbst mit (recht) großen Zahlen ausprobieren

Starten Sie in Virtual Box die AxiomXP VM. Mit dem Doppelklick auf Axiom öffnet sich die interaktive Oberfläche des Computeralgebrasystems Axiom.

Wir testen das RSA-Verfahren mit 512 Bit Modullänge (= Schlüssellänge). Beachten Sie, dass diese Modullänge heute nicht mehr ausreichend sicher ist.

6.1 Berechnung der Parameter für das RSA-Verfahren

Wir berechnen wir zunächst

```
a:= 2**256
```

Anschließend addieren wir zu a eine zufällig gewählte lange Zahl:

```
a:= a + 346203460836412080523…..
```

Jetzt erzeugen wir unsere erste Primzahl p

```
p:= nextPrime(a)
```

Wir erhöhen a noch einmal zufällig:

```
a:= a + 120426812312080523…..
```

und erzeugen dann unsere zweite Primzahl q

```
q:= nextPrime(a)
```

Aus p und q erhalten wir als Produkt den RSA-Modul m

```
m:=p*q
```

Die Bitlänge von m können wir über den Logarithmus zur Basis 2 prüfen:

```
log2(m::Float)
```

Phi(m) erhalten wir als Produkt von (q-1) und (p-1)

```
phim:=(q-1)*(p-1)
```

Als öffentlichen Exponenten e wählen wir 2**16+1

```
e:=2**16+1
```

Wir prüfen, ob e teilerfremd zu Phi(m) ist.

```
gcd(e,phim)
```

(Falls das nicht der Fall ist, sollten wir andere Primzahlen wählen.)

So, jetzt wird es spannend. Der private Exponent d ist zu bestimmen als multiplikatives Inverses von e modulo Phi(m). Das geht mit dem erweiterten Euklidischen Algorithmus.

```
extendedEuclidean(e,phim)
```

Das Ergebnis hat zwei Koeffizienten und einen Generator.

Es gilt: coef1 * e + coef2 * phim = generator

Damit ist coef1 des Ergebnisses unser gesuchter privater Exponent d.

```
d:=extendedEuclidean(e, phim).coef1
```

(Falls d negativ sein sollte, müssen wir d um Phi(m)erhöhen, um zum kleinsten nichtnegativen Rest modulo Phi(m) zu kommen. `d:=d+phim`)

Wir prüfen, ob d das Inverse von e modulo Phi(m) ist:

```
(e*d) rem phim
```

(*rem* steht dabei für Remainder, entspricht also der Restfunktion *mod*.)
Das sollte jetzt 1 ergeben. Falls nicht wiederholen Sie die Schritte und suchen Sie den Fehler.

Jetzt sind alle notwendigen Parameter für die RSA-Verfahren ausgewählt.

6.2 RSA-Verschlüsselung und RSA-Entschlüsselung

Wir wählen einen Klartext als gut wieder erkennbare ganze Zahl < m. Z.B.

```
klartext:=1000200030004000500060007000800090000
```

Der RSA-Chiffretext c ergibt sich dann als c = (klartext ** e) mod m.

Zur Potenzierung verwenden wir die Funktion *powmod*, die nach jeder modularen Multiplikation das Ergebnis wieder modulo m reduziert.

```
c:=powmod(klartext,e,m)
```

Die Entschlüsselung erfolgt durch Potenzierung von c mit dem privaten Exponenten d:

```
powmod(c,d,m)
```

Das sollte jetzt wieder den Klartext ergeben.

6.3 Weshalb nach jeder Multiplikation modulo *m* reduzieren?

Weshalb sollte für die Ver- und Entschlüsselung nicht die normale Exponentiation mit ** genutzt werden?

Nun, die Bitlänge des Ergebnisses erhöht sich bei jeder Multiplikation und verdoppelt sich bei jeder Quadrierung!

Bezeichne n:=log2(klartext::Float) die Bitlänge des Klartextes. Für den Klartext von oben wären das n =116 Bit. Das Quadrat des Klartextes hat dann schon die Bitlänge 2*n, der Klartext hoch 3 die Bitlänge 3*n, usw.

Die Zahl klartext**e hätte dann die Bitlänge 65537*n, das wären im Beispiel über 7,6 Mio. Bit! Speicherbedarf der Zahl ca. 1 MByte. Eine ganz schön große Zahl.

Bei der Entschlüsselung ist es noch viel schlimmer. Der Chiffretext c hat nämlich i.d.R. mehr als 500 Bit. Die Zahl c**d hat damit d*500 Bit. Der geheime Exponent d ist aber selbst eine Zahl in der Größenordnung 2**500. Also hat die Zahl c**d ungefähr 500*2**500 Bit, also ca. 2**510 Bit oder 2**507 Byte. Damit könnte c**d in keinem Computer gespeichert werden! 2**40 Byte sind ein TeraByte. Zur Speicherung von c**d wären also ca. 2**460 TeraByte notwendig.

Nochmal anders: Die Zahl c, die i.d.R. mehr als 500 Bit hat, müsste ca. 500 mal quadriert werden. Sie erinnern sich an die Anekdote mit dem Schachbrett und dem Reis: Ein Reiskorn auf das erste Feld, 2 auf das zweite und von Feld zu Feld jeweils doppelt so viele. Hier legen wir nicht eine, sondern c als >500 Bit Zahl auf das erste Feld und unser Schachbrett hat nicht 64, sondern 500 Felder.

Anhang

Inhalt der Datei brief.txt

```
Hallo Oma,

ist bei Euch auch so schoenes Wetter wie hier in Osnabrueck?

Ich experimentiere gerade im IT-Sicherheits-Praktikum
im Rahmen meines Studiums.

Liebe Gruesse und alles Gute von

Deinem Enkel
```

Inhalt der Datei ueberweisung.txt

```
Auftraggeber: Frau Ganzreich
IBAN Auftraggeber: DE12 3456 7890 9876 5432 11
Überweisungsbetrag: 0000100,00 EUR
Empfaenger: Herr Rechtarm
IBAN Empfaenger: DE88 8888 8888 5555 5555 55
```