Kryptosysteme

von
Daniel Vander Putten
Matrikel-Nr. 26203056

Betreuer:
Dr. Wolfram Koepf
Universität Kassel
Fachbereich 10

Volkmarsen
27.02.2013

Inhalt

0 Einführung

Motivation

Der Zweck von Kryptosystemen ist die Sicherung von Informationen gegen aktive Angriffe von Personen die nicht am Transfer beteiligt sind. Im Digitalen Zeitalter, in dem sich Nachrichten überall, in kürzester Zeit versenden lassen, sind unzählige Kryptosysteme im Einsatz. Generell gibt es zwei mögliche Angriffsziele, vor denen sich, durch Kryptosysteme, versucht wird zu schützen. Der einfache Angriff beinhaltet lediglich die Idee die Nachricht, als die nicht an der Kommunikation beteiligte Person, mitzulesen bzw. diese abzuhören. Allerdings gibt es auch die Möglichkeit, dass der Angreifer nicht nur mithören will, sondern aktiv die Nachricht verändern will. Ein einfaches Beispiel hierfür wäre eine Banküberweisung, bei der jemand den Betrag ändern möchte, um eine der beiden Parteien zu begünstigen.

Grundfunktion

Der Einsatz eines Kryptosystems funktioniert wie folgt, der Sender wendet den Sende-Algorithmus des Systems an und bekommt eine abgeänderte Nachricht. Die durch den Algorithmus abgeänderte Nachricht wird nun an den Empfänger gesendet. Der Empfänger wiederum muss nun den passenden Algorithmus anwenden um die Nachricht wieder verwenden bzw. lesen zu können. Fängt nun jemand die Nachricht ab und will sie lesen müsste er ebenfalls erst den zugehörigen Algorithmus anwenden, denn die Nachricht die übertragen ist, ist normal nicht lesbar und somit ist auch eine Änderung der ursprünglichen Nachricht nicht sicher möglich.

Definitionen und Formulierungen

Hier werden einige Grundbegriffe in Verbindung mit Kryptosystemen eingeführt, die im folgenden verwendet werden.
Ein Reintext oder Plaintext ist eine Nachricht, die sowohl Sender als auch Empfänger direkt verwenden bzw. lesen können.
Eine Verschlüsselung oder Kodierung ist der Algorithmus, den der Sender anwendet, um den Reintext unkenntlich zu machen.
Diesen „unkenntlich“ gemachten Reintext nennt man Chiffre oder Chiffrat.
Der Empfänger der Nachricht muss nun wiederum eine Entschlüsselung oder Dekodierung vornehmen, um so wieder den Reintext aus der Chiffre zu bekommen.
Damit nun nicht jeder beliebige die Entschlüsselung vornehmen kann, gibt es ein weiteres Element welches man Schlüssel nennt. Ein Schlüssel ist im kryptographischen Sinne ein Objekt welches zum Verschlüsseln und Entschlüsseln gebraucht wird, damit der Algorithmus richtig arbeitet.

Weitere Grundgedanken

Zuvor war vom Schlüssel als Objekt die Rede. Wie genau der Schlüssel nun in einzelnen Systemen aussieht kann stark variieren, von Zahlen bis hin zu materiellen Dingen, die durch eine spezielle Spezifikation eine Schlüsseleigenschaft aufweisen, genau wie ein Schlüssel, der ein normales Schloss öffnet. Es stecken zwar oft mathematische Verfahren dahinter, wie man den Reintext und den Schlüssel kombinieren muss, allerdings gibt es auch Kryptosysteme für deren Anwendung keine Rechnungen nötig sind.
Im großen und ganzen werden zwei allgemeine Arten von Verfahren unterschieden: symmetrische und unsymmetrische. Unter symmetrischen Verfahren versteht man Kryptosysteme, welche sowohl zum Verschlüsseln als auch zum Entschlüsseln den gleichen Schlüssel verwenden. Wie man schnell sieht, hat diese Art von Systemen eine entscheidende Schwäche, welche in der Übertragung oder Einigung auf einen Schlüssel liegt.

Bei dem asymmetrischen System liegt dieses Problem nicht vor. Diese Systeme verwenden unterschiedliche Schlüssel, meistens einen öffentlichen

zum Verschlüsseln und einen geheimen, privaten zum Entschlüsseln, so kann jeder mit dem öffentlichen Schlüssel Nachrichten verschlüsseln und nur derjenige mit dem geheimen Schlüssel kann diese lesen. Der Nachteil bei dieser Art Verfahren ist, das sie aufwendige mathematische Operationen erfordern und deswegen im Verhältnis zum symmetrischen System wesentlich länger für Verschlüsselung und Entschlüsselung brauchen. Eine gebräuchliche Methode ist deswegen die Kombination beider Verfahren. Man nimmt ein symmetrisches Verfahren zum allgemeinen Datenaustausch. Die Schlüssel für dieses symmetrische Verfahren werden als Chiffrat eines asymmetrischen Systems ausgetauscht. So muss zwar Schlüsselaustausch mit vergleichsweise hoher Rechenzeit akzeptiert werden, jedoch muss dies ja nur einmal gemacht werden und es kann denn, mit den sicher übertragenen Schlüsseln des symmetrischen Systems, mit geringerem Rechenaufwand kommuniziert werden.

Aufbau

Dieses Dokument ist zwar auch gedruckt zu verstehen und verwendbar, jedoch liegt einer der Grundgedanken bei der Konzipierung auf Funktionalität. Kryptographie ist eine Anwendung der Mathematik und somit soll hier nicht nur Theorie betrieben werden, sondern man kann in dem Dokument auch aktiv die Algorithmen nutzen um die Kryptosysteme auszuprobieren. Um sich hierfür nicht zeitaufwendig mit Datenstrukturen befassen zu müssen, sind im Anhang selbst noch weitere Algorithmen die alle notwendigen Datenstrukturänderungen ausführen. Weiterhin ist jeder Algorithmus selbst mit wenigstens einem Beispiel zu Verschlüsselung und Entschlüsselung ergänzt, mit dem das selbst ausprobieren leichter gemacht werden soll.

1 Historische Systeme

Historische Kryptosysteme eignen sich gut den Grundgedanken der Kryptographie zu veranschaulichen, da sie selbst ohne aufwendige technische Möglichkeiten und Berechnungen eine sichere Datenübertragung gewährleisten können. Sicher in diesem Aspekt bedeutet speziell, das die Art des Systems unbekannt vom Angreifer sein sollte. Durch diese sehr einfach konzipierten Systeme, lässt sich auch mit wenig Aufwand ein Angriff durchführen, wenn man weiß wie es funktioniert.

1.1 Skytala

Schon den alten Griechen sagt man nach, sie verwendeten Kryptographie um ihre Nachrichten vor den Spartanern geheim zu halten. Ihre Nachrichten wurden hierfür in einer speziellen Weise aufgeschrieben und man musste die gleiche Situation wieder nachstellen, um sie lesen zu können. Es wird ein beschreibbarer Materialstreifen mit der Breite eines Zeichens um einen Gegenstand gewickelt. Dieser Gegenstand war wahlweise ein Prisma oder ein zylinderartiger Gegenstand. Die Wicklung ist so vorzunehmen, dass der Materialstreifen Rand an Rand um diesen Gegenstand gewickelt ist. Die Nachrichten werden nun Zeichen für Zeichen nebeneinander also über die einzelnen Wickelungen geschrieben. Zieht man den Materialstreifen nun nach dem Aufschreiben der Zeichen ab, bleibt lediglich ein Materialstreifen mit zuordnungsbaren Zeichen, die anscheinend untereinander geschrieben sind.

(http://upload.wikimedia.org/wikipedia/commons/thumb/5/51/Skytale.png/199px-Skytale.png)

Kodieren

Der zu verschlüsselnde Reintext wird Zeile für Zeile um den zylinderartigen Gegenstand geschrieben, so dass die einzelnen Zeichen immer über den Rand des Materialstreifens in die nächste Spalte geschrieben werden. Zieht man nun den Materialstreifen vom zylinderförmigen Gegenstand ab, bleibt der einzelne Streifen mit einer untereinander stehender Zeichenkette, welche normalerweise keinen Sinn ergibt. Diese Zeichenkette ist die Chiffre und wird übertragen. Die Übertragung im historischen Kontext war die Übergabe des Materialstreifens an den Empfänger.

Schlüssel

Der Schlüssel in diesem Kryptosystem ist der zylinderförmige Gegenstand. Hierfür eignen sich Zylinder im eigentlichen Sinne, jedoch auch n-eckige Prismen. Der eigentliche Schlüsselaspekt hierbei ist der Umfang der Wickelfläche. Die Textzeilen, die der Sender im Reintext angab, ist deswegen ein Zeilenformat geworden, weil der Abstand auf dem Materialstreifen immer gleichlang war. Dieser Abstand bzw. der Umfang des Gegenstandes ist somit Schlüsselaspekt und wichtig um die Zeilen genauso wieder zu bekommen wie der Sender sie auch hatte. Der Vorteil eines Prismas ist hierbei, dass die Kanten immer einen Zeilenunterschied aussagen können und die Kodierungsarbeit, sowie die Leserlichkeit der Nachricht gefördert werden kann.

Dekodieren

Die Dekodierung bei Skytala geht schematisch gleich zur Kodierung selbst. Man nimmt den empfangenen Materialstreifen und wickelt ihn wieder Rand an Rand um einen Gegenstand, der gleiche Schlüsseleigenschaft hat, wie der Gegenstand, der zur Kodierung benutzt wurde. Wenn nun der Umfang des Gegenstandes ausreichend genau dem entspricht, wie der Umfang den der Sender hatte, werden sich die gleichen Zeilen Zuordnungen wieder zusammen setzen, wie der Sender sie beim Schreiben der Nachricht auch hatte. Zu beachten hierbei ist, dass der Umfang nicht exakt übereinstimmen muss. Je nach Zeichengröße sind Abweichungen von ein paar Millimetern zulässig.

Mathematische Verallgemeinerung

Der Gegenstand mit Schlüsselaspekt wird hierfür durch eine Matrix mit *N* Zeilen und *M* Spalten ausgetauscht. Somit wird die Nachricht wieder zeilenweise in die Matrix geschrieben und um die Chiffre zu erhalten, wird die Matrix spaltenweise ausgelesen. Mit dem Matrixprinzip lässt sich das Verfahren nun auf 2 Arten ausdrücken. Einmal kann man zum Entschlüsseln die Chiffre zeilenweise in die transponierte Matrix eintragen; diese Variante wird im Beispiel gezeigt. Das Lesen der dekodierten Nachricht würde hier jetzt spaltenweise in der transponierten Matrix geschehen. Die andere Art wäre das Eintragen der Chiffre in die gleiche Matrix aber diesmal spaltenweise. Hierbei wäre das Auslesen wieder zeilenweise. Der Schlüssel, bei der Verschlüsselung mit einer Matrix, reduziert sich auf die Abmessung der Matrix bzw. die Zahlen *N* und *M* die sie charakterisieren.

Beispiel

In diesem Beispiel wird ein 32 Zeichen Text mit einer 8x4 Matrix per Skytala verschlüsselt. Hierbei wird die Variante gewählt, bei der zum Dekodieren die transponierte Matrix verwendet wird. Weiterhin zu beachten ist, dass der Text hierbei keine Leerzeichen enthält, weder mittendrin noch am Ende um die Textlänge faktorisierbar zu machen.

Reintext:

KRYPTOGRAPHIEMACHTUNSALLENFREUDE

Kodieren:

```
K R Y P T O G R
A P H I E M A C
H Z U N S A L L
E N F R E U D E
```

Chiffre Text:

KAHERPZNYHUFPINRTESEOMAUGALDRCLE

Dekodieren:

```
K A H E
R P Z N
Y H U F
P I N R
T E S E
O M A U
G A L D
R C L E
```

Wieder erhaltener Reintext:

KRYPTOGRAPHIEMACHTUNSALLENFREUDE

Angriff auf Skytala

Ciphertext-only Angriff

Mit dem Begriff des "Ciphertext-only Angriff" ist ein Angriff gemeint, bei dem ausschließlich eine Chiffre vorliegt und weiterhin das System bekannt ist.
Unter der Annahme, das man die Chiffre abgefangen hat, besitzt man den Chiffretext, so wie die Zeichenlänge des Chiffretextes L. Da die Nachricht komplett in der N x M Matrix eingeschrieben wird gilt:
$\boldsymbol{N * M \geq L}$
somit müssen die Faktoren der Länge L gefunden werden.
Zu berücksichtigen ist, dass es hierbei sowohl mehrere Kombinationen von N und M geben kann, als auch, dass $N * M \neq M * N$, da es sich um Matrizen handelt. Weiterhin ist zu berücksichtigen, das sich die Textlänge L nicht komplett in der Matrix befinden muss, es ist durchaus zulässig einzelne Zeichenpositionen freizulassen. Ob dies nun durch ein grammatikalisches Leerzeichen entsteht oder einfach das letzte Zeichen freigelassen wird, ist egal.
Im Beispiel oben gilt $N * M = L$ jedoch ist durch Leerzeichen auch möglich, dass gilt $M * N = L - x$. Jedoch ist es unwahrscheinlich, dass x sonderlich groß ist, weiterhin würde es sehr schnell auffallen, falls x größer als N oder M wäre, da somit ganze Leerzeilen oder Leerspalten auftreten.

Algorithmus von Skytala

Dieser Algorithmus verwendet die oben beschriebene Idee der Bearbeitung in einer Matrix. Input hierfür ist *text*, welcher eine beliebige Zeichenkette sein kann. Diese Zeichenkette wird erst Zeichen für Zeichen aufgetrennt und gemäß der Verschlüsselungsvorschrift werden dann die einzelnen Zeichen zu einer neuen Zeichenkette zusammengesetzt. Hierfür ist der Input *N*, *M* wichtig, womit ausgedrückt wird wie lang eine Zeile der Matrix wäre und wie viele Zeichen untereinander geschrieben werden würden, bevor sie wieder in der gleichen Zeile stehen. Zu beachten hierfür ist, dass die Zeichenkettenlänge genau *N***M* ergeben muss. Die Entschlüsselung per Algorithmus ist durch die Symmetrie der Verschlüsselungsvorschrift eine einfache Vertauschung von *N* und *M*. Falls die Textlänge nicht Faktorisierbar ist, wird Ergänzen durch einige Leerzeichen empfohlen, denn auch Leerzeichen sind Textzeichen.

Eingaben des Skytala Algorithmus sind *text* – der zu kodierende Reintext als Zeichenkette, *N* – die Anzahl der Zeilen und *M* – die Anzahl der Spalten. Skytala liefert eine verschlüsselte Zeichenkette gleicher Länge. Ein Dekodierungs-Algorithmus wird nicht gebraucht, hierfür ist einfach *N* und *M* zu vertauschen.

```
Skytala[text_, N_, M_] :=
 Module[{T = Characters[text], out = {}, i, n, m},
  For[m = 0, m < M, m++,
   For[n = 0, n < N, n++,
    AppendTo[out, T[[n * M + m + 1]]]
   ]
  ];
  StringJoin[out]
 ]
```

Beispiele per Algorithmus

```
Skytala["KRYPTOGRAPHIEMACHTUNSALLENFREUDE",
 8, 4]
```

```
KTAEHSEEROPMTANUYGHAULFDPRICNLRE
```

```
Skytala["KTAEHSEEROPMTANUYGHAULFDPRICNLRE",
 4, 8]
```

```
KRYPTOGRAPHIEMACHTUNSALLENFREUDE
```

```
Skytala[
 "Dies ist ein Beispiel mit Leerzeichen
   auch am Ende. ", 4, 13]
```

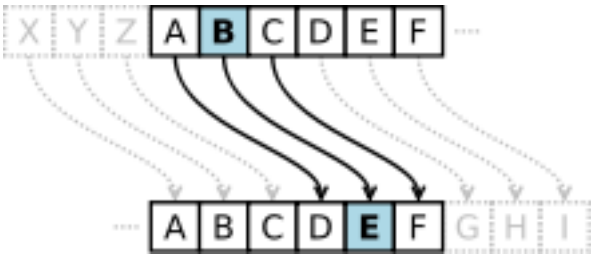
```
DBLuieeceiehssr
  pzaiiemsei tlcE  hnemediinent .  a
```

```
Skytala[
 "DBLuieeceiehssr  pzaiiemsei tlcE
   hnemediinent .  a ", 13, 4]
```

```
Dies ist ein Beispiel
  mit Leerzeichen auch am Ende.
```

1.2 Caesar Chiffre

Das Kryptosystem namens Caesar Chiffre geht auf den Römischen Herrscher Gaius Julius Caesar zurück. Auch dieses System wurde ebenfalls, wie Skytala, zur Verschlüsselung von Militärinformationen gegen Feinde entwickelt und eingesetzt. Man verwendet hierfür eine feste und geordnete Menge an Schriftzeichen, während in Skytala grundsätzlich irrelevant war welche Zeichen eingesetzt worden sind, weil die Zeichen selber nicht verändert wurden, sondern nur deren Anordnung. Hierbei werden die Zeichen selbst verändert, somit tauchen in der Chiffre nicht mehr die gleichen Zeichen in gleicher Auftrittshäufigkeit auf. Solch eine geordnete Schriftzeichenmenge wird ab jetzt als Alphabet benannt. Selbstverständlich kann ein Alphabet genügend groß sein, um eine beliebig große Menge an Zeichen zu enthalten. Ein Beispiel für ein solch großes Alphabet ist die ASCII Tabelle.
Der Grundsatz des Systems basiert auf einer Verschiebung der Zeichenindizes. In der ursprünglichen Variante wurden die Zeichen um drei Positionen nach hinten verschoben, jedoch sind auch andere Verschiebungen möglich. Wenn man durch die Verschiebung über den letzten Buchstaben hinaus gelangen würde fängt man wieder vorne an.



(http://upload.wikimedia.org/wikipedia/commons/thumb/2/2b/Caesar3.svg/220px-Caesar3.svg.png)

Kodieren

Zum Kodieren wird jedem Zeichen ein Index zugewiesen. Man könnte im Alphabet der Großbuchstaben A den Index 1 zuweisen B = 2 usw. bis Z = 26. Dieses Alphabet hat 26 Zeichen und 26 Zeichenindices. Anschließend wird jedes Zeichen einzeln durch seinen Index ausgetauscht. Diese Zeichenindices werden um eine Anzahl an Positionen in eine Richtung verschoben. Die neu erhaltenen Zeichenindices werden zum Übertragen wieder durch ihre zugehörigen Zeichen ersetzt. Selbstverständlich ist die Zuordnung der Zeichenindices nur theoretisch notwendig, wer dieses System verstanden hat, wird die Zeichen direkt austauschen. Weiterhin ist es üblich, diese Zuordnung längere Zeit bei zu behalten, da sie sowohl dem Sender als auch dem Empfänger bekannt sein muss.

Schlüssel

Der Schlüssel hierbei ist die Anzahl der Positionen, um die die Zeichenindices verschoben wurden. Die Richtung in die verschoben wird, wird nach Konvention grundsätzlich nach hinten gesetzt. Der Grund für diese Konvention ist, dass es für jede Verschiebung nach vorne eine passende Zuordnung nach hinten gibt. Nimmt man ein Alphabet mit N Zeichen und man wolle die Verschiebung um X Positionen nach vorne durchführen, entspricht dies einer Verschiebung von (N-X) nach hinten, dies nennt man den komplementären Schlüssel. Die Einigung auf ein Alphabet gehört normalerweise nicht zum Schlüsselaspekt selbst, muss jedoch als Teil der Einigung des Kryptosystems beiden Parteien bekannt sein.

Dekodieren

Um die Chiffre wieder zum Reintext zu entschlüsseln, werden nun genau wie beim Verschlüsseln wieder die Zeichenindices, einzeln um die gleiche Anzahl an Positionen, in die entgegengesetzte Richtung verschoben. Es ist, wie oben beschrieben, ebenfalls möglich den komplementären Schlüssel zu bilden, falls man die Richtung beibehalten möchte.

Mathematische Verallgemeinerung

Jedem Zeichen wird eine Zahl zugeordnet, bei einem allgemeinen Alphabet nimmt man *N* Zeichen an. Als Spezialfall hier wird ein Alphabet, das die 26 Lateinischen Buchstaben enthält, betrachtet. Dieses ist für die Übertragung beliebiger Nachrichten ohne Formatierung ausreichend. Fasst man diese Zahlen jetzt als Restklassenring der ganzen Zahlen auf, entspricht das Kodieren einer Modulo Addition und das Dekodieren einer Modulo Subtraktion. Somit lassen sich folgende Funktionen zum Verschlüsseln und Entschlüsseln definieren.

$\text{encrypt}_K(P) = (P + K) \bmod N$

$\text{decrypt}_K(C) = (C - K) \bmod N$

mit P als Reintext (Plaintext), K als Schlüssel (Key) , C als Chiffre und N als Anzahl der Alphabetelemente

Beispiel

In diesem Beispiel wird die Nachricht "kryptographiemachtunsallenfreude" mit der Caesar Chiffre verschlüsselt. Das gewählte Alphabet sind hierbei die Kleinbuchstaben in Lexikographischer Ordnung mit 26 Zeichen. Der Schlüssel wird auf 7 gesetzt. Das dritte Zeichen zeigt bei der Verschlüsselung die Modulo Berechnung, wenn man es aus dem Alphabet herausschiebt.

Reintext

"kryptographiemachtunsallenfreude"

Kodieren

k ⟺ 11; 11+7=18; 18 ⟺ r
r ⟺ 18; 18+7=25; 25 ⟺ y
y ⟺ 25; 25+7=32 mod 26=6; 6 ⟺ f
...

Chiffre Text

"ryfwavnyhwoplthjoabuzhsslumylbk"

Dekodieren

r ⟺ 18; 18-7=11; 11 ⟺ k
y ⟺ 25; 25-7=18; 18 ⟺ r
...

Wieder erhaltener Reintext

"kryptographiemachtunsallenfreude"

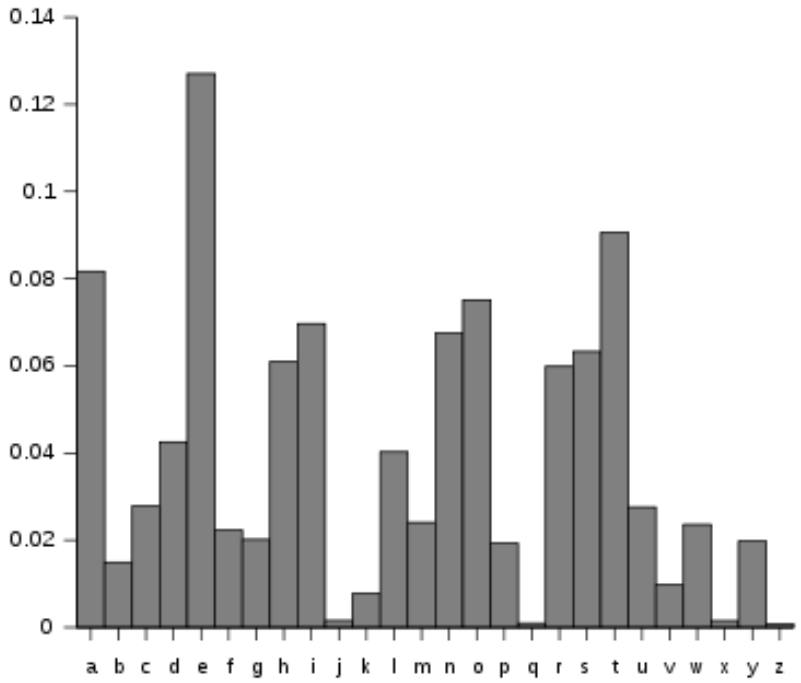
Angriff auf die Caesar Chiffre

Statistische Buchstabenverteilung

Jede Sprache hat eine Buchstabenverteilung bei der unterschiedliche Buchstaben verschieden oft auftreten. Vergleicht man die Buchstabenverteilung des Chiffre Textes mit der Verteilung der Ursprungssprache, werden gewisse Buchstaben gleich häufig auftreten. Es besteht also eine hohe Chance, das diese gleich oft auftretenden Buchstaben vertauscht wurden. Im Caesar-System, bei dem alle Buchstaben um den gleichen Schlüssel verschoben sind, wird beim Chiffre Text sogar die ganze Alphabet Verteilung (zwar Verschoben), aber mit gleichem Schema wieder auftauchen. Zählt man diese Verschiebung ab, erhält man den Schlüssel. Das Verfahren eignet sich jedoch auch für Variationen der Caesar Chiffre, bei dem Buchstaben um unterschiedliche Distanzen verschoben werden. In diesem Fall würde man die Häufigkeit der einzelnen Zeichen mit der Häufigkeit der Zeichen der Ursprungssprache abgleichen. Buchstaben, die ähnliche Auftrittshäufigkeit ausweisen, könnten deswegen vertauscht worden sein.

Zu bemerken ist hierbei, dass dies gut trennbare Auftrittshäufigkeiten der Zeichen voraussetzt. Somit eignet sich dieser Ansatz nur, wenn man eine ausreichend große Nachricht hat, bei der die einzelnen Zeichen ihre Auftrittshäufigkeit auch einnehmen. Weiterhin ist der Vergleich mit einer generellen Sprachverteilung nicht grundsätzlich zielführend, da in einem speziellen Sprachrahmen (z.b. Militärkommunikation) die Auftrittshäufigkeit der Zeichen zur Grundsprache abweichen kann.

Diese Grafik zeigt die Buchstabenverteilung in der englischen Sprache



(http://upload.wikimedia.org/wikipedia/commons/thumb/d/d5/English_letter_frequency_%28alphabetic%29.svg/400px-English_letter_frequency_%28alphabetic%29.svg.png)

Brute Force

Je nach Größe des Alphabets ist die Menge der Schlüssel stark begrenzt. Generell gilt das es in einem Alphabet mit *N* Zeichen nur *N*-1 Verschiebungen geben kann. (Ein Verschiebung um *N* selbst würde bei der Modulo-Division wieder raus fallen.) Im Falle eines gleichbleibenden Schlüssels für die ganze Nachricht ließen sich alle *N*-1 Schlüssel einfach durchprobieren.

Falls es jedoch eine nichtkonstante Variation der Zeichenzuordnungen gibt, wie sie in den Variationen noch vorgestellt wird, ist dieser Ansatz sehr arbeitsaufwendig. In diesem Fall würde man für das erste Zeichen N-1 Möglichkeiten haben, für das zweite Zeichen N-2 usw., somit gibt es (N-1)! Möglichkeiten von Zeichenzuordnungen.

Algorithmus der ursprünglichen Caesar Chiffre

Der Algorithmus hier ist ausgelegt für die 26 Lateinischen Kleinbuchstaben und stellt diese als geordnete Untermenge im ASCII dar. Als Schlüssel sind beliebige ganze Zahlen zugelassen, wobei aufgrund der Modulo-Division nur Zahlen zwischen 1 und 25 sinnvoll sind. Wählt man Zahlen außerhalb dieses Intervals werden sich identische Ergebnisse ergeben.

Eingaben des EnCaesar Algorithmus sind *text* – der zu kodierende Reintext als Zeichenkette und K – die Anzahl der Verschiebungpositionen (Schlüssel). EnCaesar liefert die verschlüsselte Zeichenkette.

```
EnCaesar[text_, K_] :=
 Module[{T = ToCharacterCode[text], out = {},
   i, test},
  For[i = 0, i < Length[T], i++,
   test = T[[i + 1]] + K;
   While[test < 97, test = test + 26];
   While[test > 122, test = test - 26];
   AppendTo[out, test];
  ];
  FromCharacterCode[out]
 ]
```

Zum Dekodieren lässt sich der Kodierungsalgorithmus mit negativem Schlüssel verwenden.Eingabeschema und Ausgabeschema sind analog zum EnCaesar.

```
DeCaesar[text_, K_] := EnCaesar[text, -K]
```

Beispiele per Algorithmus

Das erste Beispiel zeigt das oben manuell angedeutete Beispiel.

```
EnCaesar["kryptographiemachtunsallenfreude",
 7]
```

```
ryfwavnyhwoplthjoabuzhsslumylbkl
```

```
DeCaesar["ryfwavnyhwoplthjoabuzhsslumylbkl",
 7]
```

```
kryptographiemachtunsallenfreude
```

Das zweite Beispiel zeigt eine neutrale Verschiebung um 26 (Alphabetanzahl).

```
EnCaesar["kryptographiemachtunsallenfreude",
 26]
```

```
kryptographiemachtunsallenfreude
```

```
DeCaesar["kryptographiemachtunsallenfreude",
 26]
```

```
kryptographiemachtunsallenfreude
```

Dass dritte Beispiel zeigt eine Verschiebung um 33, diese Verschiebung ist ein Beispiel für Schlüssel die größer sind als die Alphabetanzahl, denn eigentlich entspricht sie der Verschiebung um 7.

```
EnCaesar["kryptographiemachtunsallenfreude",
 33]
```

```
ryfwavnyhwoplthjoabuzhsslumylbkl
```

```
DeCaesar["ryfwavnyhwoplthjoabuzhsslumylbkl",
 33]
```

```
kryptographiemachtunsallenfreude
```

Variationen

Bei der ursprünglichen Caesar Chiffre wird jeder Buchstabe um eine gleiche Distanz verschoben, jedoch ist es möglich eine beliebige Zuordnung von Buchstaben zu verwenden, diese muss noch nicht mal einer einfachen Gesetzmäßigkeit genügen. In einem solchem Fall wäre der Schlüssel die Zuordnungstabelle. Weiterhin lässt sich das "einfache" Brute Force Verfahren von oben nicht mehr anwenden, allerdings bleibt die Buchstabenverteilung der grundlegenden Sprache immer noch vorhanden und macht es deswegen anfällig gegen diese Art Angriff.
Wer auch diesen Angriff erschweren will, kann die Zuordnung nicht auf einzelnen Buchstaben belassen, sondern Textstücke mit einer Länge, welche mehr als ein Zeichen enthält, einer identischen Länge an anderen Buchstaben zuordnen, um den Schlüsselraum weiter zu vergrößern. Ist jedoch die Stücklänge bekannt, kann man analog eine Verteilung von *n*-Tupeln der grundlegenden Sprache vergleichen.
Zu bemerken bleibt jedoch, das in beiden Fällen der Schlüssel wesentlich größer wird und dieser ja selbst auch vom Sender zum Empfänger gelangen muss.

2 Block-Chiffre

Unter einer Block Chiffre versteht man ein Kryptosystem, bei dem der Reintext in Blöcke fester Länge eingeteilt wird, und diese durch eine Funktion auf einen Block gleicher Länge verschlüsselt wird.

Ab diesem Kapitel gelten folgende Notationen:
E sei eine Verschlüsselungsfunktion (Encrypt), D sei eine Dekodierungsfunktion (Decrypt), P sei der Reintext (Plaintext), P' sei der wieder erhaltene Reintext nach erfolgreicher Übertragung und C sei das Chiffrat (Code).
Speziell für diese Kapitel gilt weiterhin, dass der Reintext und das Chiffrat in Zeichenketten bestimmter Länge aufgeteilt werden. Die Indizierung an den Reintextblöcken und Chiffreblöcken, gibt die Nummer des Zeichenkettenblocks an

Definition

Sei A^n ein Reintext- und Schlüsseltextraum, worin alle Wörter der Länge n enthalten sind, welche aus dem Alphabet A zusammengesetzt sind. Für n lässt man nur natürliche Zahlen zu und nennt n die Blocklänge.

Die Verschlüsselungsfunktion (Encrypt) sei $E_K : A^n \to A^n,\ v \mapsto E(v)$ für einen Text v und einen Schlüssel K, analog sei $D_K : A^n \to A^n,\ v \mapsto \mathrm{E}^{-1}(v)$ die dazugehörige Entschlüsselungsfunktion (Decrypt).

Theorem

Die Verschlüsselungsfunktionen einer Blockchiffre sind Permutationen.

Beweis

Zu zeigen ist, dass die Funktion (E) bijektiv ist. Dies trifft hier jedoch schon zu, falls E injektiv ist, da sie von $A^n \to A^n$ geht. Injektiv ist E allerdings sowieso, da zu jedem E ein D wie oben beschrieben existiert.

q.e.d.

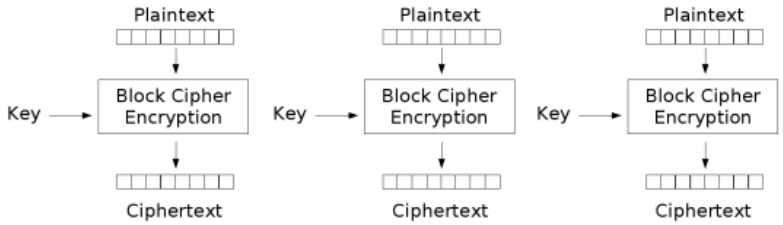
Hiernach ist klar, dass der Schlüsselraum $S(A^n)$ die Menge aller Permutationen von A^n ist. Dieser Schlüsselraum enthält $(|A|^n)!$ Elemente und das ist im allgemeinen eine sehr große Zahl. Das erste Problem, welches sofort ersichtlich wird, ist der Schlüssel *K*. Um K normal zu beschreiben, müsste man für jeden Klartext *v* ein *E*(*v*) aufschreiben, solch eine Tabelle ist jedoch ebenfalls sehr groß mit $|A|^n$ Werten und dadurch sehr unpraktikabel. Somit macht es Sinn nur einen Teil der Permutationen zuzulassen und diese auch einfach erzeugbar zu machen.

Verschlüsselungsmodi

Bei den Verschlüsselungsmodi handelt es sich um Arten der Verschlüsselung mit der Permutation.
Die Beispiele in diesem Abschnitt werden mit dem Binärraum als Alphabet, und beliebigen Permutationen angegeben.
Der Binärraum ist aufgrund der Anwendbarkeit gewählt, denn es lassen sich einfach alle möglichen Nachrichten in binäre Nachrichten umsetzen. Sei dies über die ASCII Tabelle oder die direkte Umrechnung bei Zahlen. Weiterhin ist das Signal bei Datenaustausch auf digitalen Leitungen sowieso eine binäre Zeichenkette.
Eigene programmierte Algorithmen für die Verschlüsselungsmodi selbst werden nicht angeben, jedoch sind alle mit manuell gerechneten Beispielen versehen. Die Algorithmen hierzu werden nicht in Hochsprache angegeben, weil sich die Umsetzung in diesem Setting auf Hardwareebene anbietet, z.b. auf einer Platine mit digitalen Schaltungen.
Im Anhang sind Algorithmen bereitgestellt, um beliebige Umrechnungen vom Binärraum zu machen.

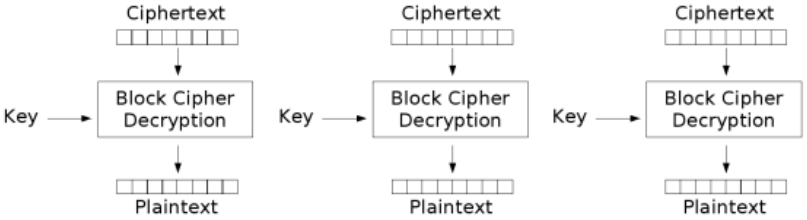
2.1 ECB Modus

Im ECB Modus wird die Verschlüsselung so naheliegend wie möglich vorgenommen. Der Reintext wird direkt permutiert um zu Verschlüsseln.
$C_j = E\left(P_j\right)$ mit E als Kodierungsfunktion (Permutation)



Electronic Codebook (ECB) mode encryption

Zum Dekodieren wird dann analog die inverse Permutation auf die Chiffre angewandt.
$P_j = D(C_j)$ mit D als Dekodierungsfunktion (Inverse Permutation).



Electronic Codebook (ECB) mode decryption

ECB Beispiel

Sei der Reintext 0110 0100 1110 0011 1010, hier in einer Blocklänge n = 4, als Permutation setzen wir A = $\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{matrix}$ eine einfache Spiegelung der Positionen.

P = 0110 0100 1110 0011 1010
P_1 = 0110, P_2 = 0100, P_3 = 1110, P_4 = 0011, P_5 = 1010

Kodieren

C_1 = 0110
C_2 = 0010
C_3 = 0111
C_4 = 1100
C_5 = 0101
C = 0110 0010 0111 1100 0101

Dekodieren

P' = 0110 0100 1110 0011 1010

ECB Angriff

Der ECB Modus ist zwar sehr einfach von der Idee, aber ebenso anfällig. Der Angriff hier ist ebenso einfach, da ein fester Block immer auf den gleichen Block projiziert wird, hat man per statistischer Analyse sehr gute Chancen die Chiffre zu brechen, hierfür würde man dann auf die Buchstabenverteilung der Länge *n* zurückgreifen. Wobei die Statistische Analyse hier natürlich auf den passenden Alphabetraum angepasst werden muss und deswegen nicht zwangsweise einfach zu erhalten ist.

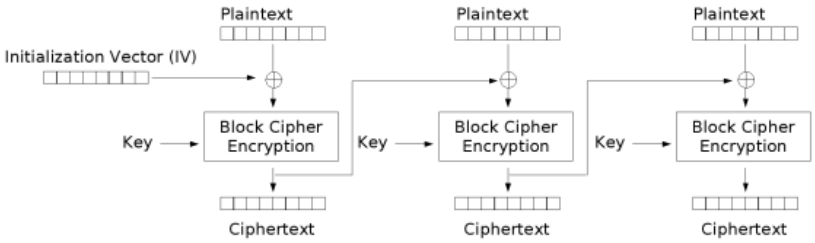
Eine andere Art ist möglich, sobald der Angreifer einen zusammengehörigen Reintext und sein Chiffrat hat, hierbei kann man die Permutationen "suchen" die der Umwandlung genügen, es ist zwar nicht sicher dass nur eine einzige Permutation übrig bleibt, jedoch wird es die Permutationen (Schlüssel) in der Regel einschränken, (Die Ausnahme wäre ein Block der nur Einsen oder nur Nullen enthält, da dieser bei jeder Permutation gleich bleibt).

2.2 CBC Modus

Im CBC wird der Reintext mit dem vorherigen Chiffretext addiert und die Summe permutiert. Da allerdings der erste Block keinen Vorgänger hat, wird ein Initialvektor (IV) benötigt.

Algorithmus der Kodierung:

$$c_0 = \text{IV}$$
$$c_j = E\left(c_{j-1} + p_j\right)$$

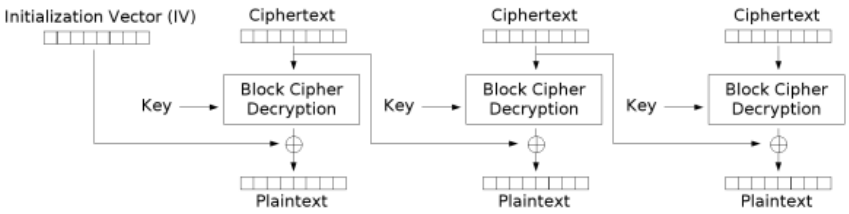


Cipher Block Chaining (CBC) mode encryption

Für die Entschlüsselung wird das Chiffrat mit der inversen Permutation getauscht und wieder mit dem vorherigen Chiffrat addiert (im Binärfall heben sich die beiden Additionen so auf), auch hier wird derselbe Initialvektor benötigt.

Algorithmus der Dekodierung:

$$c_0 = \text{IV}$$
$$m_j = c_{j-1} + D(c_j)$$



Cipher Block Chaining (CBC) mode decryption

CBC Beispiel

Sei der Reintext 0110 0100 1110 0011 1010, hier in einer Blocklänge n = 4, als Permutation setzen wir A = $\begin{matrix}1 & 2 & 3 & 4\\ 4 & 3 & 2 & 1\end{matrix}$ eine einfache Spiegelung der Positionen und der Initialvektor sei 0100.

P = 0110 0100 1110 0011 1010
P_1 = 0110, P_2 = 0100, P_3 = 1110, P_4 = 0011, P_5 = 1010

Kodieren

$C_0 = \text{IV} = 0100$
$C_1 = E(C_0 + P_1) = E(0100 + 0110) = E(0010) = 0100$
$C_2 = E(C_1 + P_2) = E(0100 + 0100) = E(0000) = 0000$
$C_3 = E(C_2 + P_3) = E(0000 + 1110) = E(1110) = 0111$
$C_4 = E(C_3 + P_4) = E(0111 + 0011) = E(0100) = 0010$
$C_5 = E(C_4 + P_5) = E(0010 + 1010) = E(1000) = 0001$

C = 0100 0000 0111 0010 0001

Dekodieren

$C_0 = \text{IV} = 0100$
$P_1 = C_0 + D(C_1) = 0100 + D(0100) = 0100 + 0010 = 0110$
$P_2 = C_1 + D(C_2) = 0100 + D(0000) = 0100 + 0000 = 0100$
$P_3 = C_2 + D(C_3) = 0000 + D(0111) = 0000 + 1110 = 1110$
$P_4 = C_3 + D(C_4) = 0111 + D(0010) = 0111 + 0100 = 0011$
$P_5 = C_4 + D(C_5) = 0010 + D(0001) = 0010 + 1000 = 1010$

P' = 0110 0100 1110 0011 1010

CBC Angriff

Um hierbei Erfolg zu haben, die Permutations Matrix zu bekommen, braucht der Angreifer ein Reintext/Chiffre Paar des gleichen Blocks sowie die Chiffre des vorherigen Blocks

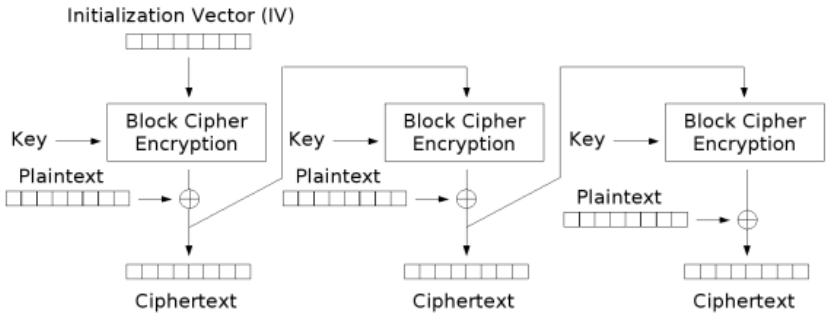
Weiter ist zu bemerken, dass die Änderung der Nachricht eines Blocks, Auswirkungen auf den aktuellen Block und den folgenden Block haben wird. Dies bedeutet insbesondere, dass selbst wenn nicht der gleiche Initialvektor vorliegt, nur der erste Textblock abweicht, danach wird er nicht mehr benötigt.

2.3 CFB Modus

Bisher erforderten die Verfahren immer serielle Berechnung und somit muss die Berechnung des Empfängers immer auf den Sender warten. Dies kann zu Wartezeiten führen. Im CFB Modus können sowohl Sender als auch Empfänger parallel Berechnungen anstellen, um diese Zeitdifferenz zu minimieren.
Es wird ein Initialvektor der Blocklänge n benötigt, dieser wird verschlüsselt per Permutation. Von diesem werden nur die ersten r Bits genommen (r < n) und diese mit dem Reintext addiert, welche ebenfalls in Stücke der Länge r aufgeteilt werden, somit erhält man das erste Stück der Chiffre. Dieses Stück Chiffre wird dann an den IV angehängt und die ersten r Bits die bereits verwendet wurden, werden abgeschnitten, somit bleibt ein Teil des IV und ein Teil der ersten Chiffre, insgesamt hat dieses Konstrukt wieder Blocklänge n.
Im nächsten Schritt, würde dieser wieder permutiert, wie der IV vom Anfang. Dies wird iteriert bis der ganze Reintext in Chiffre umgesetzt ist.

Algorithmus der Kodierung:

$O_j = E_k(I_j)$
t_j der Teil aus den ersten r Bits von O_j
$c_j = p_j + t_j$
$I_{j+1} = 2^r I_j + c_j \bmod 2^n$
(Dies ist das Abschneiden der ersten r Bits und Anhängen der Chiffre)



Cipher Feedback (CFB) mode encryption

Zum Entschlüsseln wird das Verfahren analog umgesetzt, wobei gilt, dass der Reintext bei der Addition der Stücke(IVc_1 c_1...) und der Chiffre entsteht.

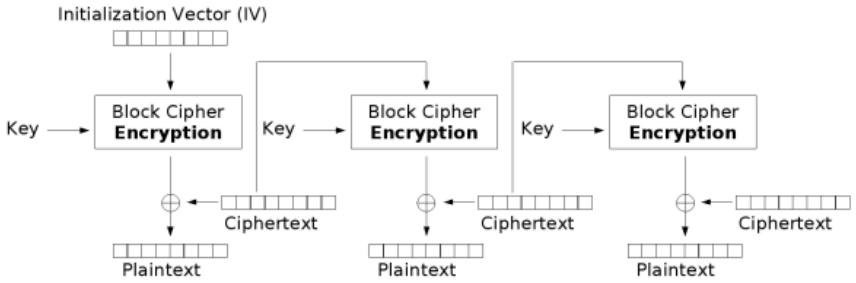
Algorithmus der Dekodierung:

$O_j = E_k(I_j)$

t_j der Teil aus den ersten r Bits von O_j

$p_j = c_j + t_j$

$I_{j+1} = 2^r I_j + c_j \bmod 2^n$



Cipher Feedback (CFB) mode decryption

Die Permutationen und alles Andere, bis auf die Addition, lässt sich parallel umsetzen, was die oben genannte Zeitersparnis bewirkt, weiterhin wird nur die Permutation auf beiden Seiten gebraucht und nicht die Inverse Permutation.

CFB Beispiel

Sei der Reintext 011 001 001 110 001 110, hier in einer Blocklänge n = 4 mit r = 3, als Permutation setzen wir A = $\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{matrix}$ eine einfache Spiegelung der Positionen und der Initialvektor sei 0100.

P = 011 001 001 110 001 110

$P_1 = 011$, $P_2 = 001$, $P_3 = 001$, $P_4 = 110$, $P_5 = 001$, $P_6 = 110$

Kodierung

$I_1 = \text{IV} = 0100$

$O_1 = E_A(I_1) = E_A(0100) = 0010$	$t_1 = 001$
$c_1 = P_1 + t_1 = 011 + 001 = 010$	$I_2 = 0100010$
$O_2 = E_A(I_2) = E_A(0010) = 0100$	$t_2 = 010$
$c_2 = P_2 + t_2 = 001 + 010 = 011$	$I_3 = 0010011$
$O_3 = E_A(I_3) = E_A(0011) = 1100$	$t_3 = 110$
$c_3 = P_3 + t_3 = 001 + 110 = 111$	$I_4 = 0011111$
$O_4 = E_A(I_4) = E_A(1111) = 1111$	$t_4 = 111$
$c_4 = P_4 + t_4 = 110 + 111 = 001$	$I_5 = 1111001$
$O_5 = E_A(I_5) = E_A(1001) = 1001$	$t_5 = 100$
$c_5 = P_5 + t_5 = 001 + 100 = 101$	$I_6 = 1001101$
$O_6 = E_A(I_6) = E_A(1101) = 1011$	$t_6 = 101$
$c_6 = P_6 + t_6 = 110 + 101 = 011$	

C = 010 011 111 001 101 011

Dekodierung

$I_1 = \text{IV} = 0100$

$O_1 = E_A(I_1) = E_A(0100) = 0010$	$t_1 = 001$
$P_1 = c_1 + t_1 = 010 + 001 = 011$	$I_2 = 0100010$
$O_2 = E_A(I_2) = E_A(0010) = 0100$	$t_2 = 010$
$P_2 = c_2 + t_2 = 011 + 010 = 001$	$I_3 = 0010011$
$O_3 = E_A(I_3) = E_A(0011) = 1100$	$t_3 = 110$
$P_3 = c_3 + t_3 = 111 + 110 = 001$	$I_4 = 0011111$
$O_4 = E_A(I_4) = E_A(1111) = 1111$	$t_4 = 111$
$P_4 = c_4 + t_4 = 001 + 111 = 110$	$I_5 = 1111001$
$O_5 = E_A(I_5) = E_A(1001) = 1001$	$t_5 = 100$
$P_5 = c_5 + t_5 = 101 + 100 = 001$	$I_6 = 1001011$
$O_6 = E_A(I_6) = E_A(1011) = 1101$	$t_6 = 101$
$P_6 = c_6 + t_6 = 011 + 101 = 110$	

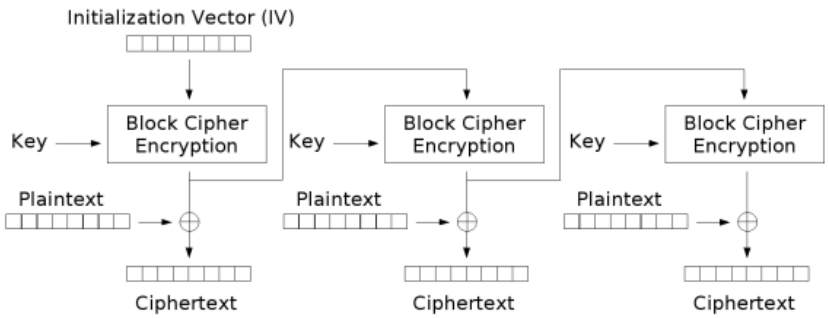
P' = 011 001 001 110 001 110

2.4 OFB Modus

Im OFB Modus wird die Idee der Parallelisierung der zeitaufwendigen Berechnungen weiter gefördert. Sowohl beim Kodieren als auch Dekodieren kann alles, bis auf eine Addition, komplett unabhängig des Reintextes und der Chiffre berechnet werden.
Ein Initialvektor wird verschlüsselt und anteilhaft auf Anteile des Reintextes addiert um das Chiffrat zu erhalten, für den nächsten Teil wird allerdings der komplette verschlüsselte Initialvektor wieder als neuer Input zum Verschlüsseln genommen.
Zu bemerken ist hier, das auch immer nur Verschlüsselt wird, somit ist die inverse Funktion zu E ist nicht erforderlich.

Algorithmus derKodierung:

$$O_j = E_k(I_j)$$
$$t_j \text{ erste } r \text{ bits von } O_j$$
$$c_j = p_j + r_j$$
$$I_{j+1} = O_j$$

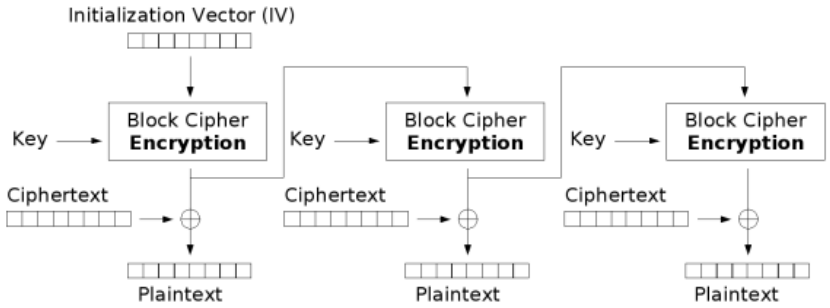


Output Feedback (OFB) mode encryption

Zur Entschlüsselung braucht man die gleiche Folge O_j, wobei die eigentliche Entschlüsselung durch die Selbstinversität der Addition im Dualraum durchgeführt wird

Algorithmus der Dekodierung:

$$O_j = E_k(I_j)$$
t_j erste r bits von O_j
$$p_j = c_j + r_j$$
$$I_{j+1} = O_j$$



Output Feedback (OFB) mode decryption

OFB Beispiel

Sei der Reintext 011 001 001 110 001 110, hier in einer Blocklänge n = 4 mit r = 3, als Permutation setzen wir A = $\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{matrix}$ eine einfache Spiegelung der Positionen und der Initialvektor sei 0100.

P = 011 001 001 110 001 110
$P_1 = 011, P_2 = 001, P_3 = 001, P_4 = 110, P_5 = 001, P_6 = 110$

Kodierung

$I_1 = \text{IV}$
$O_1 = E(I_1) = E(0100) = 0010 \quad t_1 = 001$
$c_1 = p_1 + t_1 = 011 + 001 = 010 \qquad I_2 = O_1 = 0010$
$O_2 = E(I_2) = E(0010) = 0100 \quad t_2 = 010$
$c_2 = p_2 + t_2 = 001 + 010 = 011 \qquad I_3 = O_2 = 0100$
$O_3 = E(I_3) = E(0100) = 0010 \quad t_3 = 001$
$c_3 = p_3 + t_3 = 001 + 001 = 000 \qquad I_4 = O_3 = 0010$

$O_4 = E(I_4) = E(0010) = 0100 \quad t_4 = 010$

$c_4 = p_4 + t_4 = 110 + 010 = 100 \qquad I_5 = O_4 = 0100$

$O_5 = E(I_5) = E(0100) = 0010 \quad t_5 = 001$

$c_5 = p_5 + t_5 = 001 + 001 = 000 \qquad I_6 = O_5 = 0010$

$O_6 = E(I_6) = E(0010) = 0100 \quad t_6 = 010$

$c_6 = p_6 + t_6 = 110 + 010 = 100$

($I_7 = O_6 = 0100$ wird nichtmehr benötigt)

C = 010 011 000 100 000 100

Dekodierung

$I_1 = \text{IV}$

$O_1 = E(I_1) = E(0100) = 0010 \quad t_1 = 001$

$p_1 = c_1 + t_1 = 010 + 001 = 011 \qquad I_2 = O_1 = 0010$

$O_2 = E(I_2) = E(0010) = 0100 \quad t_2 = 010$

$p_2 = c_2 + t_2 = 011 + 010 = 001 \qquad I_3 = O_2 = 0100$

$O_3 = E(I_3) = E(0100) = 0010 \quad t_3 = 001$

$p_3 = c_3 + t_3 = 000 + 001 = 001 \qquad I_4 = O_3 = 0010$

$O_4 = E(I_4) = E(0010) = 0100 \quad t_4 = 010$

$p_4 = c_4 + t_4 = 100 + 010 = 110 \qquad I_5 = O_4 = 0100$

$O_5 = E(I_5) = E(0100) = 0010 \quad t_5 = 001$

$p_5 = c_5 + t_5 = 000 + 001 = 001 \qquad I_6 = O_5 = 0010$

$O_6 = E(I_6) = E(0010) = 0100 \quad t_6 = 010$

$p_6 = c_6 + t_6 = 100 + 010 = 110$

($I_7 = O_6 = 0100$ wird nichtmehr benötigt)

P' = 011 001 001 110 001 110

OFB Angriff

Wie schon im Beispiel leicht zu erkennen ist, kann je nach Blocklänge und Funktion E bei gleichem Schlüssel die Folge der I schnell periodisch werden. Somit wird empfohlen die Permutations Matrizen in E zu verändern, was unterschiedliche Schlüssel zur Folge hat.
Weil allerdings so viele Schlüssel geheim auszutauschen die gleiche Schwierigkeit macht, wie einen Reintext der gleichen Länge geheim zu übertragen, sollte man überlegen, ob man entweder Schlüssel mehrfach verwendet oder geschickt erzeugen kann. Auf den 2. Punkt wird bei der Stromchiffre genauer eingegangen.

2.5 Stromchiffre

Die Stromchiffre ist eine simple Rekursion die zu Verschlüsselung genutzt wird. Es wird mit einem Initialvektor gestartet, welcher per Rekursion beliebig lang gezogen werden kann. Die eigentliche Chiffre entsteht durch Addition der Rekursionsfolge und des Reintextes. Weil allerdings mit gleichem IV und gleichem Rekursionsgesetz diese Folge, unabhängig der Chiffre und des Reintextes, berechnet werden kann, bietet es sich auch hier an, die beiden Folgen parallel zu berechnen. Dies kann ebenfalls zu einer anderen Blockchiffre benutzt werden, indem man nur ausgewählte Segmente der Folge zur Addition nutzt (es müssen natürlich in diesem Fall beide Seiten wissen, welche Segmente).

Rekursion

Die Rekursion im Allgemeinen ist eine Vorschrift die durch Verknüpfung der schon vorhandenen Elemente, neue Elemente erschafft. Solch eine Rekursionsfolge kann auch schon bestimmte Zwecke in anderen Systemen übernehmen, wie z.b. im OFB Modus, als Schlüsselfolge.

Beispiel:
Sei ein Initialvektor 0100 und die Vorschrift $I_{j+4} = I_{j+1} + I_{j+3}$ damit ergibt sich die Folge: 0100 1110 1001 1101...

Wie man schon im Beispiel sieht wird die Folge schnell periodisch. Durch längere Initialvektoren und durch kompliziertere Vorschriften, lässt sich die Periode zwar verlängern, jedoch ist es nicht möglich, es nicht periodisch zu machen.

Beispiel zur Stromchiffre

Sei die Rekursionsfolge R wie im Beispiel oben und der Reintext sei:
P = 0110 0100 1110 0011

Kodieren:
P = 0110 0100 1110 0011
+
R = 0100 1110 1001 1101
=
C = 0010 1010 0111 1110

Dekodieren:
C = 0010 1010 0111 1110
+
R = 0100 1110 1001 1101
=
P = 0110 0100 1110 0011

Blockchiffren außerhalb des Binärraums

Die Verfahren bis hierhin wurden ausschließlich für den Binärraum gezeigt wobei der Schwerpunkt dieser Konzeption in der Implementierung auf Hardwareebene liegt.
Selbstverständlich lassen sich Blockchiffren auch in anderen Alphabeträumen definieren, ein Beispiel hierfür ist das Kryptosystem der Linearen Chiffren, welches weiter unten noch aufgegriffen wird.

Mehrfachverschüsselung

Für eine Sicherheitssteigerung der Blockchiffre ist es möglich das Verfahren mehrfach mit verschiedenen Schlüsseln durchzuführen. Gebräuchlich ist hierbei eine E-D-E Dreifach Verschlüsselung. E steht hierbei für Encrypt und D für Decrypt, also Ver- und Entschlüsseln. Somit erhält man eine Funktion der Art:

$$c = E_{k_1}\left(D_{k_2}\left(E_{k_3}(p)\right)\right)$$

für Reintext p (Plaintext) und Chriffretext c. Es ist möglich Schlüssel auch mehrfach zu verwenden, hier z.b. $k_1 = k_3$, dies würde die Übertragung eines Schlüssels sparen.

Weiter oben wurde schon der Vorteil der Berechenbarkeit auf Hardwareebene genannt. Selbstverständlich lassen sich somit auch Mehrfachverschlüsselungen komplett auf Hardwareebene ohne großen Rechenaufwand umsetzen.

3 Ringgrundlagen

Für weitere Algorithmen werden Restklassenringe gebraucht. Hierzu ein paar Sätze und Algorithmen die in den Kryptosystemen verwendet werden. Diese Grundlagen sind für beliebige Ringe gültig, jedoch sind für den kryptographischen Rahmen fast ausschließlich Restklassenringe interessant.

Größter gemeinsamer Teiler

Der größte gemeinsame Teiler (greatest common divisor) von (a, b), mit a, $b \in \mathbb{N}$ ist die größte Zahl $c \in \mathbb{N}$ die für gilt c teilt a und c teilt b.
Eins erfüllt diese Bedingung zwar immer, jedoch muss es nicht unbedingt die größte Zahl sein

Beispiel:
gcd(81, 27) =27
gcd(81, 18) = 9
gcd(81, 17) = 1

3.1 Satz der Division mit Rest

Der Satz der Division mit Rest stellt sicher, das wir eine Division in den grundlegenden Ringen nutzen können, da eine allgemeine Division in der Ringstruktur nicht vorgesehen ist und nicht für alle Elemente möglich ist. Diese Ring-Divisionsvorschrift teilt im eigentlichen Sinn solange es möglich ist und gibt einen nicht weiter aufteilbaren Rest zusätzlich zum Quotienten aus.

Satz

Seien $x, y \in \mathbb{N}$, dann gibt es genau ein Paar (q, r) mit $q, r \in \mathbb{N}$, mit $0 \leq r < y$, so dass gilt $x = q * y + r$.

Beweisskizze

Für die Existenz wird über x induziert.

Zur Eindeutigkeit wird angenommen, es existieren zwei Paare (q, r), sodass $x = q_1 * y + r_1 = q_2 * y + r_2$mit oben beschriebenen Vorraussetzungen. Es gilt $r_1, r_2 < y \Rightarrow (q_2 - q_1)\, y = r_1 - r_2 \geq 0$. ObdA sei $r_1 \geq r_2 \Rightarrow q_2 \geq q_1$, weil gilt $r_1 < y \Rightarrow (q_2 - q_1)\, y = r_1 - r_2 \geq 0 \Leftrightarrow 0 \leq q_2 - q_1 < 1$ (Division um y). $0 \leq q_2 - q_1 < 1$ heißt aber für natürliche Zahlen q_1, q_2, dass diese gleich sein müssen. Somit gibt es nur ein q und ebenso auch nur ein r.

q.e.d.

3.2 Der Euklidische Algorithmus

Der Euklidische Algorithmus wendet iterativ Division mit Rest an, um zwei Zahlen immer weiter zu reduzieren und findet in verschiedenen Anwendungen Einsatz, z.b. die Berechnung des GCD.

Eingaben des Euklidischen Algorithmus sind zwei Ringelemente, im Setting des Restklassenringes über den ganzen Zahlen sind die Eingaben zwei ganze Zahlen a, b. Die Ausgabe ist eine ganze Zahl, die der größte gemeinsame Teiler der Eingaben ist.

Pseudo Code

Der Algorithmus kann sowohl rekursiv als auch iterativ umgesetzt werden.

Iterative Variante:
EUCLID(a,b)
1 solange b ≠ 0
2 h ← a mod b
3 a ← b
4 b ← h
5 return a

Rekursiv Variante:
EUCLID(a,b)
1 wenn b = 0
2 dann return a
3 sonst return EUCLID(b, a mod b)

mod steht hier für oben beschriebene Restdivision und gibt den Rest von a bei division um b zurück

Algorithmus

Hier die Umsetzung der rekursiven Variante:

```
EuclidAlgo[a_, b_] :=
 If[b == 0, a, EuclidAlgo[b, Mod[a, b]]]
```

Brechnung des GCD wie im Beispiel oben

```
EuclidAlgo[81, 27]
```

```
27
```

```
EuclidAlgo[81, 18]
```

```
9
```

```
EuclidAlgo[81, 17]
```

```
1
```

3.3 Lemma von Bézout

Das Lemma von Bézout gibt die Existenz einer spezielle Zerlegung des gröten gemeinsamen Teilers an. Diese Darstellung ist eine Summe aus Produkten der beiden Eingaben. Dies kann genutzt werden, um multiplikative Inverse zu Zahlen in Restklassenringen zu finden.

Lemma

Seien $a, b \in \mathbb{N}$, dann lässt sich der größte gemeinsame Teiler in folgender Form darstellen gcd(a, b) = s * a + t * b mit $s, t \in \mathbb{Z}$, insbesondere gilt 1 = s * a + t * b für a, b teilerfremd oder mindestens eine prim.

Beweis

Setze $d = s * a + t * b$ mit $d \neq 0$ und $s, t \in \mathbb{Z}$ mit d minimal positiv gewählt. Über die Transitivität der Teilbarkeit, teilt gcd(a, b) auch d. Für $d = 1$ ist jetzt schon alles bewiesen. Sei also $d > 1$, über den Satz der Division mit Rest bekommt man folgende Darstellung $a = q * d + r$ mit $0 \leq r < d$. Setzt man dieses in die Ursprungsgleichung ein und löst nach r auf, erhält man: $r = (1 - q * s) * a + (-q * t) * b$. Da allerdings d minimal gewählt worden ist, muss $r = 0$ sein und somit ist d ein Teiler von a. Analog ist d auch ein Teiler von b. Weil d nun beide teilt, gilt $d \leq$ gcd(a, b), da aber gcd(a, b) teilt d gilt, muss d = gcd(a, b) sein.

q.e.d.

Berechnung

Der Euklidische Algorithmus wird modifiziert um die Koeffizienten s, t zusätzlich zu berechnen.

3.4 Der Erweiterte Euklidische Algorithmus

Dies ist die Erweiterung des schon oben beschriebenen Euklidischen Algorithmus. Die Eingaben sind wieder zwei Ringelemente, jedoch ist die Ausgabe nicht nur der größte gemeinsame Teiler, sondern weiterhin die Zerlegung, welche im Lemma von Bézout gezeigt wurde.

Pseudo Code

Diesmal nur die rekursive Variante:

a,b: zwei Zahlen für die der erweiterte euklidische Algorithmus durchgeführt wird

EEA(a,b)
1 wenn b = 0
2 dann return (a,1,0)
3 (d',s',t') ← EEA(b, a mod b)
4 (d,s,t) ← (d',t',s' - floor(a/b)t')
5 return (d,s,t)

Algorithmus

```
EEA[a_, b_] := Module[{D, S, T, d, s, t},
  If[b == 0, d = a; s = 1; t = 0];
  If[b ≠ 0, {D, S, T} = EEA[b, Mod[a, b]];
   {d, s, t} = {D, T, S - Floor[a / b] * T}];
  {d, s, t}
 ]
```

Die Ausgabe ist eine Liste mit d als dem größten gemeinsamen Teiler, s und t sind die Koeffizienten von a, b wie im Lemma.

Beispiel

```
EEA[81, 18]
```

```
{9, 1, -4}
```

Also ist 9 = 1 * 81 + (-4) * 18

3.5 Chinesischer Restsatz

Der Chinesische Restsatz nimmt sich dem Problem eines Gleichungssystems in Restklassengleichungen an, bzw. wie sich aus zwei Repräsentanten in unterschiedlich großen Restklassenringen ein Repräsentant aus einem größerem Restklassenring finden lässt, welcher beide vorherigen Bedingung erfüllt. Für den Fall, dass mehr als zwei Gleichungen gelöst werden sollen ist eine sukzessive Anwendung mit immer zwei Gleichungen möglich.

Satz

Sei $p, q \in \mathbb{Z}$ und sei gcd(p, q) = 1, dann hat das Gleichungssystem $x \equiv a \mod p$; $x \equiv b \mod q$ eine ganzzahlige Lösung $x \in \mathbb{Z}$ welche bis auf additive Vielfache von $p*q$ eindeutig ist.

Beweis

Eindeutigkeit:
Seien x, x' zwei solcher Lösungen. So gilt $x \equiv x' \mod p$ und $x \equiv x' \mod q$, durch gcd(p, q) = 1 folgt jedoch $x \equiv x' \mod p*q$ und somit Eindeutigkeit bis auf additive Vielfache von $p*q$.
Für die Existenz wird auf den folgenden Algorithmus verwiesen.

q.e.d.

Algorithmus

Da der Algorithmus den oben verwendeten EEA Algorithmus beinhaltet, muss dieser erst initialisiert werden.
Der Algorithmus lässt sich für Systeme mit mehr als 2 Gleichungen iterativ anwenden.
Die Eingaben sind zwei Restklassengleichungen (x = a mod p und x = b mod q) und die Ausgabe ist die Restkassengleichung in der beide Eingangsgleichungen vereinigt sind. Erstes Listenelement ist der Repräsentant und zweites Listenelement ist der neue Modul.

```
CRS[a_, p_, b_, q_] := Module[{d, y, z},
  {d, y, z} = EEA[p, q];
  {Mod[a - y p (a - b), p * q], p * q}
 ]
```

Beispiel

```
CRS[1, 5, 2, 7]
```

```
{16, 35}
```

x = 1 mod 5, x = 2 mod 7 => x = 16 + (35*ℤ)

3.6 Satz von Fermat

Sei $a \in \mathbb{Z}$ und p eine Primzahl dann gilt $a^p \equiv a \bmod p$.
Für a ≯ p gilt $a^{p-1} \equiv 1 \bmod p$.

Beweis

Induktion über a für alle nichtnegativen ganzen Zahlen. (Dies ist für Anwendungen der Kryptographie ausreichend)

Es wird gezeigt das $a^p - a$ durch p teilbar ist

Induktionsanfang: $0^p - 0 = 0$ ist durch p teilbar.

Induktionsschritt: Die Behauptung gelte für beliebiges festes a,
$(a+1)^p - (a+1) = a^p + \binom{p}{1} a^{p-1} + ... + \binom{p}{p-1} a + 1 - (a+1)$ nach dem binomischen Satz.

Betrachtet man die Binomialkoeffizienten $\binom{p}{k} = \frac{p*(p-1)***(p-k+1)}{1*2***k}$ so sind alle durch p teilbar, da p für $1 \leq k \leq p$-1 nur im Zähler auftaucht.
Somit reduziert es sich auf $(a+1)^p \equiv a^p + 1 - (a+1) = a^p - a \mod p$, dies ist nach Induktionsvorraussetzung durch p teilbar

q.e.d.

Verwendung

Der Satz lässt sich als Primzahltest verwenden. Sei p eine Primzahl so gilt $a \nmid p$ und $a^{p-1} \equiv 1 \mod p$. Wichtig hierbei ist zu wissen, dass dies kein Test zur Bestätigung einer Primzahl ist, selbst wenn die Kongruenz für alle a gilt, jedoch kann man sicher sagen dass p nicht prim ist, wenn dies für ein beliebiges a nicht gilt.

```
Mod[2^(13-1), 13]
```

```
1
```

Dies zeigt, dass 13 eine Primzahl sein kann.

```
Mod[2^(14-1), 14]
```

```
2
```

Dies zeigt das 14 keine Primzahl ist

4 Lineare Chiffren

Mit dem weiteren Wissen über Ringe und spezielle Restklassenringe lassen sich nun die Linearen Chiffren erklären. Diese Systeme sind ebenfalls Blockchiffren, jedoch werden sie nicht auf Hardwareebene implementiert, sondern erfordern weiterführende Berechnungen.
In diesem Kapitel gilt folgende Konvention:
Der Zeichenraum ist ein Restklassenring mit n Zeichen .Diese Zeichen werden als Zahlen von 0 bis n-1 dargestellt. Eine Zeichenfolge der Länge m ist somit ein Vektor aus dem Restklassenring mit m Elementen.

Kodierung

Der Reintext Vektor wird mit einer quadratischen Matrix A multipliziert und wahlweise mit einem weiteren Vektor b der Länge m addiert. Nach den Berechnungsschritten bleibt ein Vektor der Länge m übrig, dieser ist der Codevektor bzw. das Chiffrat. Zu bemerken ist hierbei, dass alle Operationen im Restklassenring durchgeführt werden.

Schlüssel

Der Schlüssel bei diesem System ist die Matrix A, sowie falls benutzt, ein Zusatzvektor b. Zum erfolgreichen Dekodieren ist es erforderlich dass die Matrix A nicht nur quadratisch ist, sondern auch invertierbar. Somit ist $A \in GL(\mathbb{Z} / n\mathbb{Z})$ und $b \in (\mathbb{Z} / n\mathbb{Z})$.

Dekodierung

Zum Dekodieren wird mit der Inversen Matrix von A multipliziert und, falls der Zusatzvektor b benutzt wurde, dieser vorher wieder subtrahiert.
Hiernach erhält man den Lösungsvektor P' welcher der wieder erhaltene Reintext ist.

Beispiel

Der zu verschlüsselnde Reintextvektor sei „test“ also P = (19, 4, 18, 19) mit der trivialen Buchstabenzuordnung a = 0, b = 1 usw., der Alphabetraum hat somit n = 26 Zeichen und die Vektorlänge (Blocklänge) sei m = 2.
Wir teilen P in zwei Blöcke der Länge 2 auf: $p_1 = (19, 4)$, $p_2 = (18, 19)$.
Die Schlüssel Matrix sei: $A = \begin{pmatrix} 11 & 8 \\ 3 & 7 \end{pmatrix}$ mit ihrer Inversen

$A^{-1} = \begin{pmatrix} 7 & 18 \\ 23 & 11 \end{pmatrix}$

Verschlüsselung:

$$c_1 = \begin{pmatrix} 11 & 8 \\ 3 & 7 \end{pmatrix} \begin{pmatrix} 19 \\ 4 \end{pmatrix} = \begin{pmatrix} 241 \\ 85 \end{pmatrix} = \begin{pmatrix} 7 \\ 7 \end{pmatrix}$$
$$c_2 = \begin{pmatrix} 11 & 8 \\ 3 & 7 \end{pmatrix} \begin{pmatrix} 18 \\ 19 \end{pmatrix} = \begin{pmatrix} 350 \\ 187 \end{pmatrix} = \begin{pmatrix} 12 \\ 5 \end{pmatrix}$$

C = (7, 7, 12, 5)

Dekodieren:

$$p_1' = \begin{pmatrix} 7 & 18 \\ 23 & 11 \end{pmatrix} \begin{pmatrix} 7 \\ 7 \end{pmatrix} = \begin{pmatrix} 175 \\ 238 \end{pmatrix} = \begin{pmatrix} 19 \\ 4 \end{pmatrix}$$
$$p_2' = \begin{pmatrix} 7 & 18 \\ 23 & 11 \end{pmatrix} \begin{pmatrix} 12 \\ 5 \end{pmatrix} = \begin{pmatrix} 174 \\ 331 \end{pmatrix} = \begin{pmatrix} 18 \\ 19 \end{pmatrix}$$

P' = (19, 4, 18, 19)

Angriff auf die Lineare Chiffre

Durch die Linearität des Systems ist dieses System sehr anfällig gegen einen Angriff, bei dem schon ein paar Kombinationen von Reintext und zugehörigem Chiffretext bekannt sind. In diesem Fall lässt sich die Matrix A als Gleichungssystem lösen und man erhält den Schlüssel.

5 Public Key Verfahren

Bisher waren alle Kryptosysteme symmetrisch, d.h. sowohl Verschlüsseln als auch Entschlüsseln werden mit dem gleichen Schlüssel durchgeführt. Die Sicherheit des Systems basiert also darauf, dass beide Seiten den Schlüssel haben und geheim halten, und ggf. dass die Art der Verschlüsselung auch geheim gehalten wird. Das große Problem ist hierbei jedoch der Schlüsselaustausch, immerhin müssen beide Seiten erstmal den Schlüssel haben. Bei einem Public Key Verfahren wird sowohl die Art der Verschlüsselung als auch ein öffentlicher Schlüssel frei zugänglich gemacht, es wird nur ein geheimer Schlüssel, der ausschließlich zum Entschlüsseln verwendet wird, behalten.

5.1 Die RSA Chiffre

RSA ist ein Public Key Verfahren, welches die oben aufgeführten Bedingungen einhält

Schlüsselerzeugung

Wähle zwei Primzahlen $p \neq q$ und bestimme $n = p * q$.
Wähle nun e sodass gilt, $1 < e < (p\text{-}1)*(q\text{-}1)$ mit $\gcd(e, (p\text{-}1)*(q\text{-}1)) = 1$.
Bestimme per Erweitertem Euklidischen Algorithmus d sodass $d * e \equiv 1 \mod (p\text{-}1)*(q\text{-}1)$

Öffentlicher Schlüssel ist das Paar (n, e), Privater Schlüssel ist d

Kodierung

Für den Reintext m mit $0 \leq m < n$ erzeugt man den Chiffretext c mit:
$c = m^e \mod n$
Man bedenke dass e in der Regel eine große Zahl ist und hier schon ein großer Rechenaufwand betrieben wird, deswegen bietet es sich an, effizientere Algorithmen als sukzessive Multiplikation zu verwenden, siehe Zusatz.

Dekodierung

Für Chiffretext c wird der Reintext m nach folgendem Prinzip berechnet:
$m = c^d \bmod n$

Man sieht dieses Folgendermaßen:
Da $d * e \equiv 1 \bmod (p\text{-}1)*(q\text{-}1)$ gibt es ein l sodass $e*d = 1 + l\ (p\text{-}1)*(q\text{-}1)$.
Damit gilt $(m^e)^d = m^{e\,d} = m^{1 + l(p-1)*(q-1)} = m\left(m^{(p-1)\,(q-1)}\right)^l$
Man erkennt: $(m^e)^d \equiv m\left(m^{(p-1)}\right)^{(q-1)\,l} \equiv m \bmod p$, ist p Teiler von m ist dies Trivial, andernfalls gilt dies nach dem Satz von Fermat (siehe oben).
Analog erhält man: $(m^e)^d \equiv m \bmod q$.
Weil aber p, q verschiedene Primzahlen sind gilt: $(m^e)^d \equiv m \bmod n$.

RSA Kodierungs Algorithmus

Hier werden die Algorithmen der RSA Kodierung einmal umgesetzt, um sie später weiter verwenden zu können. Die Funktionsweise entspricht dem oben genanntem Berechnungsprinzip.
Eingaben der Kodierung sind m – der Reintext, e – die gewählte öffentliche Schlüsselkomponente und n - der RSA Modul. Mit diesen Eingaben liefert der RSA Algorithmus das zugehörige Codewort c. Eingaben der Dekodierung DeRSA sind c – das empfangene Codewort, d – der private Schlüssel und n – der RSA Modul. DeRSA liefert die Dekodierte Nachricht aus dem Codewort.

```
RSA[m_, e_, n_] := Mod[m^e, n]
```

```
DeRSA[c_, d_, n_] := Mod[c^d, n]
```

Angriff auf RSA

Die Sicherheit von RSA basiert auf der Schwierigkeit n in die Primzahlen p, q zu faktorisieren, könnte man dies, würde man mit dem so erhaltenen p, q einfach das d zu dem schon vorhandenen e berechnen und hätte auch den geheimen Schlüssel aus dem öffentlichen erhalten. Allerdings sind p, q in der Regel so gewählt das,das Faktorisieren von n ein Rechenzeit Problem von enorme Ausmaß annimmt.

Erstellung eines Beispiel RSA-Systems

Das System basiert auf zwei gewählten Primzahlen p und q. Für dieses Beispiel System werden $p = 53$ und $q = 59$ gewählt. Als nächstes berechnet man $n = p*q = 53*59 = 3127$ und $(p-1)*(q-1) = 52*58 = 3016$. Man wähle ein e mit folgenden Bedingungen: $1 < e < (p-1)*(q-1)$ und gcd(e, $(p-1)*(q-1)$).

```
EuclidAlgo[3016, 17]
```

```
1
```

$e = 17$ erfüllt beide Bedingungen. Fehlt noch ein d mit $d * e \equiv 1 \mod (p-1)*(q-1)$ welches sich über den Erweiterten Euklidischen Algorithmus berechnen läst.

```
EEA[17, 3016]
```

```
{1, -887, 5}
```

Somit gilt nun $1 = 17 * (-887) + 5 * 3016$ oder in unserem Setting $(-887) * 17 \equiv 1 \mod 3016$. Für d wählt man einen geeigneten Repräsentanten z.b. 3016-887 = 2129

Die Schlüssel dieses Beispielsystems sind öffentlich ($e = 17$, $n = 3127$) und Privat ($d = 2129$)
Die zu verschlüsselnde Nachricht sei $m = 1756$

Kodierung nach der Vorschrift $c = m^e \mod n$

```
m = 1756; e = 17; n = 3127; d = 2129;
```

```
c = RSA[m, e, n]
```

```
1786
```

Diese Chiffre wird nun mit dem privaten Schlüssel wieder dekodiert.

```
M = DeRSA[c, d, n]
```

```
1756
```

Der wieder erhaltene Reintext M ist gleich dem ursprünglichem Reintext m.

Nun wird eine Angriff auf dieses System simuliert. Hierzu Faktorisiert man gegebenes n in die Zahlen p, q

```
FactorInteger[n]
```

```
{{53, 1}, {59, 1}}
```

Dies zeigt n = $53^1 * 59^1$, wie zu erwarten war, muss n aus zwei potenzfreien Primzahlen bestehn. Setze p = 53 und q = 59.
Mit der Kenntnis über diese Primzahlen lässt sich genau analog zu oben das *d* berechnen. Mit diesem d lässt sich auch als nicht Erfinder das System entschlüsseln.

6 Zusatzalgorithmen

6.1 Algorithmus zum schnellen Exponenzieren

Da in der Kryptographie viel Exponenziert wird, ist es sinnvoll auch dort Zeit einzusparen. Normalerweise verwendet eine Exponenzierung mit Exponenten *n* genau *n*-1 Multiplikationen. Folgender Algorithmus verwendet deutlich weniger Multiplikationen, vor allem für große *n*.

Eingaben sind Basis und Exponent der zu berechnenden Potenz, die Ausgabe ist die Potenz.

```
SchnellExp[Basis_, Exponent_] :=
 Module[{out = 0, EB, ML, L},
  EB = Reverse[IntegerDigits[Exponent, 2] ];
  ML = {Basis};
  L = Basis;
  For[i = 2, i ≤ Length[EB], i++,
   AppendTo[ML, ML[[i - 1]] * ML[[i - 1]]]];
  For[j = 2, j ≤ Length[EB], j++,
   If[EB[[j]] ≠ 0, L = L * ML[[j]] ] ];
  L
 ]
```

Der Exponent wird hierbei in eine Bitliste zerlegt, diese hat Floor[$\text{Log}_2\ n$]+1 Elemente, Weiterhin wird eine Liste mit sukzessiven Quadrierungen angefertigt, diese hat die gleiche Länge und verwendet dafür auch Floor[$\text{Log}_2\ n$] Quadrate. Zum Schluss werden noch alle diese Quadrierungen zusammen multipliziert, was nochmal Floor[$\text{Log}_2\ n$] Multiplikationen sind. Somit kommt der Algorithmus mit 2*Floor[$\text{Log}_2\ n$] Multiplikationen aus, was deutlich weniger als n-1 ist, zwar wird nur noch die Umwandlung des Exponenten in Binärform gebraucht, dies ist allerdings in den meisten Fällen ein direkter Abgriff auf interne Strukturen und kein wirklicher Rechenaufwand.

Beispiel

```
SchnellExp[3, 179]
```

```
25 392 449 348 622 130 779 763 242 573 538 520 583 474 933 ⁖
  800 798 398 908 000 521 914 985 712 447 677 679 339 867
```

```
3^179
```

```
25 392 449 348 622 130 779 763 242 573 538 520 583 474 933 ⁖
  800 798 398 908 000 521 914 985 712 447 677 679 339 867
```

Zeitvergleich

```
Timing[SchnellExp[3, 1 790 000];]
```

```
{0.047, Null}
```

Folgendes zeigt die Rechenzeit für wiederholte Multiplikation, diese umständliche manuelle Mutiplikationsschleife muss sein, weil für eine normale Potenzierung in Mathematica selbst gute Algorithmen verwendet werden.

```
b = 3; e = 1 709 000; i = 1; l = 1;
Timing[For[i = 1, i <= e + 1, i++, l = l * b]]
```

```
{32.246, Null}
```

Wie zu erwarten, ist der oben beschriebene Algorithmus erheblich schneller.

Zur Vollständigkeit noch ein Timingvergleich zum *Mathematica* internen Algorithmus.

```
Timing[3^1790000;]
```

```
{0.031, Null}
```

Wie zu erwarten war, ist der interne Algorithmus noch ein bisschen schneller, jedoch sind selbst für siebenstellige Exponenten noch keine signifikanten Vorteile zu erkennen. Die sukzessive Multiplikation hingegen ist zeitlich wesentlich schlechter.

Algorithmen zum Wechsel zwischen Text, CharCode und Bitlisten

Für die meisten Algorithmen werden Eingaben in Binärform oder Zahlen z.b. CharCode benötigt, folgende Algorithmen wandeln beliebige Texte um und wandeln die Ergebnisse wieder zurück.
Je nach Algorithmus ist die Eingabe entweder ein string, eine CharList oder eine BitList. Die Algorithmennamen sind der Syntax entsprechend geschrieben Input-to-Output, sodass Text den Datentyp string angibt, Char also der CharakterCode steht für eine Charlist und Bit steht für die Bitlist, welche eine List aus binären Werten, der Länge ein Byte pro Charakter, ist.

```
TextToChar[string_] :=
 ImportString[string, {"Binary", "Byte"}]
```

Beispiel:

```
TextToChar["Test Text 1234"]
```

```
{84, 101, 115, 116, 32, 84,
 101, 120, 116, 32, 49, 50, 51, 52}
```

```
CharToText[CharList_] :=
 FromCharacterCode[CharList]
```

Beispiel:

```
CharToText[{84, 101, 115, 116, 32, 84, 101,
  120, 116, 32, 49, 50, 51, 52}]
```

```
Test Text 1234
```

```
TextToBit[string_] :=
 ImportString[string, {"Binary", "Bit"}]
```

Beispiel:

```
TextToBit["Test Text 1234"]
```

```
{0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1,
 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0,
 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0,
 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0,
 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0}
```

```
BitToText[BitList_] :=
 Module[{out = {}, char, i, j},
  For[j = 0, j <= Length[BitList] / 8 - 1, j++,
   char = 0;
   For[i = 1, i ≤ 8, i++,
    char = char + BitList[[8 * j + i]] * 2^(8 - i)
   ];
   AppendTo[out, char]
  ];
  FromCharacterCode[out]
 ]
```

Beispiel:

```
BitToText[{0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0,
  1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0,
  1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0,
  1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1,
  0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0,
  0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0,
  0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0,
  1, 0, 0}]
```

```
Test Text 1234
```

Algorithmus zu RSA Schlüsselerzeugung

Hier wird eine Algorithmus bereitgestellt, welcher ein Schlüsselpaar, für eine RSA-Chiffre, liefert. Die Eingabe ist eine Ganze Zahl G, welche angibt wie groß der RSA-Modul mindestens sein soll. Die eigentlich wählbare Zahl e wird hierbei per Random passeng gewählt. Die Ausgabe sind der öffentliche Schlüssel e, der private Schlüssel d und der RSA-Modul n. Der durch diesen Algorithmus generierte RSA Modul ist immer größer als G. Damit dieser Algorithmus funktioniert müssen alle vorherigen Algorithmen initialisiert sein.

```
RSAKEY[G_] := Module[{out = {}, p, q, n, e, d, P},
  p = NextPrime[Floor[Sqrt[G]]];
  q = NextPrime[p];
  n = p * q;
  P = (p - 1) * (q - 1);
  e = Floor[Random[Real, G]];
  While[EuclidAlgo[e, P] ≠ 1,
   e = Floor[Random[Real, G]]];
  d = EEA[e, P][[2]];
  d = Mod[d, P];
  out = {e, d, n}
 ]
```

```
RSAKEY[300]
```

```
{151, 139, 437}
```

Für eine Schlüsselerzeugung mit mindestens 300 Elementen ist der öffentliche Schlüssel e = 151, der private Schlüssel d = 139, und der RSA-Modul n = 437. Dies ist natürlich nicht die einzige Möglichkeit, denn je nach Variation des gewählten e (hier Random ausgewählt) ergeben sich unterschiedliche Schlüsselpaare. Weiterhin wird hierbei p und q so gewählt, das sie nah beieinander liegen und die nächstgrößeren Primzahlen an der Quadratwurzel von G sind. Durch Variation dieser Primzahlen lassen sich ebenfalls andere RSA Systeme bilden.

Eine Beispielverschlüsselung mit den eben erzeugten Schlüsseln.

```
c = RSA[150, 151, 437]
```

```
100
```

```
DeRSA[c, 139, 437]
```

```
150
```

7 Fallbeispiel mit RSA

Bisher wurden die Kryptosysteme nur in Testbedingungn gezeigt und die Beispiele waren speziell angepasst um einen kleinen Eindruck für das System zu erhalten. Hier wird nun eine realistische Nachricht mit RSA verschlüsselt um ein komplettes Übertragungsszenario zu zeigen.

Die zu verschlüsselnde Nachicht sei: "Dies ist die Bachlorarbeit von Daniel Vander Putten"

Erster Schritt ist eine Umkodierung anhand einer Zeichentabelle, hierfür eignet sich die ASCII-Tabelle. Einen Algorithmus für solche Umkodierung ist in den Algorithmen bereits vorhanden.

```
P = TextToChar[
  "Dies ist die Bachlorarbeit von Daniel
    Vander Putten"]
```

```
{68, 105, 101, 115, 32, 105, 115, 116, 32, 100,
 105, 101, 32, 66, 97, 99, 104, 108, 111, 114,
 97, 114, 98, 101, 105, 116, 32, 118, 111, 110,
 32, 68, 97, 110, 105, 101, 108, 32, 86, 97, 110,
 100, 101, 114, 32, 80, 117, 116, 116, 101, 110}
```

Diese Charaktercode Liste wird Eintrag für Eintrag mit RSA verschlüsselt. Hierzu erzeugen wir per Algorithmus ein Schlüsselpaar. Der ASCII Code umfast 255 Zeichen, somit würde eine RSA-Modulgröße von 256 genügen. Da wir allerdings Wissen, das der ASCII Code sehr gängig zur Zeichenkodierung ist und außerdem das größte auftretende Element 118 ist, wählen wir die untere Modulgrenze kleiner und erzeugen ein Schlüsselpaar in einem Modul mit etwas mehr als 120 Elementen.

```
{e, d, n} = RSAKEY[120]
```

```
{53, 77, 143}
```

Die Schlüssel sind e = 53, d = 77 und Modulgröße n ist 143.

c wird unser übertragenes Chiffrat

```
c = {}; For[j = 1, j <= Length[P], j++,
  AppendTo[c, RSA[P[[j]], e, n]]];
c
```

```
{74, 40, 30, 59, 54, 40, 59, 51, 54, 133, 40, 30, 54,
 66, 80, 99, 26, 36, 89, 108, 80, 108, 76, 30, 40, 51,
 54, 105, 89, 132, 54, 74, 80, 132, 40, 30, 36, 54, 47,
 80, 132, 133, 30, 108, 54, 71, 13, 51, 51, 30, 132}
```

Zum Übertragen auf einer Digitalenleitung wird dies in eine Bitliste konvertiert.
Diese Bitliste könnte dann hardwaretechnisch mit einer weiteren Blockchiffre koderit werden.

Der Empfänger bekommt also die Bitliste und wandelt sie wieder in den Charakter Code um. Weiterhin wendet er zeichenweise die RSA Dekodierung mit seinem geheimen Schlüssel d an. (In diesem Beispiel wird c einfach weiterverwendet.)

```
p = {}; For[i = 1, i <= Length[c], i++,
 AppendTo[p, RSA[c[[i]], d, n]]];
p
```

```
{68, 105, 101, 115, 32, 105, 115, 116, 32, 100,
 105, 101, 32, 66, 97, 99, 104, 108, 111, 114,
 97, 114, 98, 101, 105, 116, 32, 118, 111, 110,
 32, 68, 97, 110, 105, 101, 108, 32, 86, 97, 110,
 100, 101, 114, 32, 80, 117, 116, 116, 101, 110}
```

```
CharToText[p]
```

```
Dies ist die Bachlorarbeit
  von Daniel Vander Putten
```

Somit ist die verschlüsselte Übertragung erfolgreich beendet.

Quellen

Grafiken von angebenen Internetseiten

Vorlesungsmitschriften Kryptographie, Computer Algebra

Einführung in die Kryptographie, Buchman 4. erweiterte Auflage

Beweise:
Satz der Divison mit Rest:
Vorlesungsmitschrift Computer Algebra

Lemma von Bézout
http://de.wikipedia.org/wiki/Lemma_von_B%C3%A9zout#Beweis

Satz von Fermat
http://de.wikibooks.org/wiki/Beweisarchiv:_Zahlentheorie:_Elementare_Zahlentheorie:_Kleiner_Satz_von_Fermat

Ich versichere hiermit, dass ich diese Arbeit selbstständig verfasst und keine an
die angegebenen Quellen und Hilfsmittel benutzt habe.

Daniel Vander Putten