FACHHOCHSCHULE **TRIER**
Hochschule für Technik, Wirtschaft und Gestaltung
University of Applied Sciences
Informatik

Implementierung kryptologischer Verfahren in AXIOM

Implementation of cryptologic algorithms in AXIOM

Peter Karski

Betreuer: Prof. Dr. Ing. Alfred Scheerhorn

Trier, 26.02.2007

Kurzfassung

Im Rahmen dieser Arbeit wurden einige der wichtigsten kryptologischen Algorithmen in AXIOM implementiert. AXIOM ist ein Computeralgebra-System, das für mathematische Berechnungen und zur Entwicklung von mathematischen Algorithmen genutzt werden kann. Zu den implementierten Algorithmen zählen einige Chiffre-Funktionen unter denen sich auch der AES, als eine der modernsten Chiffren, befindet. Als Hauptbestandteil dieser Arbeit gelten jedoch die Algorithmen, die auf Blockchiffren angewendet werden können. Dazu zählen die gängigen Betriebsmodi für Blockchiffren, die Feistel Iteration, die kaskadierte Chiffre, die Funktion für Brute Force Suchen sowie die Berechnung der Abhängigkeitsmatrix für Blockchiffren. Diese Algorithmen wurden so implementiert, dass es möglich ist diese auf alle implementierten und in Zukunft hinzukommenden Chiffre-Funktionen anzuwenden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Arbeit beschäftigt sich mit der Umsetzung von kryptologischen Algorithmen in AXIOM. AXIOM ist ein Computeralgebra-System dessen Entwicklung bereits 1976 unter dem Namen Scratchpad bei IBM begann. Ziel war es ein System zu erstellen mit dem neue mathematische Algorithmen entwickelt und untersucht werden können. In den 90ern wurde das Projekt in AXIOM umbenannt und an Numerical Algorithms Group (NAG) in London verkauft, die das Produkt erstmals kommerziell vertrieben hat. Da der Erfolg ausblieb, u.a. aufgrund schlechter Vermarktungsstrategien, verlor AXIOM den Wettbewerb gegen andere Computeralgebra-Systeme und wurde im Oktober 2001 vom Markt genommen. Daraufhin erklärte sich NAG bereit AXIOM freizugeben. Seit August 2003 steht AXIOM als freie Open Source Software auf der Website von Savannah [SAVA] zum Download bereit.

Da in AXIOM bislang keine Verfahren aus dem Bereich der Kryptographie verfügbar sind, sollen auf der Basis dieses Systems einige wichtige Verfahren für den Einsatz in der Lehre implementiert werden.

2 Aufgabenstellung und Zielsetzung

Die Aufgabenstellung beinhaltet die Umsetzung der wichtigsten Verfahren für die symmetrische Kryptographie. Dazu zählen zunächst klassische und moderne Chiffre-Funktionen. Für diese Chiffren sollen anschließend Funktionen erstellt werden, die auf diese Chiffren angewendet werden können. Darunter fallen die wichtigsten Betriebsmodi für Blockchiffren, die Feistel Iteration, die Kaskadierte Chiffre, die Brute Force Suche und die Berechnung von Abhängigkeitsmatrizen. Dabei sollen alle Funktionen die Option anbieten auch Zwischenergebnisse auszugeben, damit der Verlauf der Berechnung besser nachvollzogen werden kann.

AXIOM bietet für die Programmierung mehrere Möglichkeiten. Die einfachste ist der Einsatz von Makros. Diese eignen sich aber aufgrund der schlechten Performanz und schweren Modularisierung nicht für größere Projekte. Eine weitere Möglichkeit ist die Programmiersprache SPAD (abgeleitet aus Scratchpad). In dieser ist der Großteil der AXIOM Pakete und Datentypen implementiert. Außerdem ist der Compiler bereits in AXIOM integriert. Mittlerweile wurde als Nachfolger von SPAD die Programmiersprache ALDOR ausgewählt, die eine etwas erweiterte Funktionalität bietet, allerdings bislang nur für Unix-basierte Betriebssysteme verfügbar ist. Unter dem Gesichtspunkt des Einsatzes des Ergebnisses dieser Arbeit in der Lehre wurde SPAD für die Umsetzung gewählt, da es sowohl auf Windows sowie auf Unix-basierten Systemen eingesetzt werden kann und neben AXIOM keine weiteren Installationen erfordert.

Die Chiffre-Funktionen werden in zwei Gruppen aufgeteilt. In klassische Chiffren, die auf Zeichen (Alphabeten) arbeiten und in moderne Chiffren die bit-basiert sind.

Die Funktionen, die auf Blockchiffren angewendet werden sollen, werden in zwei parametrisierten Datentypen realisiert. Zwei Datentypen sind notwendig, da die Verarbeitung von klassischen Chiffren sich von der der modernen Chiffren unterscheidet. Über einen dieser Parameter kann eine Chiffre-Funktion übergeben werden, die bei Funktionsaufrufen aus diesem Datentypen verwendet wird. Daher ist es notwendig, dass jeweils alle klassischen und alle modernden Chiffre-Funktionen den selben Aufbau im Funktionskopf aufweisen. Dies wird zum Teil mittels Wrapper-Funktionen realisiert.

Für die Ausgabe von Zwischenergebnissen wird ein Großteil der Funktionen mit einem Verbose-Parameter versehen. Im Verbose Modus 0 wird nur das Ergebnis ausgegeben. Modus 1 dient der Ausgabe der Zwischenergebnisse. Komplexe Algorithmen besitzen einen Modus 2 zur noch detailierteren Ausgabe der Berechnung.

Zur Erleichterung des Umgangs mit den Funktionen werden außerdem eine Reihe von Hilfsfunktionen, vor allem zur Konvertierung zwischen verschiedenen Datentypen, implementiert.

3 AXIOM

Dieses Kapitel soll den Einstieg in die Arbeit mit AXIOM vereinfachen. Es wird kurz auf die Syntax der AXIOM Anweisungen eingegangen und es werden einige spezifische Befehle die bei der Erkundung der AXIOM Umgebung nützlich sind erklärt. Mit diesen Informationen sind die Beispiele aus dieser Arbeit nachvollziehbar.

An dieser Stelle sei auch auf das AXIOM Buch „AXIOM – The 30 Year Horizon“ [AXIOM] hingewiesen. Es enthält neben der ausführlichen Dokumentation vieler Funktionen von AXIOM auch einige Kapitel zur Einführung. Bei der Arbeit mit AXIOM kommt man um dieses Buch nicht herum.

3.1 Erste Schritte

Nach dem Start der AXIOM-Umgebung erscheint ein Prompt, bestehend aus einer Zahl und einem Pfeil. Die Zahl ist dabei ein Anweisungszähler und wird nach jeder AXIOM-Anweisung inkrementiert. Nun können die ersten Anweisungen eingegeben werden.

```
(1) -> 4+8
                                        12
                                                    Type: PositiveInteger
```

Neben dem Ergebnis, hier 12, wird auch der Datentyp des Ergebnisses angezeigt, hier `PositiveInteger`. Genauer genommen ist `PositiveInteger` unter AXIOM ein Domain. Domains sind Datentypen und stellen meist algebraische Strukturen dar, auf die eine Reihe von Funktionen, die ebenfalls Teil dieses Domain sind, angewendet werden können. Ein Sonderfall von Domains sind die Pakete, die nur eine Sammlung von Funktionen enthalten. In diesem Beispiel ist `+` eine Funktion des Domains `PositiveInteger`. Dieser Datentyp repräsentiert alle positiven ganzen Zahlen.

```
(2) -> x := 12
                                        12
                                                    Type: PositiveInteger
```

Mit dem Operator `:=` werden unter AXIOM Zuweisungen ausgeführt. Hier wird der Variable x der Wert 12 zugewiesen. Variablen können implizit genutzt werden und müssen nicht vorher deklariert werden. Bei so einer Zuweisung ist es auch möglich den Datentyp, zu der die Variable gehören soll, zu definieren. Dies geschieht mit Hilfe des Operators `":"`.

```
(3) -> y : Integer := 12
                                                    12
                                                                                    Type: Integer
```

Dabei muss darauf geachtet werden, das die Zahl 12 auch als Objekt des Datentyps `Integer` dargestellt werden kann. Diese Methode kann nur angewendet werden, solange der Variable noch kein Datentyp zugeordnet wurde. Später kann allerdings in vielen Fällen der Operator "`::`" verwendet werden, um ein Objekt eines Datentyps in eins eines anderen Datentyps zu transformieren.

```
(4) -> y::PositiveInteger
                                                    12
                                                                            Type: PositiveInteger
```

Das Ergebnis der letzten Berechnung kann immer mit `%` abgerufen werden. Eine andere Möglichkeit auf ein älteres Ergebnis zuzugreifen ist der Operator `%%(n)`, wobei `n` eine Anweisungsnummer ist. Dabei wird also auf das Ergebnis der `n`-ten Anweisung zurückgegriffen.

```
(5) -> %
                                                    12
                                                                            Type: PositiveInteger

(6) -> %%(3)
                                                    12
                                                                                    Type: Integer
```

Oft besitzen Funktionen in mehreren Datentypen dieselbe Bezeichnung. Zum Beispiel gibt es die Funktion `gcd` ("greatest common divisor") unter anderem in dem Datentyp `PositiveInteger` und `NonNegativeInteger`. Mit dem `$`-Zeichen ist es möglich Funktionen aus bestimmten Datentypen zu selektieren.

```
(7) -> gcd(642,852)$PositiveInteger
                                                     6
                                                                            Type: PositiveInteger
```

Dies sind die nötigsten Grundlagen der AXIOM Syntax. Nun müssen noch die Datentypen und Pakete und deren Funktionen erkundet werden.

Im Folgenden wird bei den Beispielen aufgrund der besseren Übersichtlichkeit auf die Angabe der Anweisungsnummer verzichtet.

3.2 AXIOM-Befehle

In der Umgebung stehen diverse AXIOM-Befehle zur Verfügung. Diese können an einem vorangestellten "`)`" erkannt werden.

`)show`

Mit diesem Befehl lassen sich Informationen über alle verfügbaren Funktionen eines Datentyps oder eines Pakets anzeigen. Dem Befehl muss ein Domain-Name übergeben werden, also z.B. `)show PositiveInteger`.

`)display`

Damit lassen sich diverse Informationen zu Funktionen und der Arbeitsumgebung abrufen. Mit `)display op Funktionsname` kann man sich Informationen zu einer bestimmten Funktion anzeigen lassen. Mit `)display names` werden alle belegten Variablennamen der Arbeitsumgebung ausgegeben.

`)what`

Mit what kann nach Funktionen (`)what op gcd`), nach Paketen (`)what packages int`) oder nach Datentypen (`)what domains Integer`) gesucht werden, die dem übergebenen Suchmuster entsprechen.

Mit diesen Befehlen lassen sich alle Datentypen und Pakete von AXIOM erkunden. Weitere Befehle können dem AXIOM Buch [AXIOM] entnommen werden.

4 Datentypen

In diesem Kapitel werden kurz die im Rahmen der Implementierung genutzten Datentypen (Domains) vorgestellt. Darunter fallen folgende bereits in AXIOM vorgegebene Datentypen

- Integer
- NonNegativeInteger
- PositiveInteger
- String
- Character
- List
- OneDimensionalArray
- TwoDimensionalArray
- Symbol

sowie folgende selbst definierte:

- Byte
- FBits
- Alphabet

Die Datentypen werden in den nachfolgenden Abschnitten erläutert. Der Umgang mit diesen Datentypen ist eine Grundvoraussetzung für die Nutzung der kryptologischen Funktionen.

4.1 AXIOM Datentypen

AXIOM bietet bereits hunderte von vordefinierten, mathematischen Datentypen. Bei der Umsetzung der kryptologischen Funktionen wurden hauptsächlich Typen verwendet, die so oder so ähnlich auch aus höheren Programmiersprachen bekannt sind.

Eine Funktionsübersicht zu den jeweiligen Datentypen kann wie bei allen Datentypen und Paketen mit dem AXIOM-Befehl `)show` abgerufen werden.

4.1.1 Integer

Der Typ `Integer` wird von AXIOM zur Ganzzahldarstellung genutzt. Er gehört zu den Kerndatentypen, so dass er ohne einen Konstruktoraufruf implizit genutzt werden kann.

```
-5
                                        -5
                                                              Type: Integer
```

Die grundlegenden arithmetischen Funktionen, sowie einige weitere sind für den Datentyp `Integer` verfügbar.

```
5*3+2
                                       17
                                                    Type: PositiveInteger
```

Wie aus dem Beispiel hervorgeht besitzt AXIOM verschiedene Ganzzahl-Typen. `PositiveInteger` (Abk. PI) für Zahlen größer Null und `NonNegativeInteger` (Abk. NNI) für nichtnegative Zahlen sind Subtypen (Subdomains) von `Integer`.

Die Größe der Werte, die durch diese Typen dargestellt werden, ist nur durch den Speicher begrenzt, der AXIOM zur Verfügung steht. Dadurch ist es möglich mit sehr großen Zahlen zu rechnen.

4.1.2 String und Character

Der Datentyp `String` dient der Darstellung von Zeichenketten. Sie werden in AXIOM in doppelte Anführungszeichen gesetzt.

```
"test"
                                    "test"
                                                              Type: String
```

Einzelne Zeichen sind vom Typ `Character` und können mit dem Konstruktoraufruf `char` erzeugt werden. Als Parameter wird ein String der Länge 1 übergeben.

```
char("z")
                                       z
                                                           Type: Character
```

Da Strings Zeichenketten sind, können auch einzelne Zeichen ausgelesen oder der ganze String in eine Liste von Zeichen konvertiert werden.

```
"abcdefgh".5
                                       e
                                                           Type: Character

entries("abcdefgh")
                                [a,b,c,d,e,f,g,h]
                                                      Type: List Character
```

4.1.3 List

Mehrere Objekte eines Typs können in einer Liste gruppiert werden. Sie werden in AXIOM in eckigen Klammern und durch Kommata getrennt dargestellt und können auch so eingegeben werden.

```
[1,21,-33,25]
                                    [1,21,-33,25]
                                                          Type: List Integer
```

Eine andere Möglichkeit eine Liste zu erzeugen ist der Konstuktor `construct` des Datentyps `List`.

```
construct("a","bb","ccc")
                                  ["a","bb","ccc"]
                                                          Type: List String
```

4.1.4 OneDimensionalArray und TwoDimensionalArray

Mit diesen beiden Typen können ein- bzw. zweidimensionale, indizierte Felder eines Datentyps angelegt werden. Felder haben im Gegensatz zu Listen eine feste Länge, die nicht mehr geändert werden kann. Zur Erzeugung von eindimensionalen Feldern kann der Konstruktor `oneDimensionalArray` mit einer Liste als Parameter verwendet werden.

```
oneDimensionalArray([-1,2,-3,4])
                                      [-1,2,-3,4]
                                                  Type: OneDimensionalArray Integer
```

Zweidimensionale Felder werden mit dem Konstruktor `new` erzeugt. Die Parameter geben die Anzahl der Zeilen, die Anzahl der Spalten und den Wert mit dem das Array bei der Erstellung gefüllt werden soll an.

```
arr := new(3,5,0)$TwoDimensionalArray(Integer)
                                    +0  0  0  0  0+
                                    |             |
                                    |0  0  0  0  0|
                                    |             |
                                    +0  0  0  0  0+
                                                  Type: TwoDimensionalArray Integer
```

Das Feld kann anschließend z.B. mit der Funtion `setelt` modifiziert werden.

```
setelt(arr,1,1,23)
                                         23
                                                          Type: PositiveInteger

arr
                                    +23 0  0  0  0+
                                    |             |
                                    |0  0  0  0  0|
                                    |             |
                                    +0  0  0  0  0+
                                                  Type: TwoDimensionalArray Integer
```

4.1.5 Symbol

Symbole die in AXIOM eingegeben werden, die dem System nicht bekannt sind, werden automatisch als Variablenbezeichnungen interpretiert. Um eine Eingabe weiterhin als Symbol nutzen zu können, kann man den Datentyp `Symbol` benutzen.

```
encrypt::Symbol
                                        encrypt
                                                                    Type: Symbol
```

4.2 Neu implementierte Datentypen

Die weiteren Abschnitte dieses Kapitels stellen kurz die selbst erstellten Datentypen vor.

4.2.1 Byte

Für byte-orientierte Algorithmen wurde der Datentyp `Byte` erstellt. Er repräsentiert, wie in der Informatik üblich, einen Vektor mit 8 Bits und somit die ganzzahligen Werte von 0 bis 255. Der Datentyp basiert auf `IntegerMod(256)`.

```
b : Byte := 42
                                        2A
                                                                    Type: Byte
```

Wie aus dem Beispiel hervorgeht werden Bytes in Hexadezimal-Darstellung angezeigt. Dies hat den Vorteil das längere Listen von Bytes übersichtlicher sind. Natürlich kann ein `Byte`-Objekt in eine Dezimalzahl konvertiert werden.

```
b::Integer
                                        42
                                                                    Type: Integer
```

Neben den arithmetischen Operationen die `Byte` aus `InterMod(256)` erbt wurden noch ein paar Bitoperationen implementiert:

```
Xor(b,17)
                                        3B
                                                                    Type: Byte

Or(b,71)
                                        6F
                                                                    Type: Byte

And(b,71)
                                        02
                                                                    Type: Byte



shift(b,2)
                                        A8
                                                                    Type: Byte

shift(b,-2)
```

```
                                        0A
                                                                    Type: Byte
```

4.2.2 FBits

Der Datentyp `FBits` repräsentiert beliebig lange Folgen von Bits. Zur Erstellung eines `FBits`-Objekts stehen zwei Kontruktoren zur Verfügung, `new` und `construct`.

```
new(16, false)$FBits
                                00000000 00000000
                                                                  Type: FBits

construct([false,true,true,false])$FBits
                                      0110
                                                                  Type: FBits
```

`FBits` ist abgeleitet von `IndexedBits(1)` und erbt alle Funktionen von diesem Datentyp (darunter auch die oben erwähnten Konstruktoren). Der einzige Unterschied zwischen `FBits` und `IndexedBits(1)` liegt in der Darstellung des Objekts. `FBits` Objekte werden in 8 Bit Gruppen und ohne Anführungszeichen dargestellt.

4.3 Alphabet

Viele klassische Chiffren, wie z.B. die Caesar-Chiffre, arbeiten mit Alphabeten von Zeichen und nicht mit Bits, wie es bei modernen Chiffren der Fall ist. Da diese Chiffren nicht für feste Alphabete vorgesehen sind, dient dieser Datentyp für die Definition eines Eingabealphabets. Objekte des Datentyps `Alphabet` können beliebig lange Zeichenketten aufnehmen, die eine Eingabemenge repräsentieren.

Erzeugt wird ein Alphabet mit Hilfe des Konstruktors `alphabet` der einen String als Parameter bekommt.

```
a : Alphabet := alphabet("abcd1234")
                                    "abcd1234"
                                                              Type: Alphabet
```

Die Zeichen in einem Alphabet haben eine feste Reihenfolge und können auf natürliche Zahlen abgebildet werden. Gleichzeitig können auch natürliche Zahlen auf Zeichen des Alphabets abgebildet werden. Dabei beginnt die Numerierung bei 0.

```
index(char "1",a)
                                       4
                                                        Type: PositiveInteger

lookup(4,a)
                                       1
                                                              Type: Character
```

Mit den Funktionen `minIndex(Alphabet)` (per Default immer 0) und `maxIndex(Alphabet)` lassen sich die Grenzen des Alphabets abfragen.

Jedes Zeichen kommt in einem Alphabet nur einmal vor. Daher werden doppelte Zeichen bei der Erzeugung und Erweiterung eines Alphabets automatisch herausgefiltert.

```
alphabet("abcdd")
                                    "abcd"
                                                                Type: Alphabet

concat(a,"456")
                                "abcd123456"
                                                                Type: Alphabet
```

Weitere nützliche Funktionen des Datentyps `Alphabet` sind die vorgefertigten Alphabete und die `member?` Funktionen.

```
digit()$Alphabet
                                "0123456789"
                                                                Type: Alphabet

lowerCase()$Alphabet
                        "abcdefghijklmnopqrstuvwxyz"
                                                                Type: Alphabet

alphabetic()$Alphabet
          "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
                                                                Type: Alphabet

member?(char("c"),digit())$Alphabet
                                   false
                                                                 Type: Boolean

member?("3des",lowerCase())$Alphabet
                                   false
                                                                 Type: Boolean

member?("aes",lowerCase())$Alphabet
                                    true
                                                                 Type: Boolean
```

5 Klassische Chiffren

Das Paket `ClassicCiphers` enthält 2 Beispiele für klassische Chiffren. Die Caesar-Chiffre und die Vigenère-Chiffre.

5.1 Caesar Chiffre

Die Caesar-Chiffre ist eine monoalphabetische Substitutionschiffre, die nach Julius Caesar benannt wurde. Bei der Verschlüsselung wird jedes Zeichen des Eingabetextes um eine bestimmte Anzahl von Positionen im Alphabet zyklisch verschoben. Im Ursprung wurde diese Chiffre mit dem lateinischen Standardalphabet (A-Z) und 3 Positionsverschiebungen (= Schlüssel) für jedes Zeichen genutzt. Die Funktion `caesar` im Paket erlaubt es ein eigenes Alphabet sowie einen eigenen Schlüssel anzugeben. Der allgemeine Funktionsaufruf lautet:

```
caesar(n:String,k:String,a:Alphabet,direction:Symbol,v:NNI)
```

`n` ist der Klartext und `k` ist ein String der Länge 1 der als Schlüssel verwendet wird. Die Verschlüsselung wird über dem Alphabet `a` durchgeführt. Die Richtung der Berechnung wird durch das Symbol `direction` angegeben. Der letzte Parameter `v` gibt den Verbose Modus an, ist aber ohne Bedeutung. Er ermöglicht die Verwendung des Datentyps `CryptographicOperationsChar` (vgl. Kapitel 7) mit dieser Funktion. Ein Beispiel für eine Verschlüsselung ist:

```
caesar("plaintext","c",lowerCase(),encrypt,0)
                                "rnckpvgzv"
                                                              Type: String
```

Die Entschlüsselung des Chiffretextes im gleichen Alphabet mit gleichem Schlüssel sollte wieder den Klartext ergeben.

```
caesar("rnckpvgzv","c",lowerCase(),decrypt,0)
                                "plaintext"
                                                              Type: String
```

5.2 Vigenère Chiffre

Die Vigenère Chiffre, benannt nach dem französischem Kryptografen Blaise de Vigenère, ist eine polyalphabetische Substitutionschiffre und ähnelt sehr der Caesar-Chiffre. Der einzige Unterschied liegt in der Anwendung des Schlüssels, der jetzt aus mehreren Zeichen besteht. Anstatt jedes Klartextzeichen mit demselben Schlüsselzeichen zu verschlüsseln wird bei der Vigenère Chiffre mit dem aktuellen

Zeichen des Schlüssels chiffriert. Das heißt das erste Klartextzeichen wird mit dem ersten Schlüsselzeichen chiffriert, das Zweite mit dem zweiten Schlüsselzeichen usw. Ist der Schlüssel zu Ende wird wieder das erste Schlüsselzeichen benutzt.

Die Funktion `vigenere` wird wie folgt aufgerufen. Dabei entsprechen die Parameter denen der `caesar` Funktion.

```
vigenere(n:String,k:String,a:Alphabet,direction:Symbol,v:NNI)
```

Der Schlüssel `k` ist nun ein String beliebiger Länge. Es folgt ein Beispiel für eine Ver- und eine Entschlüsselung.

```
vigenere("plaintext","key",lowerCase(),encrypt,0)
                                "zpysrrobr"
                                                          Type: String

vigenere("zpysrrobr","key",lowerCase(),decrypt,0)
                                "plaintext"
                                                          Type: String
```

Ist die Länge des Klartextes nicht ein Vielfaches von der Länge des Schlüssels, so werden am Ende der Verschlüsselung nur die ersten Zeichen des Schlüssels verwendet. Ebenso wird vorgegangen falls der Klartext kürzer ist als der Schlüssel.

6 Blockchiffren

Dieses Kapitel beschreibt den Umgang mit den folgenden implementierten Blockchiffren:

- Mul17
- Mul257
- Advanced Encryption Standard

Im Gegensatz zum vorherigen Kapitel wird hier, wie bei allen modernen Chiffren, mit Bits ($\{0,1\}^n$) als Eingabe-, Schlüssel- und Ausgabemenge gerechnet.

6.1 Mul17, Mul257

Die beiden Chiffrier-Funktionen sind sehr einfache Blockchiffren, die für Demonstrationszwecke von anderen Funktionen die auf Blockchiffren angewendet werden (z.B. Blockverschlüsselungen) gut geeignet sind. `Mul17` ist eine 4-bit und `Mul257` eine 8-bit Chiffre:

$$\text{mul17: } \{0,1\}^4 \times \{0,1\}^4 \rightarrow \{0,1\}^4$$

$$\text{mul257: } \{0,1\}^8 \times \{0,1\}^8 \rightarrow \{0,1\}^8$$

6.1.1 Algorithmus

Die Werte der Eingabemenge von `mul17` können als Elemente der multiplikativen Halbgruppe $\mathbf{Z}_{16}$ betrachtet werden. Durch Addition von 1 werden diese in die multiplikative Gruppe $\mathbf{Z}^*_{17}$ überführt. Diese Gruppe besitzt die Ordnung 16, da 17 eine Primzahl ist. Ebenso wird mit dem Schlüssel verfahren. Die eigentliche Verschlüsselung wird anschließend durch eine einfache Multiplikation von Eingabewert und Schlüssel in $\mathbf{Z}^*_{17}$ realisiert. Abschließend wird das Ergebnis durch Subtraktion von 1 wieder in $\mathbf{Z}_{16}$ überführt.

Die Entschlüsselung erfolgt indem der Chiffretext mit dem Inversen des Schlüssels multipliziert wird.

Analog dazu wird Ver- und Entschlüsselung bei `mul257` durchgeführt. Der einzige Unterschied zu `mul17` besteht darin das in der Gruppe $\mathbf{Z}^*_{257}$ (Ordnung dieser Gruppe ist 256) gerechnet wird.

6.1.2 Anwendung

Die Funktionen `mul17` und `mul257` befinden sich in dem Paket `BitCiphers`. Als Parameter erwarten beide Funktionen den Klartext `n`, den Schlüssel `k`, die Chiffrierrichtung `d` und die Angabe des Verbose Modus v:

```
mul17(n:FBits,k:FBits,d:Symbol,v:NNI)

mul17(n:NNI,k:NNI,d:Symbol,v:NNI)

mul257(n:FBits,k:FBits,d:Symbol,v:NNI)

mul257(n:NNI,k:NNI,d:Symbol,v:NNI)
```

Der Klartext `n` und der Schlüssel `k` können entweder im Format `FBits` oder als `NonNegativeInteger` angegeben werden. Da `NNI` alle nicht-negativen Ganzzahlen umfaßt sollte darauf geachtet werden, das $n \in \mathbf{Z}_{16}$ bzw. $n \in \mathbf{Z}_{256}$.

```
key := construct([true,false,true,false])$FBits
                                    1010
                                                          Type: FBits

plain := construct([false,true,true,false])$FBits
                                    0110
                                                          Type: FBits

mul17(plain,key,encrypt,0)
                                    1000
                                                          Type: FBits

mul17(%,key,decrypt,0)
                                    0110
                                                          Type: FBits

mul257(85,31,encrypt,0)
                                     181
                                                 Type: PositiveInteger

mul257(%,31,decrypt,0)
                                      85
                                                 Type: PositiveInteger
```

Als Verbose Modus kann 0 oder 1 gewählt werden.

```
mul257(231,84,decrypt,1)
MUL257 CIPHER
Inverse of 84 = 127
Calculating: ((231+1)*(127+1) mod 257) - 1 = 165
                                     165
                                                 Type: PositiveInteger
```

6.2 Advanced Encryption Standard (AES)

Der AES ist eine Blockchiffre, das als Nachfolger von DES und 3DES im Oktober 2000 vom National Institute of Standards and Technology (NIST) als Standard bekannt gegeben wurde. Der AES Algorithmus besitzt eine variable Blockgröße `n` von 128, 192 oder 256 Bit und eine variable Schlüssellänge `k` von 128, 192 oder 256 Bit.

$$\text{aes: } \{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n$$

Die Implementierung des AES befindet sich in dem Paket `AESFunctions`.

6.2.1 Algorithmus

Der genaue Aufbau des Algorithmus würde den Rahmen dieses Dokuments sprengen. Daher wird hier nur ein kurzer Einblick in den Ablauf gegeben. Eine detaillierte Beschreibung kann der Federal Information Processing Standard 197 Publikation [FIPS-197] der NIST entnommen werden. Der folgende Pseudo-Code repräsentiert den groben Ablauf des AES.

```
aes(plaintext, ciphertext, key)
begin
      state = plaintext
      KeyExpansion(key)
      AddRoundKey(state, key)
      for round = 1 step 1 to Nr-1
            ByteSub(state)
            ShiftRows(state)
            MixColumns(state)
            AddRoundKey(state, key)
      end for
      ByteSub(state)
      ShiftRows(state)
      AddRoundKey(state, key)
      ciphertext = state
end
```

Der AES ist rundenbasiert, wobei der Rumpf der for-Schleife einer Runde entspricht. Die Rundenanzahl (`Nr`) ist abhängig von der Blockgröße des Klartextes und des Schlüssels und kann 10, 12 oder 14 betragen.

6.2.2 Teilfunktionen

Der AES besteht aus verschiedenen Teilfunktionen, die in Runden immer wieder aufgerufen werden. Die Bedeutung der einzelnen Funktionen wird in den folgenden Abschnitten erklärt. Das Paket `AESFunctions` wurde so geschrieben, das es auch möglich ist die Teilfunktionen des AES aufzurufen.

ByteSub

Diese Funktion führt eine monoalphabetische Verschlüsselung (Substitution) mittels einer S-Box auf dem übergebenen Block durch (→ Konfusion). Die Substitution läßt sich mathematisch herleiten, sie wird aber meistens, wie auch in dieser Implementierung, aufgrund der besseren Effizienz durch eine Look-Up-Table realisiert. Die Abbildung 6.1 zeigt diese Tabelle. Das Byte xy wird auf das Byte in der x-ten Zeile und y-ten Spalte abgebildet.

x \ y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	1	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	4	C7	23	C3	18	96	5	9A	7	12	80	E2	EB	27	B2	75
4	9	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	0	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	2	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	B	DB
A	E0	32	3A	A	49	6	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	8
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	3	F6	E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	D	BF	E6	42	68	41	99	2D	F	B0	54	BB	16

Abbildung 6.1: ByteSub Tabelle

Bei der Implementierung in AXIOM heißt die Funktion `byteSub` und bekommt einen Block `n` sowie die Anwendungsrichtung `d` als Parameter übergeben.

```
byteSub(n:TwoDimensionalArray(Byte),d:Symbol)
```

Einzelne Bytes lassen sich mit der Funktion `sbox` bzw. `invSbox` substituieren.

```
sbox(b:Byte)

invSbox(b:Byte)
```

ShiftRows

In der Funktion `shiftRows` wird die Reihenfolge der Bytes in einem Block vertauscht (→ Diffusion). Als Parameter erwartet sie einen Block `n` und die Anwendungsrichtung `d`.

```
shiftRows(n:TwoDimensionalArray(Byte),d:Symbol)
```

MixColumns

Bei dieser Funktion werden die Bits eines Block durchmischt (→ Diffusion). Die Parameter der Funktion `mixColumns` sind ein Block `n` und die Anwendungsrichtung `d`.

```
mixColumns(n:TwoDimensionalArray(Byte),d:Symbol)
```

AddRoundKey

In `addRoundKey` wird der aktuelle Rundenschlüssel mit einem Block bitweise XOR-verknüpft. Dies ist die einzige Stelle an der der Schlüssel in den Verschlüsselungsvorgang eingebunden wird. Aufgrund der selbstinversen Eigenschaft von XOR kann für die Entschlüsselung die gleiche Operation durchgeführt werden. An die Funktion muss der Block `n` und der Rundenschlüssel `rk` übergeben werden.

```
addRoundKey(n:TwoDimensionalArray(Byte),
rk:TwoDimensionalArray(Byte))
```

KeyExpansion

Der AES benutzt in jeder Verschlüsselungsrunde einen Rundenschlüssel. Die Funktion `keyExpansion` erweitert dabei die Länge des Schlüssels auf die dafür benötigte Anzahl an Bits. Die Anzahl der Bits ergibt sich aus der Blockgröße und der Rundenanzahl. Daher ist es nötig bei dem Funktionsaufruf neben dem Schlüssel `k` die Blockgröße `b` in Worten (32 Bit = 1 Wort) anzugeben.

```
mixColumns(n:TwoDimensionalArray(Byte),b:NNI)
```

In dem Paket befinden sich noch einige weitere Funktion (u.a. Unterfunktionen von `keyExpansion`, Rundenfunktionen), die ebenfalls in FIPS-192 definiert werden.

6.2.3 Anwendung

Zur Anwendung des AES stehen drei Funktion zur Verfügung, die den Klartextblock `n` und den Schlüssel `k` in verschiedenen Formen annehmen können. Neben `n` und `k` muss die Chiffrierrichtung `d` und der Verbose Modus `v` angegeben werden. Dabei müssen die vom AES unterstützten Block- und Schlüsselgrößen beachtet werden.

```
aes(n,k,d,v)
   n: OneDimensionalArray(Byte)
   k: OneDimensionalArray(Byte)
   d: Symbol
   v: NNI

aes(n:String,k:String,d:Symbol, v:NNI)

aes(n:FBits,k:FBits,d:Symbol, v:NNI)
```

Als Verbose Modus kann 0, 1 oder 2 gewählt werden. Nun folgen einige Aufrufbeispiele für 128-bit Verschlüsselungen mit dem AES.

```
n := construct([21,32,11,34,192,231,100,2,10,219,123,46,199, 203,1,77])
$OneDimensionalArray(Byte)
             [15,20,0B,22,C0,E7,64,02,0A,DB,7B,2E,C7,CB,01,4D]
                                                Type: OneDimensionalArray(Byte)

k := reverse!(copy(n))
             [4D,01,CB,C7,2E,7B,DB,0A,02,64,E7,C0,22,0B,20,15]
                                                Type: OneDimensionalArray(Byte)

aes(n,k,encrypt,0)
             [11,57,C2,CB,F5,DC,B7,E2,46,9A,EA,19,D3,F1,A3,98]
                                                Type: OneDimensionalArray(Byte)

aes(%,k,decrypt,0)
             [15,20,0B,22,C0,E7,64,02,0A,DB,7B,2E,C7,CB,01,4D]
                                                Type: OneDimensionalArray(Byte)

aes("Quick brown fox.","secret  password",encrypt,0)
                               "K"ô•Ó•ñt,]ı•y-"
                                                                  Type: String

aes(stringToBits("Quick brown fox."),
stringToBits("secret  password"),encrypt,0)
  01001011 00100010 10010011 10111010 11100000 00000001 10100100 01001010
  00001000 01110100 00101100 01011101 11010101 10111001 01111001 11110000
                                                                 Type: FBits
```

Benutzt man einen der Verbose Modi, so läßt sich der Ablauf der AES-Verschlüsselung genau nachvollziehen. Folgende Ausgabe wird bei Wahl des Verbose Modus 2 erstellt:

```
aes(n,k,encrypt,2)


+----------------+
| AES encryption |
+----------------+

Input vector:        [15,20,0B,22,C0,E7,64,02,0A,DB,7B,2E,C7,CB,01,4D]
Block size:          128 bits

Key vector: [4D,01,CB,C7,2E,7B,DB,0A,02,64,E7,C0,22,0B,20,15]
Key size:   128 bits

Number of rounds to do: 10

Expanded key:
  [4D, 01, CB, C7, 2E, 7B, DB, 0A, 02, 64, E7, C0, 22, 0B, 20, 15, 67, B6,
   92, 54, 49, CD, 49, 5E, 4B, A9, AE, 9E, 69, A2, 8E, 8B, 5F, AF, AF, AD,
   16, 62, E6, F3, 5D, CB, 48, 6D, 34, 69, C6, E6, A2, 1B, 21, B5, B4, 79,
   C7, 46, E9, B2, 8F, 2B, DD, DB, 49, CD, 13, 20, 9C, 74, A7, 59, 5B, 32,
   4E, EB, D4, 19, 93, 30, 9D, D4, 07, 7E, D4, A8, A0, 27, 8F, 9A, EE, CC,
   5B, 83, 7D, FC, C6, 57, 97, CA, 8F, 57, 37, ED, 00, CD, D9, 21, 5B, 4E,
   A4, DD, 9D, 19, 16, 94, 5B, 1E, 21, 79, 5B, D3, F8, 58, 00, 9D, 5C, 85,
   9D, 84, 01, CA, 04, 54, 20, B3, 5F, 87, D8, EB, 5F, 1A, 84, 6E, C2, 9E,
```

```
   85, EF, 0F, 0B, A5, 5C, 50, 8C, 7D, B7, 0F, 96, F9, D9, CD, 08, 86, 52,
   3F, 92, 23, 0E, 6F, 1E, 5E, B9, 60, 88, A7, 60, AD, 80]
Expanded key size: 1408 bits

--------------------------------------

Round 0:
 Start:      [15,20,0B,22,C0,E7,64,02,0A,DB,7B,2E,C7,CB,01,4D]
 Round key:  [4D,01,CB,C7,2E,7B,DB,0A,02,64,E7,C0,22,0B,20,15]
 Add RK:     [58,21,C0,E5,EE,9C,BF,08,08,BF,9C,EE,E5,C0,21,58]
Round 1:
 Start:      [58,21,C0,E5,EE,9C,BF,08,08,BF,9C,EE,E5,C0,21,58]
 ByteSub:    [6A,FD,BA,D9,28,DE,08,30,30,08,DE,28,D9,BA,FD,6A]
 ShiftRows:  [6A,DE,DE,6A,28,08,FD,D9,30,BA,BA,30,D9,FD,08,28]
 MixColumns:[19,DE,AD,6A,6C,FD,B1,24,3F,BA,B5,30,95,08,4C,D5]
 Round key:  [67,B6,92,54,49,CD,49,5E,4B,A9,AE,9E,69,A2,8E,8B]
 Add RK:     [7E,68,3F,3E,25,30,F8,7A,74,13,1B,AE,FC,AA,C2,5E]
Round 2:
 Start:      [7E,68,3F,3E,25,30,F8,7A,74,13,1B,AE,FC,AA,C2,5E]
 ByteSub:    [F3,45,75,B2,3F,04,41,DA,92,7D,AF,E4,B0,AC,25,58]
 ShiftRows:  [F3,04,AF,58,3F,7D,25,B2,92,AC,75,DA,B0,45,41,E4]
 MixColumns:[06,49,5A,15,6E,18,C5,66,7F,94,A1,DB,11,1D,40,1C]
 Round key:  [5F,AF,AF,AD,16,62,E6,F3,5D,CB,48,6D,34,69,C6,E6]
 Add RK:     [59,E6,F5,B8,78,7A,23,95,22,5F,E9,B6,25,74,86,FA]
Round 3:
 Start:      [59,E6,F5,B8,78,7A,23,95,22,5F,E9,B6,25,74,86,FA]
 ByteSub:    [CB,8E,E6,6C,BC,DA,26,2A,93,CF,1E,4E,3F,92,44,2D]
 ShiftRows:  [CB,DA,1E,2D,BC,CF,44,6C,93,92,E6,2A,3F,8E,26,4E]
 MixColumns:[CB,6B,5A,D8,01,99,4F,8C,5C,B7,A8,8E,9F,1C,2F,75]
 Round key:  [A2,1B,21,B5,B4,79,C7,46,E9,B2,8F,2B,DD,DB,49,CD]
 Add RK:     [69,70,7B,6D,B5,E0,88,CA,B5,05,27,A5,42,C7,66,B8]
Round 4:
 Start:      [69,70,7B,6D,B5,E0,88,CA,B5,05,27,A5,42,C7,66,B8]
 ByteSub:    [F9,51,21,3C,D5,E1,C4,74,D5,6B,CC,06,2C,C6,33,6C]
 ShiftRows:  [F9,E1,CC,6C,D5,6B,33,3C,D5,C6,21,74,2C,51,C4,06]
 MixColumns:[71,03,2F,E5,03,6A,9C,44,B5,55,CD,6B,69,DF,E4,ED]
 Round key:  [13,20,9C,74,A7,59,5B,32,4E,EB,D4,19,93,30,9D,D4]
 Add RK:     [62,23,B3,91,A4,33,C7,76,FB,BE,19,72,FA,EF,79,39]
Round 5:
 Start:      [62,23,B3,91,A4,33,C7,76,FB,BE,19,72,FA,EF,79,39]
 ByteSub:    [AA,26,6D,81,49,C3,C6,38,0F,AE,D4,40,2D,DF,B6,12]
 ShiftRows:  [AA,C3,D4,12,49,AE,B6,81,0F,DF,6D,38,2D,26,C6,40]
 MixColumns:[D7,42,EC,D6,4C,4E,08,DA,31,25,42,D3,B6,70,5C,17]
 Round key:  [07,7E,D4,A8,A0,27,8F,9A,EE,CC,5B,83,7D,FC,C6,57]
 Add RK:     [D0,3C,38,7E,EC,69,87,40,DF,E9,19,50,CB,8C,9A,40]
Round 6:
 Start:      [D0,3C,38,7E,EC,69,87,40,DF,E9,19,50,CB,8C,9A,40]
 ByteSub:    [70,EB,07,F3,CE,F9,17,09,9E,1E,D4,53,1F,64,B8,09]
 ShiftRows:  [70,F9,D4,09,CE,1E,B8,F3,9E,64,07,09,1F,EB,17,53]
 MixColumns:[2D,F7,21,AF,EE,D2,B5,12,85,56,EF,C8,5C,B8,2F,7B]
 Round key:  [97,CA,8F,57,37,ED,00,CD,D9,21,5B,4E,A4,DD,9D,19]
 Add RK:     [BA,3D,AE,F8,D9,3F,B5,DF,5C,77,B4,86,F8,65,B2,62]
Round 7:
 Start:      [BA,3D,AE,F8,D9,3F,B5,DF,5C,77,B4,86,F8,65,B2,62]
 ByteSub:    [F4,27,E4,41,35,75,D5,9E,4A,F5,8D,44,41,4D,37,AA]
 ShiftRows:  [F4,75,8D,AA,35,F5,37,41,4A,4D,E4,9E,41,27,D5,44]
 MixColumns:[4B,38,65,B0,18,DC,6D,1F,39,79,6D,50,7A,2F,1B,B9]
 Round key:  [16,94,5B,1E,21,79,5B,D3,F8,58,00,9D,5C,85,9D,84]
 Add RK:     [5D,AC,3E,AE,39,A5,36,CC,C1,21,6D,CD,26,AA,86,3D]
```

```
Round 8:
 Start:      [5D,AC,3E,AE,39,A5,36,CC,C1,21,6D,CD,26,AA,86,3D]
 ByteSub:    [4C,91,B2,E4,12,06,05,4B,78,FD,3C,BD,F7,AC,44,27]
 ShiftRows:  [4C,06,3C,27,12,FD,44,E4,78,AC,B2,4B,F7,91,05,BD]
 MixColumns:[89,23,5B,A0,98,DB,50,5C,E6,BD,76,00,E5,7C,B0,F7]
 Round key:  [01,CA,04,54,20,B3,5F,87,D8,EB,5F,1A,84,6E,C2,9E]
 Add RK:     [88,E9,5F,F4,B8,68,0F,DB,3E,56,29,1A,61,12,72,69]
Round 9:
 Start:      [88,E9,5F,F4,B8,68,0F,DB,3E,56,29,1A,61,12,72,69]
 ByteSub:    [C4,1E,CF,BF,6C,45,76,B9,B2,B1,A5,A2,EF,C9,40,F9]
 ShiftRows:  [C4,45,A5,F9,6C,B1,40,BF,B2,C9,CF,B9,EF,1E,76,A2]
 MixColumns:[00,43,C0,5E,EF,6A,87,20,49,C8,2E,A2,33,EB,E0,1D]
 Round key:  [85,EF,0F,0B,A5,5C,50,8C,7D,B7,0F,96,F9,D9,CD,08]
 Add RK:     [85,AC,CF,55,4A,36,D7,AC,34,7F,21,34,CA,32,2D,15]
Round 10:
 Start:      [85,AC,CF,55,4A,36,D7,AC,34,7F,21,34,CA,32,2D,15]
 ByteSub:    [97,91,8A,FC,D6,05,0E,91,18,D2,FD,18,74,23,D8,59]
 ShiftRows:  [97,05,FD,59,D6,D2,D8,FC,18,23,8A,91,74,91,0E,18]
 Round key:  [86,52,3F,92,23,0E,6F,1E,5E,B9,60,88,A7,60,AD,80]
 Add RK:     [11,57,C2,CB,F5,DC,B7,E2,46,9A,EA,19,D3,F1,A3,98]
Output:      [11,57,C2,CB,F5,DC,B7,E2,46,9A,EA,19,D3,F1,A3,98]

              [11,57,C2,CB,F5,DC,B7,E2,46,9A,EA,19,D3,F1,A3,98]
                                                 Type: OneDimensionalArray Byte
```

Diese Ausgabe zeigt die Ergebnisse nach jedem Funktionsaufruf. Die Runde 0, auch Vorrunde genannt, besteht nur aus dem Aufruf von `AddRoundKey`. Die Runden 1-9 entsprechen den Aufrufen aus der for-Schleife (vgl. Pseudo-Code 6.2.1). Abschließend wird eine Schlußrunde (Runde 10) durchgeführt, in der auf die Operation `mixColumns` verzichtet wird (Funktionsaufrufe unterhalb der for-Schleife).

7 Betriebsmodi für Blockchiffren

In der Kryptographie arbeiten Blockchiffren mit Blöcken einer festen Länge. Da zu verschlüsselnde Nachrichten eine beliebige Länge aufweisen können wurde eine Vielzahl von Betriebsmodi zur Verschlüsselung von Nachrichten entwickelt. Im Rahmen dieser Arbeit wurden die 5 wichtigsten Betriebsmodi implementiert:

- Electronic code book
- Cyclic cipher chaining
- Output feedback
- Cipher feedback
- Counter mode

Die Implementierung wurde nach dem Dokument SP800-38A [SP800-38A] der NIST vorgenommen.

Die Funktionen wurden alle in zwei verschiedenen Datentypen implementiert, da die Verarbeitung von Zeichen anders abläuft als die von Bit-Vektoren. Der Datentyp `CryptograhicOperations` beinhaltet die Betriebsmodi für Chiffren die Bits als Eingabealphabet nutzen. Für Chiffren die auf Zeichenalphabeten arbeiten wurde der Datentyp `CryptographicOperationsChar` implementiert.

Neben der Tatsache das zwei verschiedene Datentypen verarbeitet werden ist auch die in vielen Betriebsmodi eingesetzte XOR Operation ein Grund für diese Aufteilung. XOR ist eine binäre Operation die nur auf Alphabeten der Ordnung 2 angewendet werden kann. Bei zeichenorientierten Chiffren ist die Ordnung des Alphabets größer. Daher muss in diesem Fall eine Ersatzfunktion für das XOR definiert werden. Diese Ersatzfunktion muss bei der Benutzung des Datentyps `CryptographicOperationsChar` als Parameter angeben werden.

Im Folgenden werden die Parameter der beiden Datentypen und anschließend die Funktionen für die Betriebsmodi aus diesen Datentypen erläutert.

7.1 CryptographicOperations

Die Funktionen für die verschiedenen Betriebsmodi befinden sich in dem Datentyp `CryptograhicOperations`. Dabei handelt es sich um einen parametrisierten Datentyp. D.h. damit man Funktionen aus diesem Datentyp nutzen kann, müssen auch alle Parameter dieses Datentyps angegeben werden. Der Datentyp `CryptographicOperations` besitzt folgende Parameter.

```
CryptographicOperations(n,k,f)
    n: NNI
    k: NNI
    f: (FBits,FBits,Symbol,NNI)->FBits
```

Dabei gibt `n` die Blockgröße und `k` die Schlüssellänge in Bits an. Die Verschlüsselungsfunktion die bei den Operationen verwendet werden soll wird mit `f` angegeben. Der Funktionskopf von `f` muss die gleiche Struktur wie hier angegeben besitzen. D.h. f bekommt als Parameter den Klartext, den Schlüssel, die Verschlüsselungsrichtung und den Verbose Modus übergeben. Die in Kapitel 6 vorgestellten Blockchiffre-Funktionen erfüllen diese Anforderungen.

Bei häufigeren Aufrufen einer Funktion dieses Datentyps ist es empfehlenswert den Aufruf mit Parametern in einer Variable zu speichern.

```
DMUL257 := CryptographicOperations(8,8,mul257$BitCiphers)
     CryptographicOperations(8,8,theMap(BCIPHER;mul257;2FbSNniFb;6,7))
                                                          Type: Domain
```

Die Variable `DMUL257` wird bei der Erläuterung der Funktionen dieses Datentyps in den folgenden Abschnitten wieder verwendet.

7.2 CryptographicOperationsChar

Dieser Datentyp stellt die Betriebsmodi (und weitere Funktionen) für Chiffren bereit, die auf Zeichensätzen und nicht auf Bits basieren. Dazu zählen z.B. die Chiffren aus dem Paket `ClassicCiphers`. Damit die Funktionen ordnungsgemäß eingesetzt werden können, müssen einige Besonderheiten beachtet werden. Zunächst werden die Parameter des Datentyps betrachtet.

```
CryptographicOperationsChar(n,k,a,f,g)
    n: NNI
    k: NNI
    a: Alphabet
    f: (String,String,Alphabet,Symbol,NNI)->FBits
    g: (Character,Character,Symbol)->Character
```

Die Blockgröße `n` und die Schlüssellänge `k` wird als Anzahl von Zeichen aus dem Alphabet a angegeben. Da die Funktionen mit verschiedenen Alphabeten arbeiten können, muss auch das verwendetete Alphabet `a` als Parameter übergeben werden. Durch `f` wird die Verschlüsselungsfunktion angegeben. f erwartet als Parameter den Klartext, den Schlüssel, ein Alphabet, die Chiffrierrichtung und den Verbose Modus. Die in Kapitel 5 vorgestellten klassischen Chiffren besitzen diesen Aufbau. Mit der Funktion `g` wird ein Ersatz für die Bit-Operation XOR übergeben. Bei den zeichenorientierten Chiffren ist die kleinste Einheit ein Zeichen, daher verarbeitet `g` zwei Zeichen. Im Gegensatz zur XOR Operation muss g nicht kommutativ sein, sondern nur folgende Eigenschaft erfüllen:

$$(c1, c2, c3) : Character \in Alphabet\ a : \quad g(c1, c2, encrypt) = c3 \quad \Rightarrow \quad g(c3, c2, decrypt) = c1$$

Auch hier wird empfohlen den Datentyp mit Parametern in einer Variable zu speichern.

```
a := alphabet("abcd")
                                        "abcd"
                                                                    Type: Alphabet

DVIG := CryptographicOperationsChar(4,4,a,vigenere$ClassicCiphers,gxor)
  CryptographicOperationsChar(4,4,abcd,theMap(CLCIPHER;vigenere;2SASNniS;),
                              theMap *3;gxor;1;initial)
                                                                    Type: Domain
```

Die Funktion `gxor` ist nicht Teil dieser Arbeit und muss daher durch eine eigene Implementierung ersetzt werden. Die hier verwendete Funktion gxor lässt sich durch folgende Tabelle darstellen:

	a	b	c	d
a	d	c	b	a
b	c	d	a	b
c	b	a	d	c
d	a	b	c	d

Tabelle 7.1: gxor

gxor(x,y,encrypt) wird auf das Zeichen in der x-ten Zeile und der y-ten Spalte abgebildet (z.B. gxor(b,a,encrypt) → c). Bei der Entschlüsselung kann genauso vorgegangen werden.

Die Funktion gxor und die Variable DVIG werden bei der Erläuterung der Funktionen des Datentyps `CryptographicOperationsChar` in den folgenden Abschnitten wieder verwendet.

7.3 Electronic code book (ECB)

Der einfachste Betriebsmodus ist ECB. Die Nachricht wird in Blöcke unterteilt und anschließend wird auf diese Blöcke die Chiffrierfunktion angewendet. Nachteil dieser Methode ist die Abbildung von gleichen Klartexten auf gleiche Chiffretexte.

ECB Verschlüsselung: $C_j = E(K, P_j)$

ECB Entschlüsselung: $P_j = D(K, C_j)$

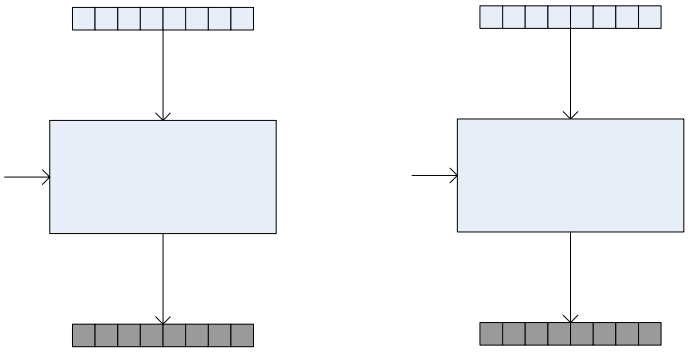

Abbildung 7.1: ECB Betriebsmodus (Verschlüsselung)

Der Funktionname für Verschlüsselungen im Electronic code book Betriebsmodus lautet `ecb`. Die Inverse Funktion zur Entschlüsselung `invecb`.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**`ecb(p:String,k:String,v:NNI)`** **`invecb(p:String,k:String,v:NNI)`**
Parameter:	**`p`** Die zu verschlüsselnde Nachricht **`k`** Der Schlüssel **`v`** Verbose Modus (0,1,2)

```
ecb("dbccbdca","acca",0)$DVIG
                                "ddacbbaa"
                                                        Type: String

invecb("ddacbbaa","acca",0)$DVIG
                                "dbccbdca"
                                                        Type: String
```

sel K

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**`ecb(p:FBits,k:FBits,v:NNI)`** **`invecb(p:FBits,k:FBits,v:NNI)`**
Parameter:	**`p`** Die zu verschlüsselnde Nachricht **`k`** Der Schlüssel **`v`** Verbose Modus (0,1,2)

```
ecb(stringToBits("k2"),stringToBits("$"),0)$DMUL257
                              10001100 01010111
                                                        Type: Fbits

invecb(stringToBits("îW"),stringToBits("$"),0)$DMUL257
                              01101011 00110010
                                                        Type: FBits
```

7.4 Cipher block chaining (CBC)

Der CBC Modus ist eine Verbesserung des ECB. Damit nicht gleiche Klartexte auf gleiche Chiffretexte abgebildet werden, wird zunächst der Klartext-Block mit dem vorherigen Chiffre-Block mittels XOR verknüpft. Für den ersten Klartext muss für diese Verknüpfung ein Initialisierungsvektor (IV) bereitgestellt werden. Damit ist jeder Chiffretext-Block abhängig von allen vorhergehenden Klartexten.

CBC Verschlüsselung: $C_1 = E(K, P_1 \oplus IV)$

$C_j = E(K, P_j \oplus C_{j-1})$

CBC Entschlüsselung: $P_1 = D(K, C_1) \oplus IV$

$P_j = D(K, C_j) \oplus C_{j-1}$

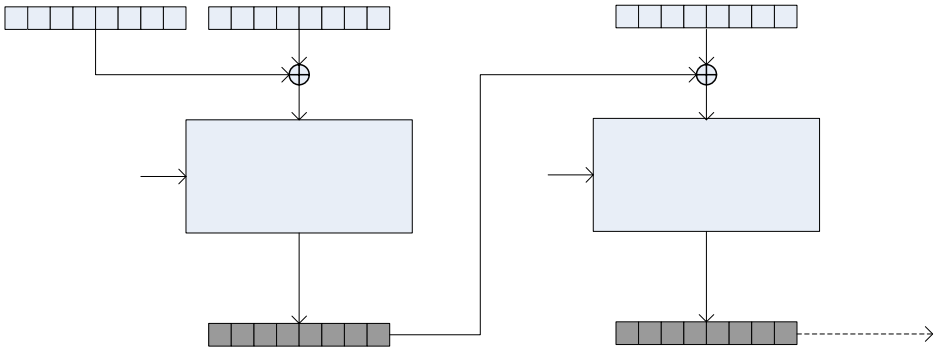

Abbildung 7.2: CBC Betriebsmodus (Verschlüsselung)

Die Funktionen für Verschlüsselungen im CBC Modus heißen `cbc` und `invcbc`.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**`cbc(p:String,k:String,iv:String,v:NNI)`** **`invcbc(p:String,k:String,iv:String,v:NNI)`**
Parameter:	**`p`** Die zu verschlüsselnde Nachricht
	`k` Der Schlüssel
	`iv` Der Initialisierungsvektor
	`v` Verbose Modus (0,1,2)

```
cbc("dbccbdca","acca","badc",0)$DVIG
                                "baaddcda"
                                                        Type: String

invcbc("baaddcda","acca","badc",0)$DVIG
                                "dbccbdca"
                                                        Type: String
```

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**cbc(p:FBits,k:FBits,iv:FBits,v:NNI)** **invcbc(p:FBits,k:FBits,iv:FBits,v:NNI)**
Parameter:	**p** Die zu verschlüsselnde Nachricht
	k Der Schlüssel
	iv Der Initialisierungsvektor
	v Verbose Modus (0,1,2)

```
cbc(stringToBits("k2"),stringToBits("$"),stringToBits("R"),0) $DMUL257
                              01011001 10001100
                                                              Type: Fbits

invcbc(stringToBits("Yî"),stringToBits("$"),stringToBits("R"),0) $DMUL257
                              01101011 00110010
                                                              Type: FBits
```

7.5 Cipher feedback (CFB)

Der CFB Modus verwandelt eine Blockchiffre in eine Stromchiffre. Hierbei wird die Verschlüsselungsfunktion nicht mehr auf den Klartext, sondern auf den Initialisierungsvektor angewendet. Anschließend werden die ersten b-Bit des verschlüsselten IV mittels XOR mit den ersten b-Bit der Nachricht verknüpft. Daraus ergibt sich der Chiffretext. Für den nächsten Block wird der IV modifiziert indem dieser um b-Bit nach links geschoben und mit dem Chiffretext des letzten Blocks aufgefüllt wird. Demnach muss bei der Anwendung des CFB Modus, neben dem IV, noch ein Wert für b definiert werden.

Durch die selbstinverse Eigenschaft von XOR kann für die Ver- und Entschlüsselung der selbe Algorithmus verwendet werden.

CFB Verschlüsselung:

$$I_1 = IV$$

$$I_j = LSB_{n-b}(I_{j-1}) \mid C_{j-1} \qquad n : \text{Blockgröße}$$

$$O_j = E(K, I_j)$$

$$C_j = P_j \oplus MSB_b(O_j)$$

CFB Entschlüsselung:

$$I_1 = IV$$

$$I_j = LSB_{n-b}(I_{j-1}) \mid C_{j-1}$$

$$O_j = E(K, I_j)$$

$$P_j = C_j \oplus MSB_b(O_j)$$

$LSB_x(Y)$: *x geringwertigen Bits von Y*

$MSB_x(Y)$: *x höchstwertigen Bits von Y*

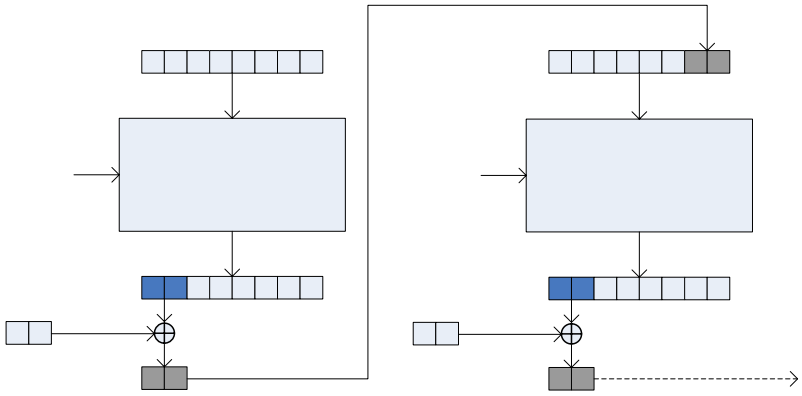

Abbildung 7.3: CFB Betriebsmodus

Die Funktionen für Verschlüsselungen im CFB Modus heißen `cfb` und `invcfb`.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**`cfb(p:String,k:String,iv:String,`** **`b:PositiveInteger,v:NNI)`** **`invcfb(p:String,k:String,iv:String,`** **`b:PositiveInteger,v:NNI)`**
Parameter:	**`p`** Die zu verschlüsselnde Nachricht **`k`** Der Schlüssel **`iv`** Der Initialisierungsvektor **`b`** Blockgröße für den CFB Modus **`v`** Verbose Modus (0,1,2)

```
cfb("dbccbdca","acca","badc",2,0)$DVIG
                                "bacbdcda"
                                                        Type: String

invcfb("bacbdcda","acca","badc",2,0)$DVIG
                                "dbccbdca"
                                                        Type: String
```

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**`cfb(p:FBits,k:FBits,iv:FBits,`** **`b:PositiveInteger,v:NNI)`** **`invcfb(p:FBits,k:FBits,iv:FBits,`** **`b:PositiveInteger,v:NNI)`**
Parameter:	**`p`** Die zu verschlüsselnde Nachricht **`k`** Der Schlüssel **`iv`** Der Initialisierungsvektor **`b`** Blockgröße für den CFB Modus **`v`** Verbose Modus (0,1,2)

```
cfb(stringToBits("k2"),stringToBits("$"),stringToBits("R"),4,0)$DMUL257
                                10011011 01000010
                                                                    Type: Fbits

invcfb(stringToBits("øB"),stringToBits("$"),stringToBits("R"),4,0)$DMUL257
                                01101011 00110010
                                                                    Type: FBits
```

7.6 Output feedback (OFB)

Der OFB Modus verwandelt ebenfalls eine Blockchiffre in eine Stromchiffre. Der IV wird verschlüsselt und anschließend mit der Nachricht per XOR verknüpft. Der verschlüsselte IV wird in der nächsten Runde wieder verschlüsselt und mit dem nächsten Block der Nachricht verarbeitet. Die Entschlüsselung funktioniert exakt genauso.

OFB Verschlüsselung:

$$I_1 = IV$$
$$I_j = O_{j-1}$$
$$O_j = E(K, I_j)$$
$$C_j = P_j \oplus O_j$$

OFB Entschlüsselung:

$$I_1 = IV$$
$$I_j = O_{j-1}$$
$$O_j = E(K, I_j)$$
$$P_j = C_j \oplus O_j$$

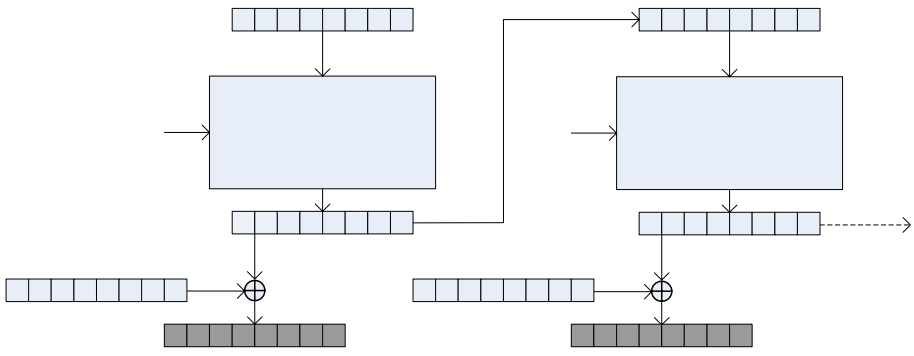

Abbildung 7.4: OFB Betriebsmodus

Die Funktionen für Verschlüsselungen im OFB Modus unter AXIOM heißen `ofb` und `invofb`.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**`ofb(p:String,k:String,iv:String,v:NNI)`** **`invofb(p:String,k:String,iv:String,v:NNI)`**

Parameter:	**p**	Die zu verschlüsselnde Nachricht
	k	Der Schlüssel
	iv	Der Initialisierungsvektor
	v	Verbose Modus (0,1,2)

```
ofb("dbccbdca","acca","badc",0)$DVIG
                                "baaddacb"
                                                        Type: String

invofb("baaddacb","acca","badc",0)$DVIG
                                "dbccbdca"
                                                        Type: String
```

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**ofb(p:FBits,k:FBits,iv:FBits,v:NNI)** **invofb(p:FBits,k:FBits,iv:FBits,v:NNI)**	
Parameter:	**p**	Die zu verschlüsselnde Nachricht
	k	Der Schlüssel
	iv	Der Initialisierungsvektor
	v	Verbose Modus (0,1,2)

```
ofb(stringToBits("k2"),stringToBits("$"),stringToBits("R"),0) $DMUL257
                             10011000 00010010
                                                        Type: Fbits

invofb(stringToBits("ÿ•"),stringToBits("$"),stringToBits("R"),0) $DMUL257
                             01101011 00110010
                                                        Type: FBits
```

7.7 Counter mode (CTR)

Im Counter Modus wird ebenfalls der IV verschlüsselt. Nach der Verschlüsselung wird das Ergebnis mit dem ersten Klartext-Block mittels XOR verknüpft. Daraus resultiert der Chiffretext. Für die nachfolgenden Blöcke wird der IV jeweils um eins inkrementiert. Der CTR hat ebenfalls den Vorteil, das er selbstinvers ist.

CTR Verschlüsselung: $I_1 = IV$

$I_j = I_{j-1} + 1$

$C_j = P_j \oplus E(K, I_j)$

CTR Entschlüsselung: $I_1 = IV$

$I_j = I_{j-1} + 1$

$P_j = C_j \oplus E(K, I_j)$

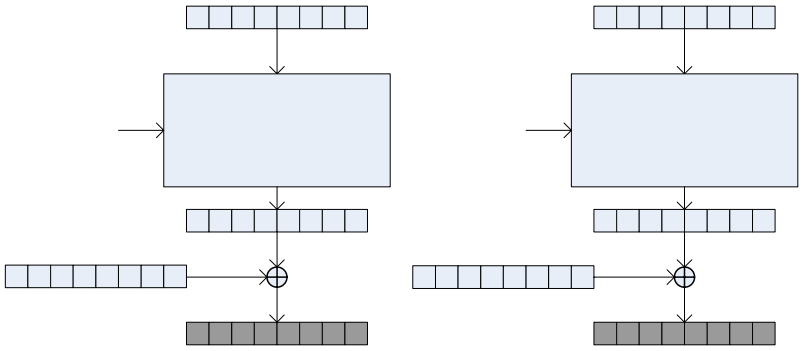

Abbildung 7.5: CTR Betriebsmodus

Die Funktionen für Verschlüsselungen im CTR Modus unter AXIOM heißen `ctr` und `invctr`.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**`ctr(p:String,k:String,iv:String,v:NNI)`** **`invctr(p:String,k:String,iv:String,v:NNI)`**	
Parameter:	**`p`**	Die zu verschlüsselnde Nachricht
	`k`	Der Schlüssel
	`iv`	Der Initialisierungsvektor
	`v`	Verbose Modus (0,1,2)

Schlüssel K

```
ctr("dbccbdca","acca","badc",0)$DVIG
                                  "baaddcaa"
                                                              Type: String

invctr("baaddcaa","acca","badc",0)$DVIG
                                  "dbccbdca"
                                                              Type: String
```

Klartext

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**ctr(p:FBits,k:FBits,iv:FBits,v:NNI)** **invctr(p:FBits,k:FBits,iv:FBits,v:NNI)**
Parameter:	**p** Die zu verschlüsselnde Nachricht
	k Der Schlüssel
	iv Der Initialisierungsvektor
	v Verbose Modus (0,1,2)

```
ctr(stringToBits("k2"),stringToBits("$"),stringToBits("R"),0)$DMUL257
                              10011000 00100101
                                                                Type: Fbits

invctr(stringToBits("ÿ%"),stringToBits("$"),stringToBits("R"),0)$DMUL257
                              01101011 00110010
                                                                Type: FBits
```

7.8 Padding bei den Betriebsmodi

Klartext-Padding

Bei einigen der Betriebsmodi (ECB und CBC) muss die Länge des Klartextes ein Vielfaches der Blockgröße sein. Da dies nicht immer der Fall ist, wird dem Klartext ein Padding angehangen, so dass diese Eigenschaft erfüllt wird. Bei den Blockchiffren wird der Klartext mit "0" und bei den zeichenorientierten Chiffren mit dem ersten Zeichen des Alphabets verlängert. Wenn ein Padding angehangen wird, erscheint während der Verschlüsselung eine Warnung. Bei OFB, CFB und CTR ist kein Padding erforderlich, da diese Modi die Eigenschaft einer Stromchiffre besitzen und somit die Länge des Klartextes beliebig sein kann.

Bei der Entschlüsselung muss darauf geachtet werden ein möglicherweise vorhandenes Padding wieder zu entfernen, da dies nicht automatisch geschieht.

Schlüssel-Padding

Um die Handhabung mit den Betriebsmodi zu erleichtern werden auch kürzere Schlüssel, als von der Blockchiffre verlangt, akzeptiert. Dieser wird nach dem selben Prinzip mit Bits bzw. Zeichen auf die erforderliche Größe erweitert.

8 Algorithmen für Blockchiffren

Neben den verschiedenen Betriebsmodi bieten die Datentypen `CryptographicOperations` und `CryptographicOperationsChar` noch weitere Funktionen, die auf Blockchiffren angewendet werden können:

- Feistel Iteration
- Abhängigkeitsmatrix
- Kaskadierte Chiffre
- Brute Force Suche

8.1 Feistel Iteration

Die Feistel Iteration ist eine Blockchiffre der Länge 2n, wobei `n` die Blockgröße einer Chiffre `E` ist die mit der Feistel Iteration angewendet wird. Bei dem Durchlauf der Iteration werden die Blöcke in zwei Hälften (`L` und `R`) geteilt und einzeln weiterverarbeitet. Dabei wird bei jeder Iteration ein anderer Rundenschlüssel verwendet. Die am Ende wieder zusammengefügten Hälften ergeben den Chiffretext.

Verschlüsselung: $L_j = R_{j-1}$

$R_j = L_{j-1} \oplus E(K_{j-1}, R_{j-1})$

Entschlüsselung: $L_{j-1} = R_j \oplus E(K_j, R_j)$

$R_{j-1} = L_j$

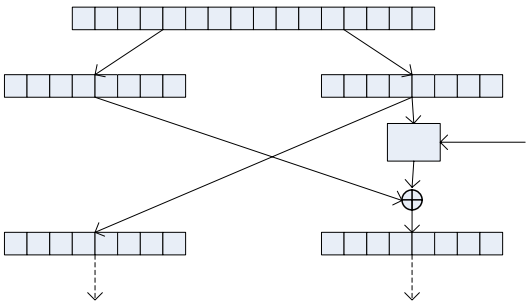

Abbildung 8.1: Feistel Iteration

Die Sicherheit der Feistel Chiffre ist abhängig von der verwendeten Funktion `E`. Im Allgemeinen wird jedoch die Verschlüsselung von `E` verbessert. Ein Vorteil der Iteration ist, daß sie selbstinvers ist. `E` muss daher nicht invertierbar sein. Bei der

Entschlüsselung muss nur die Reihenfogle der Schlüssel vertauscht werden. Die Anzahl der Iterationen ist frei wählbar (üblich sind 16 Runden).

Auf diesem Prinzip basieren viele moderne Chiffren, wie z.B. Blowfish, DES, CAST, Twofish.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**feistel(p:String,kl:List(String),d:Symbol,v:NNI)**	
Parameter:	**p**	Die zu verschlüsselnde Nachricht (doppelte Blocklänge der Ausgangschiffre)
	kl	Liste mit Schlüsseln (Länge = n → n Iterationen)
	d	Richtung (encrypt oder decrypt)
	v	Verbose Modus (0,1,2)

```
feistel("dbccbdca",["acca","badc","dcda"],encrypt,0)$DVIG
                                "cbbcddca"
                                                        Type: String

feistel("cbbcddca",["acca","badc","dcda"],decrypt,0)$DVIG
                                "dbccbdca"
                                                        Type: String
```

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**feistel(p:FBits,kl:List(FBits),d:Symbol,v:NNI)**	
Parameter:	**p**	Die zu verschlüsselnde Nachricht (doppelte Blocklänge der Ausgangschiffre)
	kl	Liste mit Schlüsseln (Länge = n → n Iterationen)
	d	Richtung (encrypt oder decrypt)
	v	Verbose Modus (0,1,2)

```
feistel(stringToBits("k2"),[stringToBits("$"),stringToBits("R"),
stringToBits("s")],encrypt,0)$DMUL257
                            11000100 01010101
                                                        Type: Fbits

feistel(stringToBits("•U"),[stringToBits("$"),stringToBits("R"),
stringToBits("s")],decrypt,0) $DMUL257
                            01101011 00110010
                                                        Type: FBits
```

8.2 Abhängigkeitsmatrix

Eine Abhängigkeitsmatrix wird oft eingesetzt um eine Block-Chiffre auf diverse Kriterien zu untersuchen. Sei F eine Chiffre mit F: $\{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n$. Dann ist die Abhängigkeitsmatrix eine (n+k × n)-Matrix. Die Werte in der Matrix an der Stelle (i,j) geben die Wahrscheinlichkeit an, dass bei einer Änderung des i-ten Inputbits das j-te Outputbit negiert wird. Als Inputbits gelten sowohl die Klartext- als auch die Schlüsselbits.

Der Datentyp `CryptographicOperations besitzt` zwei Implementierungen für die Abhängigkeitsmatrix. Eine zur genauen Berechnung (`dependenceMatrix`) und eine zur näherungsweisen Berechnung (`partialDependenceMatrix`). Diese Aufteilung ist notwendig, da die Berechnung der Abhängigkeitsmatrix sehr aufwendig ist. Für die genaue Berechnung sind $2^{(n+k)} \cdot (n+k)$ Verschlüsselungen notwendig. Bei der näherungsweisen Berechnung wird mit Zufallsvektoren gerechnet. Die Anzahl der Zufallsvektoren lässt sich über einen Parameter steuern.

8.2.1 Anwendung

Die Funktion `dependenceMatrix` erwartet keine Parameter. Die Nötigen Informationen werden dem angegebenen Datentyp entnommen, aus dem die Funktion aufgerufen wird. Im folgenden Beispiel wird der aus Kapitel 7 bekannte Datentyp `DMUL17` (mit Chiffre-Funktion `mul17`) benutzt.

```
dependenceMatrix()$DMUL17
+--------------------+
| Dependence Matrix |
+--------------------+

10% done...
20% done...
30% done...
40% done...
50% done...
60% done...
70% done...
80% done...
90% done...
100% done...

                      +0.4375  0.4375  0.4375  0.4375+
                      |                               |
                      |0.4375  0.4375  0.4375  0.4375|
                      |                               |
                      |0.4375  0.4375  0.4375  0.4375|
                      |                               |
                      |0.4375  0.4375  0.4375  0.4375|
                      |                               |
                      |0.4375  0.4375  0.4375  0.4375|
                      |                               |
                      |0.4375  0.4375  0.4375  0.4375|
                      |                               |
                      |0.4375  0.4375  0.4375  0.4375|
```

```
                        |                                   |
                       +0.4375  0.4375  0.4375  0.4375+
                                                  Type: Matrix Float
```

Da mul17 eine Chiffre mit 4-Bit Blockgröße und 4-Bit Schlüssellänge ist, ist das Ergebnis eine (8 × 4)-Matrix.

Die Funktion `partialDependenceMatrix` erwartet einen Parameter des Typs `NNI`. Dieser gibt die Anzahl der Zufallswerte an, die zur Berechnung herangezogen werden. Dabei ist zu beachten, dass dieser Wert nicht der Anzahl der durchgeführten Verschlüsselungen entspricht. Pro Zufallswert werden n+k Verschlüsselungen durchgeführt.

```
partialDependenceMatrix(200)$DMUL17
+---------------------------+
| Partial Dependence Matrix |
+---------------------------+

10% done...
20% done...
30% done...
40% done...
50% done...
60% done...
70% done...
80% done...
90% done...
100% done...

                        +0.425  0.445  0.445  0.45 +
                        |                           |
                        |0.45   0.405  0.465  0.44  |
                        |                           |
                        |0.465  0.42   0.47   0.41  |
                        |                           |
                        |0.415  0.435  0.425  0.495 |
                        |                           |
                        |0.405  0.435  0.445  0.465 |
                        |                           |
                        |0.44   0.45   0.445  0.43  |
                        |                           |
                        |0.445  0.42   0.43   0.44  |
                        |                           |
                        +0.415  0.405  0.435  0.465+
                                                  Type: Matrix Float
```

8.2.2 Aufbau der Ergebnismatrix

Um die Ergebnismatrix deuten zu können, muss die Aufteilung der Zeilen und Spalten bekannt sein. Seien $x_{n-1}x_{n-2}\ldots x_1x_0$ die Bits des Klartextes, $s_{k-1}s_{k-2}\ldots s_1s_0$ die Bits des Schlüssels und $y_{n-1}y_{n-2}\ldots y_1y_0$ die Ausgabebits von der angegebenen Chiffre. Dann sind die Zeilen und Spalten der Matrix nach folgendem Schema angeordnet:

	y_0	y_1	…	y_{n-2}	y_{n-1}

s_0	#	#	#	#	#
s_1	#	#	#	#	#
…	#	#	#	#	#
s_{n-2}	#	#	#	#	#
s_{n-1}	#	#	#	#	#
x_0	#	#	#	#	#
x_1	#	#	#	#	#
…	#	#	#	#	#
x_{n-2}	#	#	#	#	#
x_{n-1}	#	#	#	#	#

8.2.3 Deutung der Abhängigkeitsmatrix

Aus der Abhängigkeitsmatrix lassen sich einige Eigenschaften der Chiffre F ablesen. Sei (i, j) der Wert aus der i-ten Zeile und der j-ten Spalte, dann gilt:

$\exists$ i, j : (i, j) = 0	→ Die entsprechenden Bits sind unabhängig. → F ist nicht vollständig
$\forall$ i, j : (i, j) > 0	→ F ist vollständig
$\exists$ i, j : (i, j) = 1	→ Lineare Abhängigkeit zwischen den entsprechenden Bits
$\forall$ i, j : (i, j) $\in$ {0,1}	→ F ist linear
$\exists$ j, $\forall$ i : (i,j) $\in$ {0,1}	→ F ist partiell linear
$\forall$ i, j : (i, j) $\approx$ 0,5	→ F erfüllt das strikte Avalanche-Kriterium
Mittelwert aller (i, j) $\approx$ 0,5	→ F weist den Avalanche-Effekt auf
Anteil der (i, j) > 0	→ Grad der Vollständigkeit von F

(Quelle: [TUDRESD])

8.3 Kaskadierte Chiffre

Von einer kaskadierten Chiffre spricht man, wenn ein Klartext mehrfach hintereinander mit unabhängigen Schlüsseln mittels eine Funktion F verschlüsselt wird.

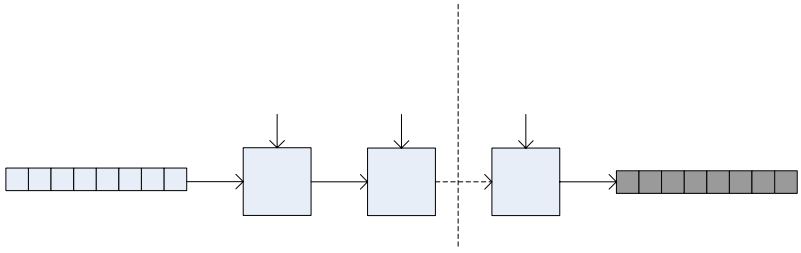

Abbildung 8.2: Kaskadierte Chiffre

Die Funktion für diesen Algorithmus heißt unter AXIOM `cascade`.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**cascade(p:String,kl:List(String),d:Symbol,v:NNI)**	
Parameter:	**p**	Die zu verschlüsselnde Nachricht
	kl	Liste mit Schlüsseln (Länge = n → n Iterationen)
	d	Richtung (encrypt oder decrypt)
	v	Verbose Modus (0,1,2)

```
cascade("dbcc",["acca","badc","dcdd"],encrypt,0)$DVIG
                                    "dbcd"
                                                              Type: String

cascade(%,["acca","badc","dcdd"],decrypt,0)$DVIG
                                    "dbcc"
                                                              Type: String
```

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**cascade(p:FBits,kl:List(FBits),d:Symbol,v:NNI)**	
Parameter:	**p**	Die zu verschlüsselnde Nachricht
	kl	Liste mit Schlüsseln (Länge = n → n Iterationen)
	d	Richtung (encrypt oder decrypt)
	v	Verbose Modus (0,1,2)

```
cascade(stringToBits("k"),[stringToBits("$"),stringToBits("R"),
stringToBits(":")],encrypt,0)$DMUL257
                                  10101110
                                                               Type: Fbits

cascade(%,[stringToBits("$"),stringToBits("R"),stringToBits(":")],
decrypt,0)$DMUL257
                                  01101011
                                                               Type: FBits
```

8.4 Brute Force Suche

Unter Brute Force Suche versteht man die Suche nach dem Schlüssel zu einem (Klartext, Chiffretext)-Paar durch erschöpfendes Ausprobieren aller möglichen Schlüssel. Der Aufwand dieser Methode steigt mit wachsender Schlüssellänge exponentiell. Daher wird diese Methode nur für Chiffren mit kurzen Schlüsseln empfohlen.

Die implementierte AXIOM-Methode heißt `bruteForce`. Ihr wird eine Liste p mit Klartexten und eine Liste c mit Chiffretexten übergeben. Dabei bilden die Elemente aus p und c jeweils (Klartext, Chiffretext)-Paare, d.h. f(p.i, k) = c.i. Es wird solange gesucht, bis ein Schlüssel k gefunden wird, der für alle Paare in Frage kommt.

Implementierung für zeichenorientierte Chiffren:

Funktionskopf:	**bruteForce(p:List(String),c:List(String),v:NNI)**
Parameter:	**p** Liste mit Klartexten
	c Liste mit Chiffretexten
	v Verbose Modus (0,1)

```
bruteForce(["cada","bbab"],["acca","dddb"],0)$DVIG
                                    "ccda"
                                                                    Type: String
```

Implementierung für bitorientierte Chiffren:

Funktionskopf:	**bruteForce(p:List(Fbits),c:List(FBits),v:NNI)**
Parameter:	**p** Liste mit Klartexten
	c Liste mit Chiffretexten
	v Verbose Modus (0,1)

```
k := randomBits(8)
                                    01001110
                                                                    Type: FBits

c1 := mul257(stringToBits("k"),k,encrypt,0)
                                    00110010
                                                                    Type: FBits

c2 := mul257(stringToBits("$"),k,encrypt,0)
                                    01011111
                                                                    Type: FBits

bruteForce([stringToBits("k"),stringToBits("$")],[c1,c2],0)$DMUL257
                                    01001110
                                                                    Type: FBits
```

9 Hilfsfunktionen

Um dem Umgang mit den Datentypen und Paketen dieser Arbeit zu erleichtern wurde ein Paket mit Hilfsfunktionen implementiert. Der Name dieses Pakets lautet `CryptographicUtilities`, dessen Funktionen in diesem Kapitel erläutert werden.

9.1 Konvertierungsfunktionen

Das Paket bietet u.a. eine Reihe von Funktionen für Konvertierungen zwischen verschiedenen Datentypen.

```
stringToBytes: String -> OneDimensionalArray(Byte)
bytesToString: OneDimensionalArray(Byte) -> String
```

Die Beiden konvertieren einen `String` in ein Array von Bytes bzw. umgekehrt. Der Datentyp `OneDimensionalArray(Byte)` wird u.a. für den Aufruf einiger Teilfunktionen des AES benötigt. Informationen zur Zeichencodierung können dem Anhang B entnommen werden.

```
bytes := stringToBytes("Kryptographie")
                  [4B,72,79,70,74,6F,67,72,61,70,68,69,65]
                                          Type: OneDimensionalArray Byte

bytesToString(bytes)
                               "Kryptographie"
                                                              Type: String
```

```
arrayToArray2d: OneDimensionalArray(Byte) ->
                                          TwoDimensionalArray(Byte)
array2dToArray: TwoDimensionalArray(Byte) ->
                                          OneDimensionalArray(Byte)
```

Diese Funktionen dienen der Konvertierung zwischen 1- und 2-dimensionalen Arrays von Bytes. Bei dem 2-dimensionalen Array werden jeweils 4 Bytes (1 Wort) pro Spalte angeordnet. Diese Anordnung wird von allen AES-Teilfunktionen erwartet, die ein 2-D Array erwarten. Die Anwendung dieser Funktion ist nur zum Testen der AES-Teilfunktionen sinnvoll.

```
arrayToArray2d(Bytes)
                              +4B  74  61  65+
                              |               |
                              |72  6F  70  00|
                              |               |
                              |79  67  68  00|
                              |               |
                              +70  72  69  00+
                                              Type: TwoDimensionalArray Byte

array2dToArray(%)
             [4B,72,79,70,74,6F,67,72,61,70,68,69,65,00,00,00]
                                              Type: OneDimensionalArray Byte
```

bitsToBytes: FBits -> OneDimenionalArray(Byte)
bytesToBits: OneDimenionalArray(Byte) -> FBits

Damit läßt sich ein Bitvektor von Typ `FBits` in ein Array von Bytes umwandeln. `bytesToBits` führt die umgekehrte Umwandlung aus. Bei der Angabe eines Bitvektors muss darauf geachtet werden, das dessen Länge ein Vielfaches von 8 ist.

```
bits := bytesToBits(bytes)
   01001011 01110010 01111001 01110000 01110100 01101111 01100111 01110010
   01100001 01110000 01101000 01101001 01100101
                                                                  Type: FBits

bitsToBytes(bits)
                  [4B,72,79,70,74,6F,67,72,61,70,68,69,65]
                                              Type: OneDimensionalArray Byte
```

bitsToInteger: FBits -> NNI
integerToBits: (NNI, NNI) -> FBits

Um einen Bitvektor als eine nicht-negative Dezimalzahl darzustellen, können diese Funktionen verwendet werden. Das zweite Argument der Funktion `integerToBits` gibt die Anzahl der Bits an, die der Ergebnisvektor besitzen soll.

```
bitsToInteger(bits)
                      597754029076652793205476716784 5
                                                  Type: PositiveInteger

integerToBits(%, 104)
   01001011 01110010 01111001 01110000 01110100 01101111 01100111 01110010
   01100001 01110000 01101000 01101001 01100101
                                                                  Type: FBits
```

stringToBits: String -> FBits
bitsToString: FBits -> String

Eine Konvertierung von `String` in einen Bitvektor ist hiermit möglich. Auch hier muss im umgekehrten Fall (`FBits` → `String`) bei der Angabe des Bitvektors darauf geachtet werden, das dessen Länge ein Vielfaches von 8 ist.

```
stringToBits("Kryptographie")
   01001011 01110010 01111001 01110000 01110100 01101111 01100111 01110010
   01100001 01110000 01101000 01101001 01100101
                                                                  Type: FBits

bitsToString(%)
                              "Kryptographie"
                                                                  Type: String
```

9.2 Zufallsvektoren

Das Paket `CryptographicUtilities` enthält auch einige Funktionen zur Erzeugung von Zufallsvektoren in verschiedenen Datentypen.

`randomString: NNI -> String`

Damit lassen sich zufällige Strings generieren. Mit dem Parameter `n` muss die Länge des Zufallsvektors gewählt werden.

```
randomString(23)
                          "xZ••DTñ7•uL}s?;*Í0¾••••"
                                                                  Type: String
```

`randomString: (NNI,Alphabet) -> String`

Diese Funktion erzeugt ebenfalls Zufallszeichenketten mit dem Unterschied, das ein Alphabet angegeben werden kann. Dieses Alphabet gibt an welche Zeichen der Ergebnisstring enthalten darf.

```
randomString(23,alphabet "a2d4f6")
                          "4d6d6da624f4af6f6d266ad"
                                                                  Type: String
```

`randomBits: NNI -> FBits`

Zufällige Bitvektoren des Typs `FBits` können mit `randomBits(n)` erzeugt werden. Der Parameter `n` gibt die Länge des Bitvektors an.

```
randomBits(42)
              00101001 11001101 00000001 00001000 00101100 11
                                                                  Type: FBits
```

`randomArrayBytes: NNI -> OneDimensionalArray(Byte)`

Die Funktion erzeugt ein Array, das mit zufälligen Byte-Werten gefüllt wird.

```
randomArrayBytes(17)
            [A0,DD,02,78,74,D3,DF,13,EC,5F,9E,DD,27,DD,CB,50,3A]
                                                Type: OneDimensionalArray Byte
```

randomArray2Bytes: (NNI,NNI) -> TwoDimensionalArray(Byte)

Ähnlich wie bei der vorherigen Funktion, wird auch hier ein Array mit Zufallswerten generiert. Jedoch ist dieses Array 2-dimensional. Über den ersten Parameter wird die Zeilenzahl und über den zweiten Parameter die Spaltenzahl definiert.

```
randomArray2Bytes(3,6)
                            +06  68  6B  07  C9  27+
                            |                     |
                            |E4  16  9D  F5  2F  DA|
                            |                     |
                            +DE  54  90  58  63  3B+
                                                Type: TwoDimensionalArray Byte
```

9.3 Formatierungsfunktionen

Neben den in 9.1 und 9.2 erwähnten Funktionen besitzt das Paket `CryptographicUtilities` noch einige Funktionen zur Formatierung von Ausgaben. Erkennbar sind diese Funktion an dem Präfix `fo`. Diese werden hauptsächlich für die Verbose-Ausgaben anderer Funktionen benötigt und sind nicht für den direkten Gebrauch geeignet. Daher werden die Funktionen nur durch einen Beispielaufruf erläutert.

fo_box: OutputForm -> Void

```
fo box("Titel"::OutputForm)

+---------+
| "Titel" |
+---------+

                                                                    Type: Void
```

fo_sep: NNI -> Void

```
fo sep(30)

------------------------------

                                                                    Type: Void
```

fo_error: OutputForm -> Void

```
fo error("Schlüssel zu kurz"::OutputForm)

+----------------------------+
| ERROR: "Schlüssel zu kurz" |
```

```
+----------------------------+

                                                                    Type: Void
```

fo_println: OutputForm -> Void

```
fo println("Informationen"::OutputForm)
"Informationen"

                                                                    Type: Void
```

fo_print: OutputForm -> Void

```
fo print("Informationen"::OutputForm)
"Informationen"
                                                                    Type: Void
```

fo_println: List OutputForm -> Void

```
fo println(["Mehrere"::OutputForm, " "::OutputForm, "Infos"::OutputForm])
"Mehrere"" ""Infos"

                                                                    Type: Void
```

fo_print: List OutputForm -> Void

```
fo print(["Mehrere"::OutputForm, " "::OutputForm, "Infos"::OutputForm])
"Mehrere"" ""Infos"
                                                                    Type: Void
```

Bei einem Aufruf aus einem kompilierten Paket heraus entfallen die Anführungszeichen der Zeichenketten. Dadurch verbessert sich das Gesamtbild der Ausgaben deutlich.

10 Korrektheit

Bei einigen komplexen Algorithmen dieser Arbeit kann es bei der Implementierung sehr leicht zu Fehlern kommen. Daher ist es wichtig die Korrektheit der implementierten Funktionen zu überprüfen.

Der komplizierteste Algorithmus dieser Arbeit ist der AES. In dem Dokument der NIST zum AES [FIPS-197] sind Testvektoren für 128 Bit Blockgröße und 128, 192 und 256 Bit Schlüssellänge angegeben. Anhand dieser konnte die Korrektheit der Implementierung validiert werden. Für die Überprüfung der Verschlüsselungen mit 192 Bit und 256 Bit Blockgröße wurden Testvektoren von der Homepage von Brian Gladman [GLADM] herangezogen.

Für die Betriebsmodi stehen ebenfalls Testvektoren zur Verfügung. Diese können dem Dokument [SP800-38A] der NIST entnommen werden. Hiermit wurde die Korrektheit vom ECB, CBC, OFB, CFB und CTR Modus überprüft. Die Betriebsmodi für zeichenbasierte Chiffren wurden manuell geprüft.

Die weiteren Funktionen wurden so weit möglich manuell auf ihre Korrektheit geprüft.

10.1 Beispielprüfung des AES

Gemäß [FIPS-197] soll der AES auf den Klartext

```
0011 2233 4455 6677 8899 AABB CCDD EEFF (128 Bit)
```

und den Schlüssel

```
0001 0203 0405 0607 0809 0A0B 0C0D 0E0F (128 Bit)
```

den Schlüsseltext

```
69C4 E0D8 6A7B 0430 D8CD B780 70B4 C55A
```

liefern.

Prüfung:

```
plain := construct([0,17,34,51,68,85,102,119,136,153,170,187,204,221,
238,255])$OneDimensionalArray(Byte)
             [00,11,22,33,44,55,66,77,88,99,AA,BB,CC,DD,EE,FF]
                                              Type: OneDimensionalArray Byte

key := construct([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
$OneDimensionalArray(Byte)
             [00,01,02,03,04,05,06,07,08,09,0A,0B,0C,0D,0E,0F]
                                              Type: OneDimensionalArray Byte

aes(plain,key,encrypt,0)
```

```
            [69,C4,E0,D8,6A,7B,04,30,D8,CD,B7,80,70,B4,C5,5A]
                                                Type: OneDimensionalArray Byte
```

10.2 Beispielprüfung des CBC

Gemäß [SP800-38A] soll der AES im CBC-Modus auf den Klartext

```
6BC1 BEE2 2E40 9F96 E93D 7E11 7393 172A
AE2D 8A57 1E03 AC9C 9EB7 6FAC 45AF 8E51
30C8 1C46 A35C E411 E5FB C119 1A0A 52EF
F69F 2445 DF4F 9B17 AD2B 417B E66C 3710 (512 Bit),
```

den Schlüssel

```
2B7E 1516 28AE D2A6 ABF7 1588 09CF 4F3C (128 bit)
```

und den IV

```
0001 0203 0405 0607 0809 0A0B 0C0D 0E0F (128 Bit)
```

den Schlüsseltext

```
7649 ABAC 8119 B246 CEE9 8E9B 12E9 197D
5086 CB9B 5072 19EE 95DB 113A 9176 78B2
73BE D6B8 E3C1 743B 7116 E69E 2222 9516
3FF1 CAA1 681F AC09 120E CA30 7586 E1A7
```

liefern.

Prüfung:

```
plain :=
construct([107,193,190,226,46,64,159,150,233,61,126,17,115,147,23,42,174,
45,138,87,30,3,172,156,158,183,111,172,69,175,142,81,48,200,28,70,163,92,22
8,17,229,251,193,25,26,10,82,239,246,159,36,69,223,79,155,23,173,43,65,123,
230,108,55,16])$OneDimensionalArray(Byte)
    [6B, C1, BE, E2, 2E, 40, 9F, 96, E9, 3D, 7E, 11, 73, 93, 17, 2A,
     AE, 2D, 8A, 57, 1E, 03, AC, 9C, 9E, B7, 6F, AC, 45, AF, 8E, 51,
     30, C8, 1C, 46, A3, 5C, E4, 11, E5, FB, C1, 19, 1A, 0A, 52, EF,
     F6, 9F, 24, 45, DF, 4F, 9B, 17, AD, 2B, 41, 7B, E6, 6C, 37, 10]
                                                Type: OneDimensionalArray Byte

key := construct([43,126,21,22,40,174,210,166,171,247,21,136,9,207,
79,60])$OneDimensionalArray(Byte)
            [2B,7E,15,16,28,AE,D2,A6,AB,F7,15,88,09,CF,4F,3C]
                                                Type: OneDimensionalArray Byte

iv := construct([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
$OneDimensionalArray(Byte)
            [00,01,02,03,04,05,06,07,08,09,0A,0B,0C,0D,0E,0F]
                                                Type: OneDimensionalArray Byte

DAES := CryptographicOperations(128,128,aes)
     CryptographicOperations(128,128,theMap(AES;aes;2FbSNniFb;25,756))
                                                                  Type: Domain




cbc(bytesToBits(plain),bytesToBits(key),bytesToBits(iv),0)$DAES
  01110110 01001001 10101011 10101100 10000001 00011001 10110010 01000110
  11001110 11101001 10001110 10011011 00010010 11101001 00011001 01111101
```

```
  01010000 10000110 11001011 10011011 01010000 01110010 00011001 11101110
  10010101 11011011 00010001 00111010 10010001 01110110 01111000 10110010
  01110011 10111110 11010110 10111000 11100011 11000001 01110100 00111011
  01110001 00010110 11100110 10011110 00100010 00100010 10010101 00010110
  00111111 11110001 11001010 10100001 01101000 00011111 10101100 00001001
  00010010 00001110 11001010 00110000 01110101 10000110 11100001 10100111
                                                                Type: FBits

bitsToBytes(%)
    [76, 49, AB, AC, 81, 19, B2, 46, CE, E9, 8E, 9B, 12, E9, 19, 7D,
     50, 86, CB, 9B, 50, 72, 19, EE, 95, DB, 11, 3A, 91, 76, 78, B2,
     73, BE, D6, B8, E3, C1, 74, 3B, 71, 16, E6, 9E, 22, 22, 95, 16,
     3F, F1, CA, A1, 68, 1F, AC, 09, 12, 0E, CA, 30, 75, 86, E1, A7]
                                          Type: OneDimensionalArray Byte
```

11 Installation

AXIOM selbst ist auf der offiziellen Seite des AXIOM Projekts (http://savannah.nongnu.org/projects/axiom/) verfügbar. AXIOM ist zur Zeit für Windows und für Linux verfügbar. Es gibt bereits kompilierte Versionen für verschiedene Linux-Distributionen. Anleitungen zur Installation sind ebenfalls auf der Homepage von AXIOM vorzufinden. Bei einigen Linux-Distribution kann AXIOM auch bequem über die Paketverwaltung (z.B. Synaptic unter Ubuntu) installiert werden.

Die Pakete und Datentypen dieser Arbeit bedürfen keiner expliziten Installation. Es müssen lediglich alle Dateien in einen beliebigen Ordner abgelegt werden. Um die Funktionen unter AXIOM nutzen zu können müssen diese zunächst geladen werden. Hierfür muss in der AXIOM-Umgebung zunächst mit Hilfe des AXIOM-Befehls `)cd` in das Verzeichnis gewechselt werden, indem sich die Dateien befinden. Anschließend kann mit `)r loadall` ein Makro gestartet werden, das alle Datentypen und Pakete dieser Arbeit lädt. Vor dem ersten Aufruf müssen jedoch einmalig alle Dateien kompiliert werden. Auch hierfür steht ein Makro bereit, das mit `)r compileall` gestartet werden kann.

Dateiliste dieser Arbeit:

`aes.spad`	Paket AESFunctions
`alph.spad`	Datentyp Alphabet
`bcipher.spad`	Paket BitCiphers
`byte.spad`	Datentyp Byte
`clcipher.spad`	Paket ClassicCiphers
`compileall.input`	Makro das alle SPAD-Dateien compiliert
`cryptop.spad`	Datentyp CryptographicOperations
`cryptopc.spad`	Datentyp CryptographicOperationsChar
`cryputil.spad`	Paket CryptographicUtilities
`fbits.spad`	Datentyp FBits
`loadall.input`	Makro das alle Datentypen und Pakete lädt

Literatur

AXIOM Richard D. Jenks, Robert S. Sutor: *Axiom – The 30 Year Horizon*. http://axiom.risc.uni-linz.ac.at/public/book2.pdf, 2005.

C850 Unicode Consortium: *Definition der Codepage 850*, ftp://ftp.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/PC/CP850.TXT

FIPS-197 Federal Information Processing Standards Publication 197: *Advanced Encryption Standard (AES)*. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

SP800-38A NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf, 2001.

GLADM Homepage von Brian Gladman, http://fp.gladman.plus.com/AES/index.htm

SAVA Website von Savannah, http://savannah.nongnu.org/

TUDRESD Technische Universität Dresden: *Vorlesung Kryptographie und Kryptoanalyse*, Folie 98, http://www.inf.tu-dresden.de/content/institutes/sya/dud/lectures/2006sommersemester/Kryptoanalyse/Kryptographie_und_Kryptoanalyse_05.pdf

Glossar

AES	Advanced Encryption Standard
CBC	Cyclic cipher chaining
CFB	Cipher feedback
CTR	Counter
ECB	Electronic code book
IV	Initialisierungsvektor
NIST	National Institute of Standards and Technology
NNI	NonNegativeInteger
OFB	Output feedback

Anhang A: Datentyp/Paket Übersicht

Dieses Kapitel bietet eine Übersicht über alle implementierten Datentypen und Pakete. Die Übersicht zeigt die Ausgabe des `)show` Befehls.

```
)show AESFunctions
AESFunctions  is a package constructor
 Abbreviation for AESFunctions is AES
 This constructor is exposed in this frame.
 Issue )edit X:/~/aes.spad to see
 algebra source code for AES

------------------------------ Operations ------------------------------
 invSbox : Byte -> Byte                sbox : Byte -> Byte
 addRoundKey : (TwoDimensionalArray Byte,TwoDimensionalArray Byte) ->
      TwoDimensionalArray Byte
 aes : (OneDimensionalArray Byte,OneDimensionalArray
      Byte,Symbol,NonNegativeInteger) -> OneDimensionalArray Byte
 aes : (String,String,Symbol,NonNegativeInteger) -> String
 aes : (FBits,FBits,Symbol,NonNegativeInteger) -> FBits
 byteSub : (TwoDimensionalArray Byte,Symbol) -> TwoDimensionalArray Byte
 finalRound : (TwoDimensionalArray Byte,TwoDimensionalArray Byte) ->
      TwoDimensionalArray Byte
 invFinalRound : (TwoDimensionalArray Byte,TwoDimensionalArray Byte) ->
      TwoDimensionalArray Byte
 invRijndael : (TwoDimensionalArray Byte,TwoDimensionalArray Byte) ->
      TwoDimensionalArray Byte
 invRound : (TwoDimensionalArray Byte,TwoDimensionalArray Byte) ->
      TwoDimensionalArray Byte
 keyExpansion : (TwoDimensionalArray Byte,NonNegativeInteger) ->
      TwoDimensionalArray Byte
 mixColumns : (TwoDimensionalArray Byte,Symbol) -> TwoDimensionalArray Byte
 rcon : PositiveInteger -> OneDimensionalArray Byte
 rijndael : (TwoDimensionalArray Byte,TwoDimensionalArray Byte) ->
      TwoDimensionalArray Byte
 rotWord : OneDimensionalArray Byte -> OneDimensionalArray Byte
 round : (TwoDimensionalArray Byte,TwoDimensionalArray Byte) ->
      TwoDimensionalArray Byte
 shiftRows : (TwoDimensionalArray Byte,Symbol) -> TwoDimensionalArray Byte
 subWord : OneDimensionalArray Byte -> OneDimensionalArray Byte

)show Alphabet
Alphabet  is a domain constructor
 Abbreviation for Alphabet is ALPH
 This constructor is exposed in this frame.
 Issue )edit X:/~/alph.spad to see
 algebra source code for ALPH

------------------------------ Operations ------------------------------
 alphabet : String -> %                alphabetic : () -> %
 alphanumeric : () -> %                coerce : % -> OutputForm
 concat : (%,%) -> %                   concat : (%,String) -> %
 digit : () -> %                       index : (Character,%) -> Integer
 lowerCase : () -> %                   member? : (String,%) -> Boolean
 upperCase : () -> %
 lookup : (NonNegativeInteger,%) -> Character
 maxIndex : % -> NonNegativeInteger
 member? : (Character,%) -> Boolean
 minIndex : % -> NonNegativeInteger
```

```
)show BitCiphers
BitCiphers  is a package constructor
 Abbreviation for BitCiphers is BCIPHER
 This constructor is exposed in this frame.
 Issue )edit X:/~/bcipher.spad to
 see algebra source code for BCIPHER

------------------------------ Operations ------------------------------
 mul17 : (NonNegativeInteger,NonNegativeInteger,Symbol,NonNegativeInteger)
      -> NonNegativeInteger
 mul17 : (FBits,FBits,Symbol,NonNegativeInteger) -> FBits
 mul257 : (NonNegativeInteger,NonNegativeInteger,Symbol,NonNegativeInteger)
      -> NonNegativeInteger
 mul257 : (FBits,FBits,Symbol,NonNegativeInteger) -> FBits

)show Byte
Byte  is a domain constructor
 Abbreviation for Byte is BYTE
 This constructor is exposed in this frame.
 Issue )edit X:/~/byte.spad to see
 algebra source code for BYTE

------------------------------ Operations ------------------------------
 ?*? : (%,%) -> %                      ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?**? : (%,PositiveInteger) -> %
 ?+? : (%,%) -> %                      ?-? : (%,%) -> %
 -? : % -> %                           ?=? : (%,%) -> Boolean
 And : (%,%) -> %                      1 : () -> %
 Or : (%,%) -> %                       Xor : (%,%) -> %
 0 : () -> %                           ?^? : (%,PositiveInteger) -> %
 coerce : Integer -> %                 coerce : % -> OutputForm
 convert : % -> Integer                hash : % -> SingleInteger
 index : PositiveInteger -> %          init : () -> %
 latex : % -> String                   lookup : % -> PositiveInteger
 one? : % -> Boolean                   random : () -> %
 recip : % -> Union(%,"failed")        sample : () -> %
 shift : (%,Integer) -> %              size : () -> NonNegativeInteger
 zero? : % -> Boolean                  ?~=? : (%,%) -> Boolean
 ?*? : (NonNegativeInteger,%) -> %
 ?**? : (%,NonNegativeInteger) -> %
 ?^? : (%,NonNegativeInteger) -> %
 characteristic : () -> NonNegativeInteger
 nextItem : % -> Union(%,"failed")
 subtractIfCan : (%,%) -> Union(%,"failed")

)show ClassicCiphers
ClassicCiphers  is a package constructor
 Abbreviation for ClassicCiphers is CLCIPHER
 This constructor is exposed in this frame.
 Issue )edit X:/~/clcipher.spad to
 see algebra source code for CLCIPHER

------------------------------ Operations ------------------------------
 caesar : (String,String,Alphabet,Symbol,NonNegativeInteger) -> String
 vigenere : (String,String,Alphabet,Symbol,NonNegativeInteger) -> String

)show CryptographicOperations
CryptographicOperations(n: NonNegativeInteger,k: NonNegativeInteger,f:
      ((FBits,FBits,Symbol,NonNegativeInteger) -> FBits))  is a package
      constructor
 Abbreviation for CryptographicOperations is CRYPTOP
 This constructor is exposed in this frame.
```

```
 Issue )edit X:/~/cryptop.spad to
 see algebra source code for CRYPTOP

------------------------------ Operations ------------------------------
 bitsAdd1 : FBits -> FBits
 cbc : (FBits,FBits,FBits,NonNegativeInteger) -> FBits
 cfb : (FBits,FBits,FBits,PositiveInteger,NonNegativeInteger) -> FBits
 ctr : (FBits,FBits,FBits,NonNegativeInteger) -> FBits
 ecb : (FBits,FBits,NonNegativeInteger) -> FBits
 feistel : (FBits,List FBits,Symbol,NonNegativeInteger) -> FBits
 invcbc : (FBits,FBits,FBits,NonNegativeInteger) -> FBits
 invcfb : (FBits,FBits,FBits,PositiveInteger,NonNegativeInteger) -> FBits
 invctr : (FBits,FBits,FBits,NonNegativeInteger) -> FBits
 invecb : (FBits,FBits,NonNegativeInteger) -> FBits
 invofb : (FBits,FBits,FBits,NonNegativeInteger) -> FBits
 ofb : (FBits,FBits,FBits,NonNegativeInteger) -> FBits
```

)show CryptographicOperationsChar

```
CryptographicOperationsChar(n: NonNegativeInteger,k: NonNegativeInteger,a:
      Alphabet,f: ((String,String,Alphabet,Symbol,NonNegativeInteger) ->
      String),g: ((Character,Character,Symbol) -> Character))  is a package
      constructor
 Abbreviation for CryptographicOperationsChar is CRYPTOPC
 This constructor is exposed in this frame.
 Issue )edit X:/~/cryptopc.spad to
 see algebra source code for CRYPTOPC

------------------------------ Operations ------------------------------
 stringAdd1 : String -> String
 cbc : (String,String,String,NonNegativeInteger) -> String
 cfb : (String,String,String,PositiveInteger,NonNegativeInteger) -> String
 ctr : (String,String,String,NonNegativeInteger) -> String
 ecb : (String,String,NonNegativeInteger) -> String
 feistel : (String,List String,Symbol,NonNegativeInteger) -> String
 invcbc : (String,String,String,NonNegativeInteger) -> String
 invcfb : (String,String,String,PositiveInteger,NonNegativeInteger) ->
      String
 invctr : (String,String,String,NonNegativeInteger) -> String
 invecb : (String,String,NonNegativeInteger) -> String
 invofb : (String,String,String,NonNegativeInteger) -> String
 ofb : (String,String,String,NonNegativeInteger) -> String
```

)show CryptographicUtilities

```
CryptographicUtilities  is a package constructor
 Abbreviation for CryptographicUtilities is CRYPUTIL
 This constructor is exposed in this frame.
 Issue )edit X:/~/cryputil.spad to
 see algebra source code for CRYPUTIL

------------------------------ Operations ------------------------------
 bitsToString : FBits -> String        fobox : OutputForm -> Void
 foerror : OutputForm -> Void          foprint : OutputForm -> Void
 foprint : List OutputForm -> Void     foprintln : OutputForm -> Void
 fosep : NonNegativeInteger -> Void    stringToBits : String -> FBits
 array2dToArray : TwoDimensionalArray Byte -> OneDimensionalArray Byte
 arrayToArray2d : OneDimensionalArray Byte -> TwoDimensionalArray Byte
 bitsToBytes : FBits -> OneDimensionalArray Byte
 bitsToInteger : FBits -> NonNegativeInteger
 bytesToBits : OneDimensionalArray Byte -> FBits
 bytesToString : OneDimensionalArray Byte -> String
 foprintln : List OutputForm -> Void
 integerToBits : (NonNegativeInteger,NonNegativeInteger) -> FBits
```

```
 randomArray2Bytes : (NonNegativeInteger,NonNegativeInteger) ->
      TwoDimensionalArray Byte
 randomArrayBytes : NonNegativeInteger -> OneDimensionalArray Byte
 randomBits : NonNegativeInteger -> FBits
 randomString : NonNegativeInteger -> String
 randomString : (NonNegativeInteger,Alphabet) -> String
 stringToBytes : String -> OneDimensionalArray Byte
```

)show FBits

```
FBits  is a domain constructor
 Abbreviation for FBits is FBITS
 This constructor is exposed in this frame.
 Issue )edit X:/~/fbits.spad to see
 algebra source code for FBITS

------------------------------- Operations --------------------------------
 ?/\? : (%,%) -> %                    ?<? : (%,%) -> Boolean
 ?<=? : (%,%) -> Boolean              ?=? : (%,%) -> Boolean
 ?>? : (%,%) -> Boolean               ?>=? : (%,%) -> Boolean
 ?\/? : (%,%) -> %                    ^? : % -> %
 ?and? : (%,%) -> %                   coerce : % -> OutputForm
 concat : (%,Boolean) -> %            concat : (Boolean,%) -> %
 concat : (%,%) -> %                  concat : List % -> %
 construct : List Boolean -> %        copy : % -> %
 delete : (%,Integer) -> %            ?.? : (%,Integer) -> Boolean
 empty : () -> %                      empty? : % -> Boolean
 entries : % -> List Boolean          eq? : (%,%) -> Boolean
 hash : % -> SingleInteger            index? : (Integer,%) -> Boolean
 indices : % -> List Integer          insert : (%,%,Integer) -> %
 latex : % -> String                  max : (%,%) -> %
 min : (%,%) -> %                     nand : (%,%) -> %
 nor : (%,%) -> %                     not? : % -> %
 ?or? : (%,%) -> %                    qelt : (%,Integer) -> Boolean
 reverse : % -> %                     sample : () -> %
 xor : (%,%) -> %                     ~? : % -> %
 ?~=? : (%,%) -> Boolean
 #? : % -> NonNegativeInteger if $ has finiteAggregate
 any? : ((Boolean -> Boolean),%) -> Boolean if $ has finiteAggregate
 convert : % -> InputForm if Boolean has KONVERT INFORM
 copyInto! : (%,%,Integer) -> % if $ has shallowlyMutable
 count : ((Boolean -> Boolean),%) -> NonNegativeInteger if $ has
      finiteAggregate
 count : (Boolean,%) -> NonNegativeInteger if $ has finiteAggregate and
      Boolean has SETCAT
 delete : (%,UniversalSegment Integer) -> %
 elt : (%,Integer,Boolean) -> Boolean
 ?.? : (%,UniversalSegment Integer) -> %
 entry? : (Boolean,%) -> Boolean if $ has finiteAggregate and Boolean has
      SETCAT
 eval : (%,List Equation Boolean) -> % if Boolean has EVALAB BOOLEAN and
      Boolean has SETCAT
 eval : (%,Equation Boolean) -> % if Boolean has EVALAB BOOLEAN and Boolean
      has SETCAT
 eval : (%,Boolean,Boolean) -> % if Boolean has EVALAB BOOLEAN and Boolean
      has SETCAT
 eval : (%,List Boolean,List Boolean) -> % if Boolean has EVALAB BOOLEAN
      and Boolean has SETCAT
 every? : ((Boolean -> Boolean),%) -> Boolean if $ has finiteAggregate
 fill! : (%,Boolean) -> % if $ has shallowlyMutable
 find : ((Boolean -> Boolean),%) -> Union(Boolean,"failed")
 first : % -> Boolean if Integer has ORDSET
 insert : (Boolean,%,Integer) -> %
```

```
less? : (%,NonNegativeInteger) -> Boolean
map : ((Boolean -> Boolean),%) -> %
map : (((Boolean,Boolean) -> Boolean),%,%) -> %
map! : ((Boolean -> Boolean),%) -> % if $ has shallowlyMutable
maxIndex : % -> Integer if Integer has ORDSET
member? : (Boolean,%) -> Boolean if $ has finiteAggregate and Boolean has
     SETCAT
members : % -> List Boolean if $ has finiteAggregate
merge : (((Boolean,Boolean) -> Boolean),%,%) -> %
merge : (%,%) -> % if Boolean has ORDSET
minIndex : % -> Integer if Integer has ORDSET
more? : (%,NonNegativeInteger) -> Boolean
new : (NonNegativeInteger,Boolean) -> %
parts : % -> List Boolean if $ has finiteAggregate
position : ((Boolean -> Boolean),%) -> Integer
position : (Boolean,%) -> Integer if Boolean has SETCAT
position : (Boolean,%,Integer) -> Integer if Boolean has SETCAT
qsetelt! : (%,Integer,Boolean) -> Boolean if $ has shallowlyMutable
reduce : (((Boolean,Boolean) -> Boolean),%,Boolean,Boolean) -> Boolean if
     $ has finiteAggregate and Boolean has SETCAT
reduce : (((Boolean,Boolean) -> Boolean),%,Boolean) -> Boolean if $ has
     finiteAggregate
reduce : (((Boolean,Boolean) -> Boolean),%) -> Boolean if $ has
     finiteAggregate
remove : (Boolean,%) -> % if $ has finiteAggregate and Boolean has SETCAT
remove : ((Boolean -> Boolean),%) -> % if $ has finiteAggregate
removeDuplicates : % -> % if $ has finiteAggregate and Boolean has SETCAT
reverse! : % -> % if $ has shallowlyMutable
select : ((Boolean -> Boolean),%) -> % if $ has finiteAggregate
setelt : (%,Integer,Boolean) -> Boolean if $ has shallowlyMutable
setelt : (%,UniversalSegment Integer,Boolean) -> Boolean if $ has
     shallowlyMutable
size? : (%,NonNegativeInteger) -> Boolean
sort : (((Boolean,Boolean) -> Boolean),%) -> %
sort : % -> % if Boolean has ORDSET
sort! : (((Boolean,Boolean) -> Boolean),%) -> % if $ has shallowlyMutable
sort! : % -> % if $ has shallowlyMutable and Boolean has ORDSET
sorted? : (((Boolean,Boolean) -> Boolean),%) -> Boolean
sorted? : % -> Boolean if Boolean has ORDSET
swap! : (%,Integer,Integer) -> Void if $ has shallowlyMutable
```

Anhang B: Zeichenkodierung

Die Kodierung der Zeichen ist abhängig vom Zeichensatz der Shell in der AXIOM läuft. Die Tabelle zeigt die Kodierung von der Eingabeaufforderung (Command-Shell) unter Windows. Dabei handelt es sich um die Codepage 850 [C850], auch als DOS-Latin1 bekannt.

0		43	+	86	V	129	ü	172	¼	215	Î
1	☺	44	,	87	W	130	é	173	¡	216	Ï
2	☻	45	-	88	X	131	â	174	«	217	┘
3	♥	46	.	89	Y	132	ä	175	»	218	┌
4	♦	47	/	90	Z	133	à	176	░	219	█
5	♣	48	0	91	[	134	å	177	▒	220	▄
6	♠	49	1	92	\	135	ç	178	▓	221	¦
7		50	2	93	]	136	ê	179	│	222	Ì
8		51	3	94	^	137	ë	180	┤	223	▀
9		52	4	95	_	138	è	181	Á	224	Ó
10		53	5	96	`	139	ï	182	Â	225	ß
11	♂	54	6	97	a	140	î	183	À	226	Ô
12	♀	55	7	98	b	141	ì	184	©	227	Ò
13		56	8	99	c	142	Ä	185	╣	228	õ
14	♫	57	9	100	d	143	Å	186	║	229	Õ
15	☼	58	:	101	e	144	É	187	╗	230	µ
16	►	59	;	102	f	145	æ	188	╝	231	þ
17	◄	60	<	103	g	146	Æ	189	¢	232	Þ
18	↕	61	=	104	h	147	ô	190	¥	233	Ú
19	‼	62	>	105	i	148	ö	191	┐	234	Û
20	¶	63	?	106	j	149	ò	192	└	235	Ù
21	§	64	@	107	k	150	û	193	┴	236	ý
22	▬	65	A	108	l	151	ù	194	┬	237	Ý
23	↨	66	B	109	m	152	ÿ	195	├	238	¯
24	↑	67	C	110	n	153	Ö	196	─	239	´
25	↓	68	D	111	o	154	Ü	197	┼	240	
26	→	69	E	112	p	155	ø	198	ã	241	±
27	←	70	F	113	q	156	£	199	Ã	242	‗
28	∟	71	G	114	r	157	Ø	200	╚	243	¾
29	↔	72	H	115	s	158	×	201	╔	244	¶
30	▲	73	I	116	t	159	ƒ	202	╩	245	§
31	▼	74	J	117	u	160	á	203	╦	246	÷
32	[Space]	75	K	118	v	161	í	204	╠	247	¸
33	!	76	L	119	w	162	ó	205	═	248	°
34	"	77	M	120	x	163	ú	206	╬	249	¨
35	#	78	N	121	y	164	ñ	207	¤	250	·
36	$	79	O	122	z	165	Ñ	208	ð	251	¹
37	%	80	P	123	{	166	ª	209	Ð	252	³
38	&	81	Q	124	|	167	º	210	Ê	253	²
39	'	82	R	125	}	168	¿	211	Ë	254	■
40	(	83	S	126	~	169	®	212	È	255	
41	)	84	T	127	⌂	170	¬	213	ı		
42	*	85	U	128	Ç	171	½	214	Í		

Hinweis: Manche Zeichen sind nicht druckbar (Steuerzeichen) und besitzen keine Alternativdarstellung.