Fachhochschule Ravensburg-Weingarten

Simulation und Analyse von Verschlüsselungsalgorithmen am Beispiel von UMTS

Diplomarbeit im Studiengang Angewandte Informatik
Fachbereich Elektrotechnik und Informatik
Fachhochschule Ravensburg-Weingarten
University of Applied Sciences

vorgelegt von

Axel Bolta
(Matrikel Nr.: 012540)

Betreuer:
Prof. Dr. W. Ertel
Dipl.-Ing. H. Schuller

Tag der Anmeldung: 01. April 2003
Tag der Abgabe: 31. Juli 2003

Vorwort

Diese Diplomarbeit entstand in Zusammenarbeit mit Nortel Networks Germany GmbH & Co. KG. Der Abschnitt „Zur Firma“ enthält einige Informationen und ein Profil der Firma.

In meiner Arbeit werde ich zunächst einen Überblick über die UMTS Systemarchitektur geben und die einzelnen Komponenten eines Netzes und deren Aufgaben beschreiben. Ich werde hierbei nicht näher auf die nachrichtentechnischen Hintergründe und Details des zellularen Mobilfunks eingehen, da sich diese Arbeit hauptsächlich mit den sicherheitsrelevanten Aspekten eines solchen Netzes und vor allem mit den verwendeten Algorithmen zur Sicherheit auf der Luftschnittstelle beschäftigen soll. Wer sich genauer über die technischen Einzelheiten im Mobilfunk informieren möchte, der sei auf [BS02] oder [WAS01] verwiesen.

Anschließend werde ich die Sicherheitsarchitektur von UMTS erläutern und die implementierten Mechansimen erklären. Die Analyse der verwendeten Blockchiffre KASUMI stellt das letzte Kapitel dar, bevor ich die Implementierung der Algorithmen in *Mathematica* und web*Mathematica* dokumentiere.

Auf eine Einführung in die Kryptographie und zugehörige Begriffserklärungen verzichte ich, da es ausreichend Literatur gibt, der solche Informationen entnommen werden können. Ich verweise deshalb auf [Ert01], [Sch96] und [Sti95] zu diesem Thema.

Alle in dieser Arbeit verwendeten Abkürzungen und Symbole sind im Anhang A erklärt.

Zur Firma

Nortel Networks Germany (NNG) ist ein führender Anbieter im Bereich Internet und Kommunikation. Nortel Networks Germany ist eine 100-prozentige Tochter von Nortel Networks. NNG wurde 1995 gegründet und beschäftigt heute ca. 700 Mitarbeiter auf 25 Standorten in Deutschland. Hauptsitz der Firma ist in Friedrichshafen, wo auch das Technologiezentrum untergebracht ist. Der Sitz der Geschäftsführung ist in Frankfurt am Main.

Nortel Networks

- Führender Anbieter von Netzwerk- und Kommunikationslösungen und -Infrastrukturen für Service Provider und Unternehmen
- Kunden in mehr als 150 Ländern, ca. 40.000 Mitarbeiter weltweit

- 100 Jahre Erfahrung
- Mehr als 75% des Backbone-Internet-Verkehrs in Nordamerika
- 50% des Datenverkehrs in Europa läuft über Netzwerke mit Nortel Networks Lösungen

Gemeinsam mit der Muttergesellschaft, dem globalen Netzausrüster Nortel Networks, baut Nortel Networks Germany das neue High-Performance Internet, das erheblich zuverlässiger und schneller als bestehende Netze ist. Zu den Kunden von Nortel Networks Germany zählen Netzwerkbetreiber, Service Provider sowie kleinere, mittlere und große Unternehmen wie z.B. O2, Deutsche Telekom, Vodafone oder die Deutsche Bahn. Der Netzausrüster setzt neue Maßstäbe für die Wirtschaftlichkeit und Qualität von Netzwerken einschließlich des Internet und eröffnet damit neue Möglichkeiten für Zusammenarbeit, Kommunikation und Handel.

Zu dem Portfolio der Nortel Networks gehören:

Optical Long Haul Networks

- Optische Netzwerke mit mehr Intelligenz und neue Services im gesamten Netz für mehr Flexibilität und bessere Kontrolle der Versorgung
- Einführung skalierbarer 40-80 Gbps-Systeme für neue Anforderungen und äußerst günstige Kosten pro Networking-Bit

Wireless Networks

- Verknüpfung mobiler Kommunikation mit intelligenten IP-Lösungen
- Migration mobiler Netze von Leitungs- auf paketbasierte Vermittlung - eine Senkung der Betriebskosten um den Faktor 10
- Der Aufbau der Infrastruktur für mobile Kommunikation der 3. Generation (UMTS) als Basis für profitable Dienste und Anwendungen

Metro und Enterprise Networks

- Voice over IP (VoIP)-Services über jedes Service-Delivery-Modell
- Optimiertes Portfolio für die Metro-Region
- Intelligent Internet: Netzwerk-Infrastruktur für Mehrwertdienste

Danksagung

Mein Dank gilt vor allem Herrn Prof. Dr. Ertel und Herrn Herbert Schuller, die mich während der vier Monate betreut haben. Ich danke auch Nortel Networks Germany für die Unterstützung und die Bereitstellung der Mittel. Außerdem danke ich all denen, die mich in irgendeiner Weise während des Verfassens dieser Diplomarbeit und während meines Studiums unterstützt haben.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Axel Bolta Friedrichshafen, den 31. Juli 2003

Inhaltsverzeichnis

Vorwort **iii**

1 Einleitung **1**

2 Grundlagen UMTS **3**

2.1 Systemarchitektur 4

2.1.1 UMTS Basisarchitektur 4

2.1.2 Einteilung in Domänen 5

2.2 Komponenten der UMTS-Architektur 6

2.2.1 Mobile Switching Centre (MSC) 6

2.2.2 Visitor Location Register (VLR) 6

2.2.3 Home Location Register (HLR) 7

2.2.4 Transcoder Rate Adaption Unit (TRAU) 7

2.2.5 Authentication Center (AuC) 7

2.2.6 Serving GPRS Support Node (SGSN) 7

2.2.7 Gateway GPRS Support Node (GGSN) 7

2.2.8 Radio Network Controller (RNC) 7

2.2.9 Node B 8

2.2.10 User Equipment (UE) 8

3 UMTS Sicherheitsarchitektur **9**

3.1 Netzzugangssicherheit 11

3.1.1 Schutz der Identität 11

3.1.2 Authentifikation und Schlüsselvereinbarung 13

3.1.3 Schlüsselverwaltung 20

3.1.4 Integrität 20

3.1.5 Verschlüsselung 25
3.2 Netzwerksicherheit 30
3.3 Benutzersicherheit 30
3.4 Anwendungssicherheit 30
3.5 Sichtbarkeit und Konfigurierbarkeit der Sicherheit 31
3.5.1 Sichtbarkeit 31
3.5.2 Konfigurierbarkeit 31

4 KASUMI 33
4.1 Übersicht 34
4.2 Die Rundenfunktion 34
4.2.1 Die Unterfunktionen *FL, FO* und *FI* 34
4.2.2 Die Rundenteilschlüssel 35
4.3 Sicherheit 35
4.3.1 Nichtlinearität der S-Boxen 37
4.3.2 Mathematische Analyse 38
4.3.3 Pragmatische Sicherheit 39
4.3.4 Seiteneffekte 39
4.4 Komplexität 39

5 Implementierung in web*Mathematica* 41
5.1 *Mathematica* und web*Mathematica* 41
5.2 KASUMI in *Mathematica* 44
5.2.1 Kasumi[in_, K_] 46
5.2.2 Kasumi[in_, K_, roundout_] 47
5.2.3 FL[in_, round_] 47
5.2.4 FO[in_, round_] 47
5.2.5 FI[in_, round_, roundFO_] 48
5.2.6 KeySchedule[K_] 48
5.2.7 S7[xlist_] 48
5.2.8 S9[xlist_] 48
5.2.9 f8[plain_, CK_] 49
5.2.10 f9[msg_, IK_] 49

5.2.11 Das Crypto Paket 49
5.2.12 Analyse-Berechnungen 50
5.3 Erstellung der web*Mathematica* Seiten 53
5.3.1 index.msp 53
5.3.2 kasumi_index.msp, f8_index.msp, f9_index.msp, des_index.msp 54
5.3.3 kasumiv1.msp, kasumiv1_1.msp 54
5.3.4 kasumiv2.msp, kasumiv2_1.msp 56
5.3.5 kasumiv3.msp, kasumiv3_1.msp 57
5.3.6 f8v1.msp, f8v1_1.msp 58
5.3.7 f8v2.msp, f8v2_1.msp 60
5.3.8 f8v3.msp, f8v3_1.msp 60
5.3.9 f9vX.msp, f9vX_Y.msp 60

6 Zusammenfassung und Ausblick **61**

A Abkürzungen, Symbole **63**
A.1 Abkürzungsverzeichnis 63
A.2 Symbole 65

B KASUMIs S-Boxen **67**
B.1 S7 67
B.2 S9 69

C Tabellen **71**

D *Mathematica* Quellcode **73**
D.1 kasumi_pack.m 73
D.2 kasumi_use.m 88
D.3 crypto.m 90
D.4 des_pack.m 97
D.5 des_use.m 106

Literatur **108**

1. Einleitung

Die Mobilfunkkommunikation ist eine der am stärksten wachsenden Informationstechnologien der letzten 10 Jahre. War ein Mobiltelefon noch Anfang der 90er nur ein Privileg für diejenigen, die es sich leisten konnten, ist es heute fast schon selbstverständlich, eines zu besitzen. Vor ca. 40 Jahren erschien die erste Generation der Mobilfunksysteme. Damals noch analog und auf nationale Ebene beschränkt (A-/B-/C-Netze in Deutschland). Mit der als GSM (*Global System for Mobile communication*) bekannten und heute noch aktuellen zweiten Generation (2G) wurden die Netze digital, womit die Qualität und die Sicherheit erhöht wurde. GSM basiert auf einem weltweiten Standard, um Netze unterschiedlicher internationaler Betreiber kompatibel zueinander zu machen und ist von den 2G Systemen auch am weitesten verbreitet. Trotzdem gibt es in manchen Ländern eigene Variationen oder Standards von Systemen, was eine weltweite Erreichbarkeit sehr schwierig macht. Ende der 90er kam dann die Idee eines wirklich einheitlichen mobilen Systems auf - die dritte Generation (3G). Anfangs hatte man noch die Vorstellung, die vielen verschiedenen privaten und öffentlichen digitalen Systeme, Satelliten, Pager, etc. zu vereinen. Auch wenn heute bei der Entwicklung von UMTS dieser Gedanke der großen Einheit noch vorhanden ist, muss man vor allem mit dem immensen Anstieg der Datenübertragungen, auch aufgrund der Ausbreitung des Internet, fertig werden. 2001 fielen immerhin 47,6% der weltweiten Telfonanschlüsse auf mobile Anschlüsse (vgl. [ITU03]). Ende 2002 wurden in den deutschen Mobilfunknetzen (D1, D2, E1, E2) 59,2 Mio. Teilnehmer erreicht. Das entspricht einer Penetrationsrate [1] von 71,7% und einem Jahreszuwachs von 2,955 Mio. Teilnehmern. Die Zahl der Mobilfunkanschlüsse überstieg damit 2002 die Zahl der Festnetzanschlüsse bereits deutlich um ungefähr 10 Mio. Anschlüsse [ITU03].

Die Vielzahl der mit UMTS neu eingeführten Dienste, wie z.B. Multimedia-Messaging (MMS), Mobiler E-Commerce (M-Commerce) oder Location Based Services (LBS) erfordern aber nicht nur mehr Bandbreite auf den Übertragungskanälen sondern stellen auch höhere Anforderungen an die Sicherheit. Gerade durch die immer größer werdende Popularität des Mobilfunks und den Nutzen in allen möglichen Bereichen des Informationsaustauschs, wird auch die Gefahr durch Angreifer und Betrüger im-

[1] Penetration = Mobilfunkteilnehmer/Einwohner

mer größer. Die übertragenen Daten werden wichtiger und vertraulicher und mit der steigenden Gewohnheit der Benutzer an das System, nimmt auch die Erwartung zu, die Integrität und die Geheimhaltung der übertragenen Daten sowohl auf der Luftschnittstelle, als auch auf jeder anderen Schnittstelle des Netzwerks auf dem Land zu gewähren.

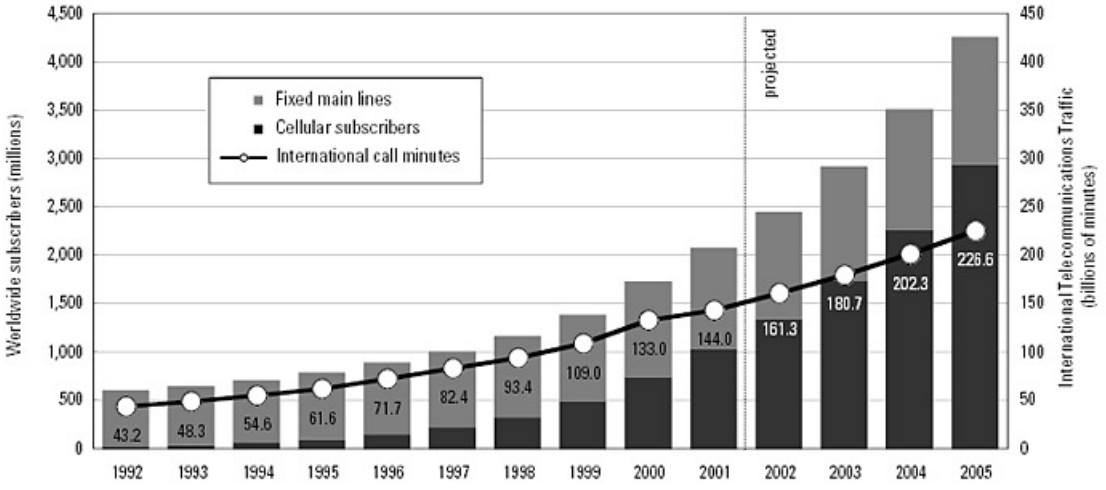

Abbildung 1.1: Wachstum des Datenverkehrs (Quelle: [Inc03])

2. Grundlagen UMTS

UMTS, das *Universal Mobile Telecommunication System*, ist die europäische Bezeichnung von Mobilfunksystemen der dritten Generation. Weltweit wird in Fachkreisen und im Bereich der ITU (*International Telecommunication Union*) die Bezeichnung IMT-2000 verwendet für *International Mobile Telecommunications*. Mit diesem System soll eine weltweit einheitliche mobile Kommunikation mit mobilen Multimedia-Diensten, mobilem Internet sowie modernen personen- und ortsbezogenen Diensten realisiert werden.

Marktforschungsinstitute gehen davon aus, dass die Zahl der mobilen Internetzugänge bereits in wenigen Jahren größer sein wird als die der Festnetzzugänge und das Übertragungsvolumen von Daten das der Sprache ebenfalls übertreffen werde. Der Bedarf der Mobilfunkkunden an immer höheren Datenraten, neuen und komfortableren Diensten für unterwegs und der Zuverlässigkeit und Sicherheit der Übertragung steigt stetig an. Nur durch die Entwicklung einer neuen Generation von Mobilfunksystemen kann dieser enorme Anspruch an Technik und Entwicklung erfüllt werden.

Ein Problem von 2G Systemen, wie GSM, ist hinsichtlich der Datenübertragung die Eigenschaft der Kanalvermittlung [WAS01]. Der entscheidende Punkt ist hierbei, dass gerade bei Datenverbindungen die Leitung zwar nicht ständig benutzt wird – z.B. beim Surfen im Internet –, dennoch aber reserviert bleibt und somit Netzressourcen verschwendet werden. Zum anderen sind 2G-Netze relativ schmalbandig, da sie ursprünglich hauptsächlich auf Sprachtelefonie ausgelegt waren mit Datenraten zwischen 5 und 15kbit/s. Das System wurde in dieser Hinsicht erweitert und verbessert und mit der Einführung des *General Packet Radio Service* (GPRS) wurde auch ein Paketvermittlungsdienst eingeführt, mit dem Datenraten bis maximal 171,2kbit/s (realistisch in der Praxis sind 30kbit/s) erreicht werden können. Der EDGE-Standard (*Enhanced Datarate for Global Evolution*) ermöglicht es GSM sogar noch höhere Datenraten zu erreichen. Da diese neuen Dienste aber alle auf der relativ alten Technik der bestehenden GSM-Infrastrukur aufsetzen, werden sie längerfristig nicht mehr hinreichend sein. UMTS soll von Anfang an schnellere Datenraten (bis 2Mbit/s im Indoor-Bereich), höhere Stabilität und eine bessere Netzeffizienz liefern, was durch eine neue Infrastruktur und neue Übertragungsverfahren erreicht werden soll.

2.1 Systemarchitektur

2.1.1 UMTS Basisarchitektur

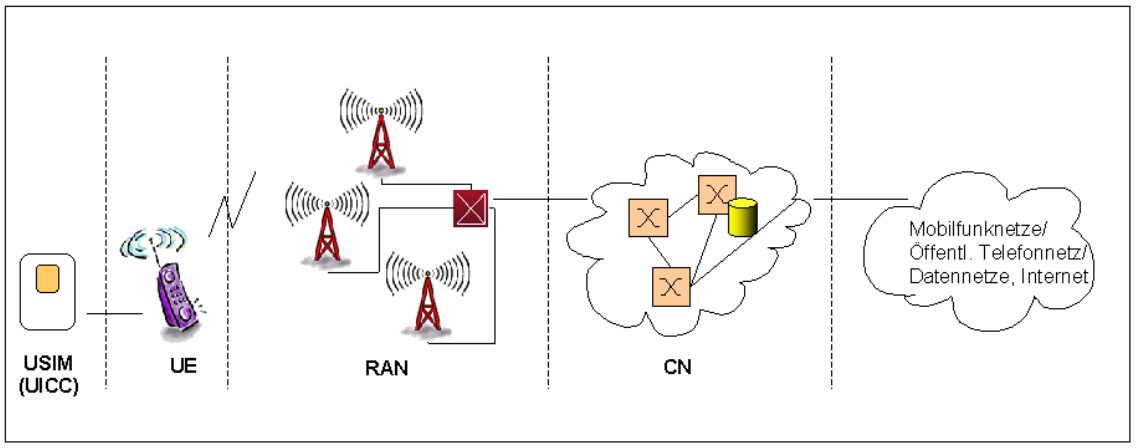

Abbildung 2.1: UMTS Basisarchitektur (Quelle: Eigene Darstellung)

Ein UMTS-Netz kann man in vier logische Blöcke mit unterschiedlichen Aufgaben unterteilen:

- USIM (*UMTS Subscriber Identity Module*)
  Die USIM ist eine Anwendung, die als individuelles und geräteunabhängiges Sicherheitsmodul im Endgerät des Teilnehmers dient. Die USIM enthält Funktionen und Informationen zur eindeutigen Identifizierung und Authentifizierung eines Teilnehmers, der 3G-Dienste benutzt. Zusätzlich enthält die USIM die IMSI (*International Mobile Subscriber Identity*) und alle benötigten Sicherheitsparameter. Sie bietet geschützten Speicherplatz für System-, oder Privatdaten des Teilnehmers, wie z.B. Telefonnummern.

  Die USIM ist in einer entnehmbaren IC-Karte implementiert, die als UICC (*Universal Integrated Circuit Card*) bezeichnet wird und eine SmartCard darstellt.

- UE (*User Equipment*)
  Die USIM wird ins ME (*Mobile Equipment*) eingesetzt, die beide zusammen das UE bilden. Es implementiert alle Protkollstapel der Funkschnittstelle, als auch die Bedienelemente für die Benutzerschnittstelle.

- RAN (*Radio Access Network*)
  Das RAN ist für die Funkübertragung im Netzwerk und die damit verbundenen Aufgaben zuständig. Es umfasst die als Node B bezeichneten Basisstationen und die Kontrollknoten, die als RNC (*Radio Network Controller*) bezeichnet werden.

- CN (*Core Network*)
  Das Kernnetz oder CN übernimmt die Transportfunktionen der Sprache und

Daten zum jeweiligen Ziel. Es enthält hierfür Vermittlungseinrichtungen – auch zu externen Netzen – und Datenbanken zur Mobilitäts- und Teilnehmerverwaltung und zur Abrechnung. Außerdem sind im CN die nötigen Einrichtungen für das Netzmanagement vorhanden.

2.1.2 Einteilung in Domänen

Abbildung 2.2 zeigt, wie ein UMTS-Netz in Funktionsblöcke, sogenannte Domänen, eingeteilt wird.

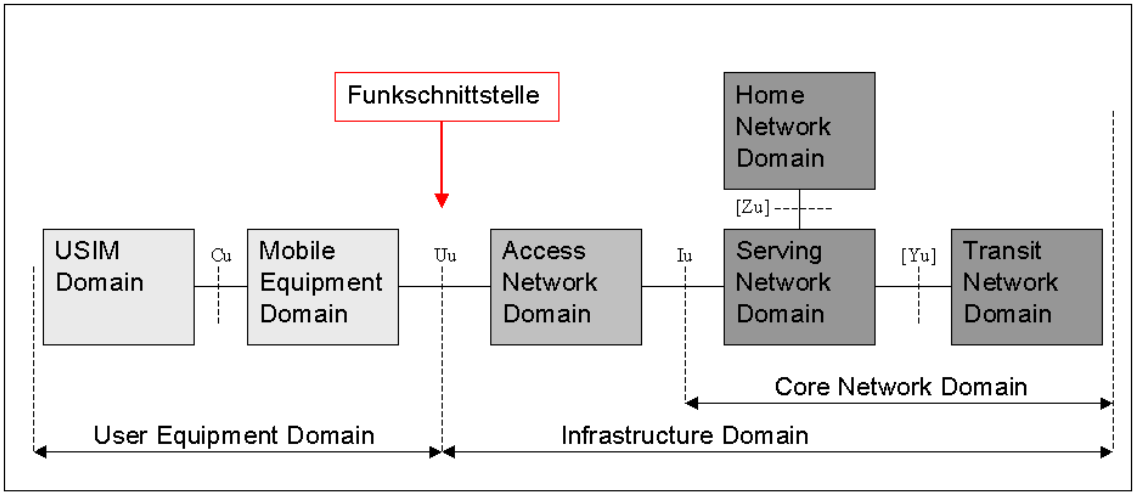

Abbildung 2.2: Einteilung des UMTS-Netzes in Domänen (Quelle: Eigene Darstellung)

- Die *User Equipment*-Domäne besteht aus den bereits beschriebenen Komponenten USIM und ME und stellt alle Funktionen zur Verfügung, die der Benutzer für den Zugang zum UMTS-Netz benötigt, wie z.B. die Verschlüsselungsfunktionen für die Übertragung der Daten über die Funkschnittstelle.
- Die *Access Network*(AN)-Domäne enthält alle Knoten und Funktionen des RAN und verbindet den Teilnehmer mit dem Transportnetz.
- Die *Core Network*-Domäne wird in drei Domänen unterteilt (vgl. [WAS01]):
  - Die *Serving Network*(SN)-Domäne enthält die Funktionen desjenigen CN, das ein Teilnehmer zu einem bestimmten Zeitpunkt für den Zugang zu UMTS-Diensten nutzt.
  - Für den Zugang zu bestimmten Diensten müssen Datenbankabfragen im Heimatnetz des Teilnehmers durchgeführt werden. Falls das *Serving Network* nicht direkt mit dem Heimatnetz (*Home Network*) verbunden ist, passieren die Daten sogenannte Transitnetze. Die Funktionen dieser Transitnetze sind in der *Transit Network*-Domäne enthalten.
  - Alle Funktionen, die im Heimatnetz des Teilnehmers realisiert werden, fallen in die *Home Network*-Domäne.

2.2 Komponenten der UMTS-Architektur

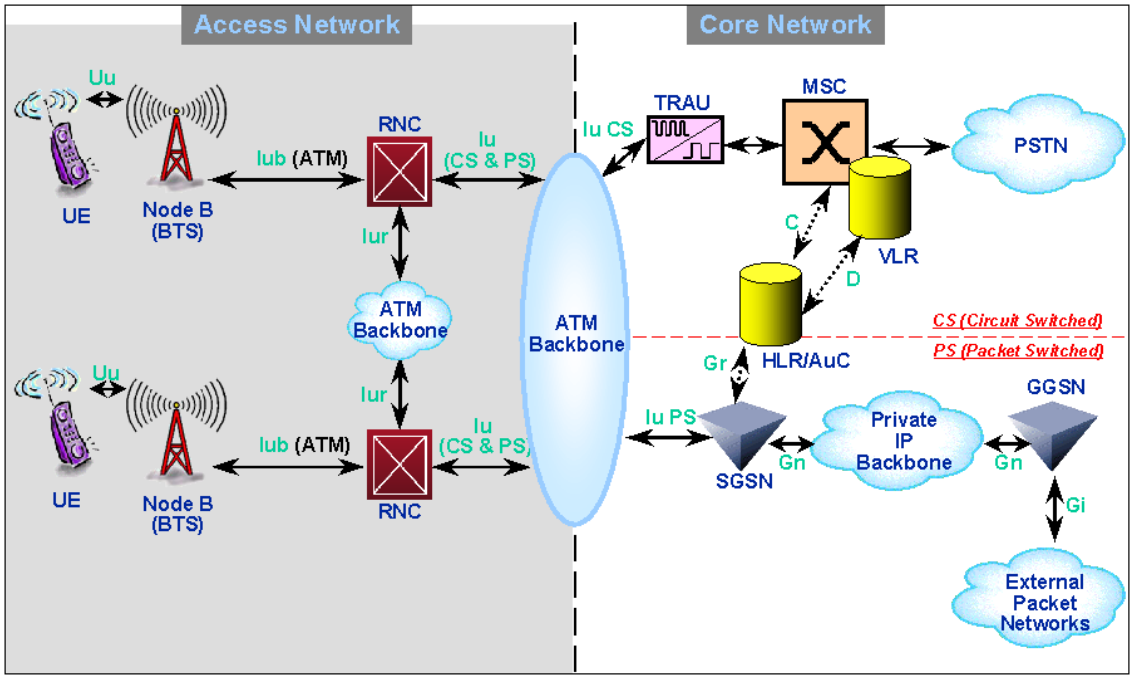

Abbildung 2.3: Die wichtigsten Komponenten im UMTS-Netz (Quelle: Nortel Networks)

UMTS unterstützt sowohl Kanalvermittlung – CS (*Circuit Switched*) –, als auch Packetvermittlung – PS (*Packet Switched*). Es werden hierfür unterschiedliche Knoten im CN verwendet.

2.2.1 Mobile Switching Centre (MSC)

Die MSC ist die zentrale Vermittlungseinheit im CN, die leitungsvermittelnd arbeitet. Zu ihren Aufgaben gehören außerdem, die Mobilität der Teilnehmer zu unterstützen und die Gespräche dem, sich bewegenden, Teilnehmer nachzuführen (*Handover*). Die MSC merkt sich für die Verbindungsherstellung den aktuellen Aufenthaltsort des Teilnehmers in einer Datenbank, im VLR. Des weiteren ist die MSC an den Mechanismen zur Authentifizierung von Teilnehmern sowie an der Verschlüsselung der Teilnehmerdaten beteiligt.

Eine Variante einer MSC ist die GMSC (*Gateway Mobile Switching Centre*), welche die Verbindung zu externen Netzen, z.B. dem ISDN oder fremden Mobilfunknetzen, liefert.

2.2.2 Visitor Location Register (VLR)

Das VLR ist wie das HLR eine Datenbank, die Teilnehmerdaten speichert. Jede MSC besitzt ihr eigenes VLR, in der lokale Kopien der Datensätze aus dem HLR über die im Zuständigkeitsgebiet der MSC befindlichen Teilnehmer gespeichert werden. Die Daten weden im VLR also nicht dauerhaft gespeichert.

2.2.3 Home Location Register (HLR)

Das HLR ist eine Datenbank, in der für jeden Teilnehmer signifikante Daten, wie z.B. Tarifmodell, Telefonnummer und die entsprechenden Berechtigungen und Schlüssel, gespeichert werden. In dem HLR ist ein Verweis auf den letzten bekannten Aufenthaltsort des Teilnehmers im Mobilfunknetz (vgl. VLR) gespeichert, so dass bei einem eingehenden Anruf die Verbindung entsprechend weitergeleitet und hergestellt werden kann.

2.2.4 Transcoder Rate Adaption Unit (TRAU)

Die TRAU übernimmt zwischen MSC und dem RAN die Sprachkompression, damit die Datenmenge an der Funkschnittstelle möglichst gering bleibt. Sie passt außerdem die Datenraten an die entsprechenden Transportverfahren (z.B. ATM) an.

2.2.5 Authentication Center (AuC)

Das AuC ist ein Teilsystem des HLR und ist zuständig für das Management sicherheitsrelevanter Daten für die Authentifizierung der Teilnehmer. Das AuC speichert die Schlüssel und Authentisierungsalgorithmen. Es erzeugt mittels dieser Schlüssel Parametersätze, die an die HLRs und damit auch an die VLRs zur Authentisierung von Teilnehmern weitergegeben werden.

2.2.6 Serving GPRS Support Node (SGSN)

Der SGSN ist ähnlich wie die MSC für ein bestimmtes geographisches Gebiet zuständig, in dem ihm ähnliche Aufgaben im Bereich der paketvermittelnden Datenübertragung zukommen wie der MSC und dem VLR im leitungsvermittelnden Bereich. Im SGSN ist ebenso die aktuelle Position des Teilnehmers gespeichert für die direkte Zustellung eines ankommenden Datenpakets. Neben Routingfunktionen übernimmt der SGSN auch die Authentifizierung und hält eine lokale Kopie der Teilnehmerinformationen gespeichert.

2.2.7 Gateway GPRS Support Node (GGSN)

Wie die GMSC für leitungsvermittelnde Netze, sorgt der GGSN für die Schnittstelle zu externen paketvermittelnden Netzen, z.B. dem Internet. Eingehende Pakete werden in der Regel durch eine integrierte Firewall gefiltert und an den entsprechenden SGSN über das GTP-Protokoll (*GPRS Tunnel Protocol*) weitergeleitet. Genaue Information über das GTP kann bei Interesse aus [3GP] entnommen werden.

2.2.8 Radio Network Controller (RNC)

Der RNC verwaltet im Funkzugangsnetz die Betriebsmittel in allen angeschlossenen Zellen (Kanalzuweisung, Handover, Leistungssteuerung). Ein großer Teil der zwischen UE und RAN verwendeten Protokolle ist im RNC implementiert. Ein RNC kommuniziert über die *Iu*-Schnittstelle jeweils mit genau einem Knoten aus dem CN – MSC oder SGSN – dem er direkt zugeordnet ist. Er hat zusätzlich aber auch die Möglichkeit, über die *Iur*-Schnittstelle direkt mit benachbarten RNCs zu kommunizieren. Zu den Aufgaben gehören, neben den bereits erwähnten, u.a. die Rufannahmesteuerung, Ver- und Entschlüsselung und die ATM-Vermittlung.

2.2.9 Node B

Der Node B stellt die unmittelbare Verbindung zur Funkschnittstelle dar und ist daher für sämtliche damit zusammenhängende Aufgaben zuständig. Kontrolliert bzw. gesteuert wird er vom RNC über die *Iub*-Schnittstelle. Ein Node B kann eine oder mehrere Zellen verwalten. Zu einem Node B gehören neben der Antennenanlage der CDMA[1]-Empfänger, der die Signale der Funkschnittstelle in einen Datenstrom umsetzt und diesen an den RNC weiterleitet. In der Gegenrichtung bereitet ein CDMA-Sender die ankommenden Daten für den Transport über die Funkschnittstelle auf und leitet sie an den Leistungsverstärker weiter.

2.2.10 User Equipment (UE)

Das direkte Gegenstück zur Node B auf der Teilnehmerseite ist das Teilnehmerendgerät UE. Es enthält die UICC mit der USIM und unterstützt einen oder mehrere Funkstandards. Seine Aufgaben bestehen unter anderem in der Verarbeitung des Funksignals – inklusive Fehlerkorrektur, Modulation, Anpassung der Sendeleistung etc. –, der Signalisierung zum Verbindungsauf- und Abbau, der Ver- und Entschlüsselung, dem Mobilitätsmanagement in Zusammenarbeit mit dem CN und der gegenseitigen Authentifizierung mit dem CN. Ein UE kann eindeutig auf der Welt anhand seiner, im Endgerät gespeicherten, IMEI (*International Mobile Equipment Identity*) bestimmt werden.

Zusätzlich zu diesen rein mobilfunkspezifischen Aufgaben wird von einem UE auch erwartet, sich der Integration in die Multimedia-Welt anzupassen. Deshalb müssen beispielsweise eine Kamera, Audio- und Videocodecs implementiert sein, es muss eine größere und farbige Anzeige haben und die Leistungsfähigkeit bzw. Kapazität der CPU und des Speichers müssen entsprechend erhöht werden.

[1] CDMA (*Code Division Multiple Access*) ist das bei UMTS verwendete Vielfachzugriffsverfahren auf der Funkschnittstelle [BS02]

3. UMTS Sicherheitsarchitektur

Die Sicherheit im UMTS Netzwerk basiert auf der Sicherheit der GSM Netze. Alle Mechanismen, die sich in GSM als positiv und unverzichtbar erwiesen haben, sind in UMTS weiterhin, teilweise unverändert, verwendet. Sämtliche Schwachstellen des Netzes sollten allerdings durch neue Mechanismen eliminiert werden. Ebenso sollten die neuen Mechanismen erweiterbar sein und der Zeit und dem Fortschritt der Technologie entsprechend auch später angepasst werden können.

Die Sicherheitsarchitektur ergibt sich aus der Kombination der Systemarchitektur zusammen mit den implementierten Sicherheitsmechanismen und teilt sich in verschiedene Schichten, in denen die Sicherheit eine Rolle spielt und solche Mechanismen angewandt werden sollten. Dieses Kapitel beschreibt diese Schichten und deren Sicherheitseigenschaften in UMTS. Besonderes Augenmerk verdient hier der Netzzugang (I), der am stärksten potenziellen Angriffen ausgesetzt ist.

Es sind insgesamt fünf Gruppen von Sicherheitsaspekten definiert, die in Abbildung 3.1 aufgezeigt sind:

- (I) **Netzzugangssicherheit**: Hierzu zählen die Sicherheitsmerkmale, die den sicheren Zugang zum UMTS-Netz gewährleisten, insbesondere der Abhör- und Angriffsschutz auf der Luftschnittstelle.
- (II) **Netzwerksicherheit**: In diesen Bereich fällt besonders die Sicherheit auf den Signalisierungsstrecken aber auch aller anderen Daten, die zwischen den Knoten im drahtgebundenen Netzwerk ausgetauscht werden.
- (III) **Benutzersicherheit**: Unter diesem Begriff versteht man den sicheren Zugriff auf mobile Endgeräte.
- (IV) **Anwendungssicherheit**: Dies sind die Sicherheitsmerkmale, die es Anwendungen im Netz des Dienst-Anbieters (*Provider*) und auf Benutzerseite auf der USIM erlauben, Nachrichten sicher untereinander auszutauschen.
- (V) **Sichtbarkeit und Konfigurierbarkeit der Sicherheit**: Diese Merkmale ermöglichen es dem Anwender, sich zu informieren ob und welche Sicherheitsmechanismen aktiviert sind. Desweiteren kann die Benutzbarkeit und

Verfügbarkeit verschiedener Dienste in Abhängigkeit der entsprechenden Sicherheitsmechanismen abgefragt werden.

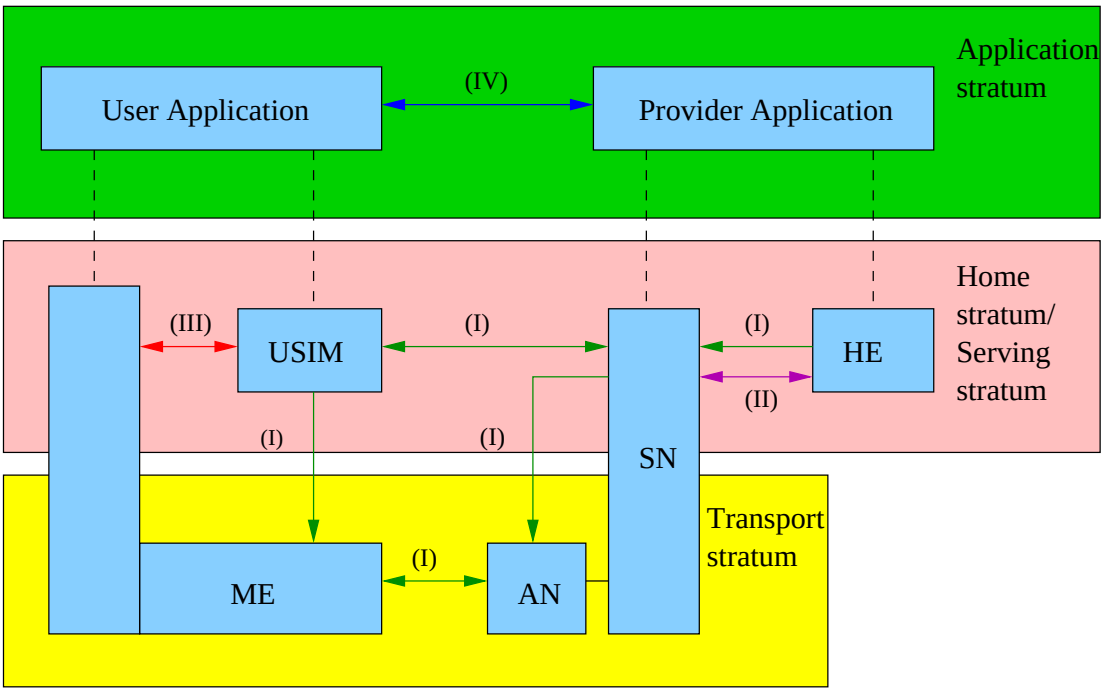

Abbildung 3.1: UMTS Sicherheitsarchitektur (Quelle: [3GP02b])

Abbildung 3.2 gibt einen Überblick über die Registrierungs- und Verbindungsprinzipien bei UMTS mit einer CS-Service-Domain und einer PS-Service-Domain. Teilnehmeridentifikation, Authentifizierung und Schlüsselwahl werden in jeder Service-Domain unabhängig durchgeführt.

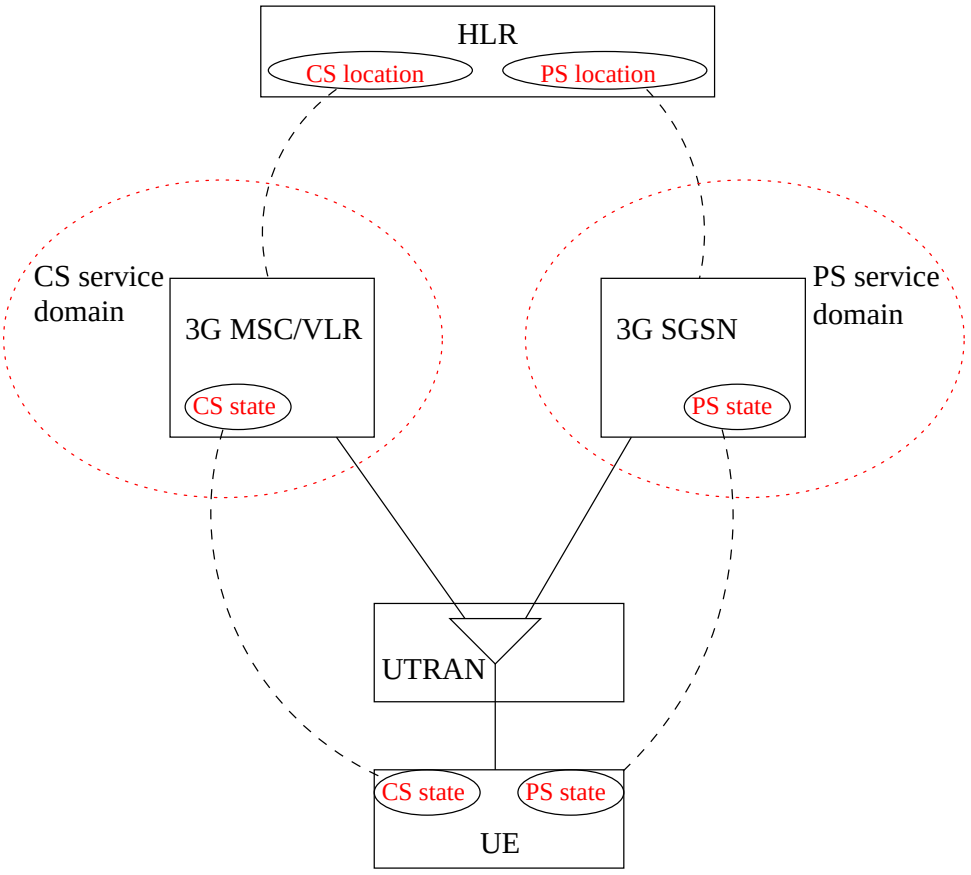

Abbildung 3.2: Überblick der UE Registrierungs- und Verbindungsprinzipien mit den unterschiedlichen CN-Architekturen (Quelle: [3GP02b])

3.1 Netzzugangssicherheit

3.1.1 Schutz der Identität

Unter dem Schutz der Identität eines Mobilfunkteilnehmers versteht man, dass er sich frei im Einzugsgebiet des Netzes bewegen und telefonieren kann, bzw. alle Dienste nutzen kann, ohne dass sich unauthorisierte Personen ein Profil über seine Aufenthaltsorte und Bewegungen erstellen können. Über dies ist es vor allem wichtig, dass seine Identität nicht auf der Luftschnittstelle ermittelt werden kann und eine Zuordung von Daten und Personen durch Angriffe auf der Luftschnittstelle nicht realisierbar ist. Im Zusammenhang mit dem Schutz der Identität eines Teilnehmers sind die folgenden Merkmale vorhanden:

- **Geheimhaltung der Identität:** Die Eigenschaft, dass die ständige Identitätskennung (IMSI) eines Teilnehmers, der einen Dienst in Anspruch nimmt, nicht auf der Luftschnittstelle abgehört werden kann.
- **Geheimhaltung des Aufenthaltsortes:** Die Eigenschaft, dass der Aufenthalt oder die Ankunft eines Teilnehmers in einem bestimmten Gebiet nicht durch Abhören auf der Luftschnittstelle festgestellt werden kann. Hierdurch wird verhindert, dass Bewegungsprofile über einen Teilnehmer erstellt werden können.

- **Geheimhaltung der Dienste:** Die Eigenschaft, dass ein Angreifer nicht durch Abhören der Luftschnittstelle erkennen kann, welche Dienste einem Teilnehmer zur Verfügung stehen, bzw. welche Dienste dieser gerade in Anspruch nimmt.

Um diese Ziele zu erreichen, wird der Teilnehmer durch das besuchte Netz über eine temporäre Identitätskennung, die sog. TMSI (*Temporary Mobile Subscriber Identity*) identifiziert. Die eigentliche Identität des Teilnehmers, die IMSI, wird somit verschleiert. Zusätzlich werden sämtliche Signalisierungs- und Nutzdaten, die die Identität des Teilnehmers verraten könnten, auf der Luftschnittstelle mit kryptographischen Algorithmen verschlüsselt.

Mit den temporären Identitäten wird die wahre Identität eines Teilnehmers verschleiert und durch eine temporär und lokal begrenzt gültige Identität ersetzt. Eine TMSI ist nur innerhalb einer geographischen Zone, einer sog. *Location Area* (LA), in welcher der Teilnehmer registriert ist, gültig. Außerhalb der LA wird zur Vermeidung von Zweideutigkeiten zusätzlich der entspechende *Location Area Identifier* (LAI) mitgeführt. Die Identität setzt sich dann aus den Parametern TMSI/LAI zusammen. Die Zuordnung zwischen permanenter und temporärer Identität wird vom VLR bzw. SGSN des bedienenden Netzes (SN – *Serving Network*) verwaltet, in dem der Teilnehmer gerade registriert ist.

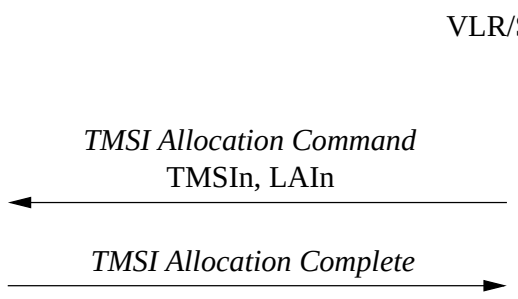

Abbildung 3.3: Zuweisung der TMSI/LAI

Die Prozedur in Abbildung 3.3 zeigt wie dem Teilnehmer ein neues TMSI/LAI Paar durch das VLR bzw. SGSN zugeordnet wird. Diese Prozedur wird erst durchgeführt, nachdem die Verschlüsselung initialisiert wurde. Von der Netzseite wird die Zuordung angestoßen, indem das VLR eine neue temporäre Identität ($TMSI_n$) erstellt und wenn nötig, zusammen mit der neuen Zonenkennung LAI_n an den Teilnehmer schickt. Das VLR speichert die neuen Parameter zusammen mit der permanenten Identität des Teilnhemers in seiner Datenbank. Wenn der Teilnehmer die $TMSI_n$ empfängt, speichert er diese und löscht sofort seine Zuordungen mit allen TMSI's, die er früher zugewiesen bekommen hat. Er antwortet mit einer Bestätigungsnachricht, um den Vorgang abzuschließen, worauf das VLR die Zuordung mit der alten $TMSI_o$ und IMSI – wenn es denn eine gab – aus seiner Datenbank löschen kann.

Kann ein Teilnehmer nicht über eine TMSI vom bedienenden Netz identifiziert werden, z.B. bei der ersten Registrierung eines Teilnehmers in einem (fremden) Netz, ist

es erforderlich, dass er sich mit seiner permanenten Identität IMSI beim Netzwerk meldet. Nach [3GP02b] wird die IMSI bei dieser Prozedur im Klartext übertragen, was eine erhebliche Lücke in der Geheimhaltung der Identität darstellt! Das Risiko wird hier jedoch als gering eingestuft, da dieses Szenario äußerst selten auftritt. Eine in früheren Spezifikationen aufgeführte, sicherere Prozedur wurde wieder verworfen, da der hierdurch erzielte Nutzen zu gering im Vergleich zur stark erhöhten Komplexität war.

3.1.2 Authentifikation und Schlüsselvereinbarung

In UMTS sind im Unterschied zu GSM zwei Arten der Authentifikation implementiert:

- **Teilnehmerauthentifikation:** Der Teilnehmer wird wie bei GSM aufgrund seiner Identität durch das Netzwerk authentifiziert.
- **Netzwerkauthentifikation:** Bei dieser sehr wichtigen Erweiterung des UMTS gegenüber GSM muss sich das Netzwerk auch gegnüber dem Teilnehmer identifizieren. Bei GSM hat dieser Mangel zu Angriffen wie dem IMSI-Catching geführt. Hierbei täuscht der Angreifer dem Teilnehmerendgerät eine Basisstation vor und fängt so die IMSI des Teilnehmers ab (siehe [Fox97]).

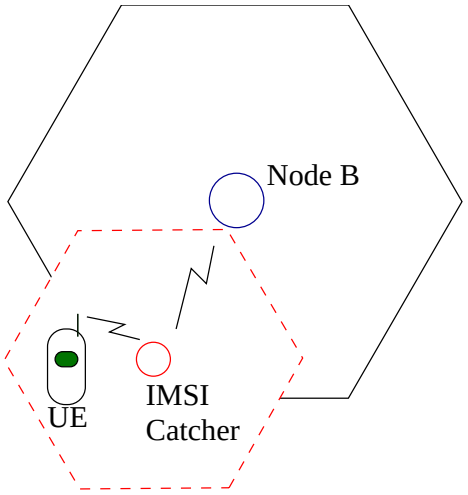

Abbildung 3.4: IMSI Catcher (Quelle: Eigene Darstellung)

Ein Angriff mit einem IMSI-Catcher[1] ist ein sog. Man-in-the-Middle-Angriff (s. [Ert01]) wie er z.B. aus der Public-Key-Kryptographie bekannt ist. Die Mobilstation kann aufgrund der fehlenden Authentifikation des Netzes gegenüber dem Teilnehmer nicht erkennen, ob es sich tatsächlich um eine vertrauenswürdige Gegenstelle eines Netzbetreibers handelt. Der IMSI-Catcher verhält sich dem Teilnehmer gegenüber wie eine Basisstation und der eigentlichen Basisstation gegenüber wie eine Mobilstation. Der Angreifer leitet die Daten in jede Richtung weiter und zeichnet

[1] Preis: 200.000 - 300.000 EUR

dabei die wahre Identität (IMSI) des bzw. der Teilnehmer auf, um den zu belauschenden Teilnehmer später aus der Menge der Verbindungen herauszufiltern. Als vorgetäuschte, „maskierte“ Basisstation hat der Angreifer die Möglichkeit, die Mobilstation anzuweisen, Verschlüsselung zu deaktivieren und die Daten unverschlüsselt zu übertragen[2]. Somit kann der Angreifer Gespräche direkt aufzeichnen.

In UMTS wurden Erweiterungen vorgenommen, um diese Schwachstelle zu beheben. Bei jedem Verbindungsaufbau zwischen Netzwerk und Teilnehmer wird in beide Richtungen eine Authentifikation vorgenommen. Das implementierte Verfahren basiert auf einem „Challenge-and-Response“ Protokoll (s. [Ert01]) in Kombination mit einem Sequenznummern basierenden „One-Pass“ Protokoll (s. [ISO99]) zur Netzwerkauthentifikation, welches vom Standard ISO/IEC 9798-4 [ISO99] abgeleitet ist.[3]

Zwei Mechanismen zur Authentifikation wurden eingeführt: Der eigentliche Authentifikationsmechanismus, der mit einem Authentifikationsvektor arbeitet, welcher vom Heimatnetz (HE – *Home Environment*) des Teilnehmers dem bedienenden Netzwerk zur Verfügung gestellt wird und Teilnehmer und Netzwerk gegenseitig authentifiziert. Ein anderer Authentifikationsmechanismus ist für die lokale Authentifikation zuständig – also für die Authentizität der anschließend übertragenen Informationen zwischen UE und SN – und benutzt den Integritätsschlüssel, der zuvor zwischen Teilnehmer und bedienendem Netz in einer Prozedur ermittelt wurde.

Hält sich ein Teilnehmer in einem anderen als seinem Heimatnetz auf, so wird die Authentifikation und Schlüsselvereinbarung (AKA - *Authentication and Key Agreement*) mit dem VLR/SGSN des entspechenden, fremden Netzes durchgeführt. Das fremde VLR/SGSN tritt dann mit dem HE/AuC, dem AuC des Heimatnetzes des Teilnehmers, wie in Abbildung 3.5 dargestellt in Verbindung.

[2]Wurde in den GSM-Standard aufgenommen mit Rücksicht auf die Länder, in denen Kryptographie verboten ist. Z.B. in Frankreich bis 1999.

[3]Die Gründe, warum man sich bei UMTS für ein „One-Pass“ Protokoll entschieden hat und nicht für ein sichereres „Two-Pass“ Protokoll, sind hauptsächlich auf die geringere Komplexität und damit höhere Geschwindigkeit erster zurückzuführen, als auch auf die bessere Interworking-Funktionalität mit GSM-Netzwerken.

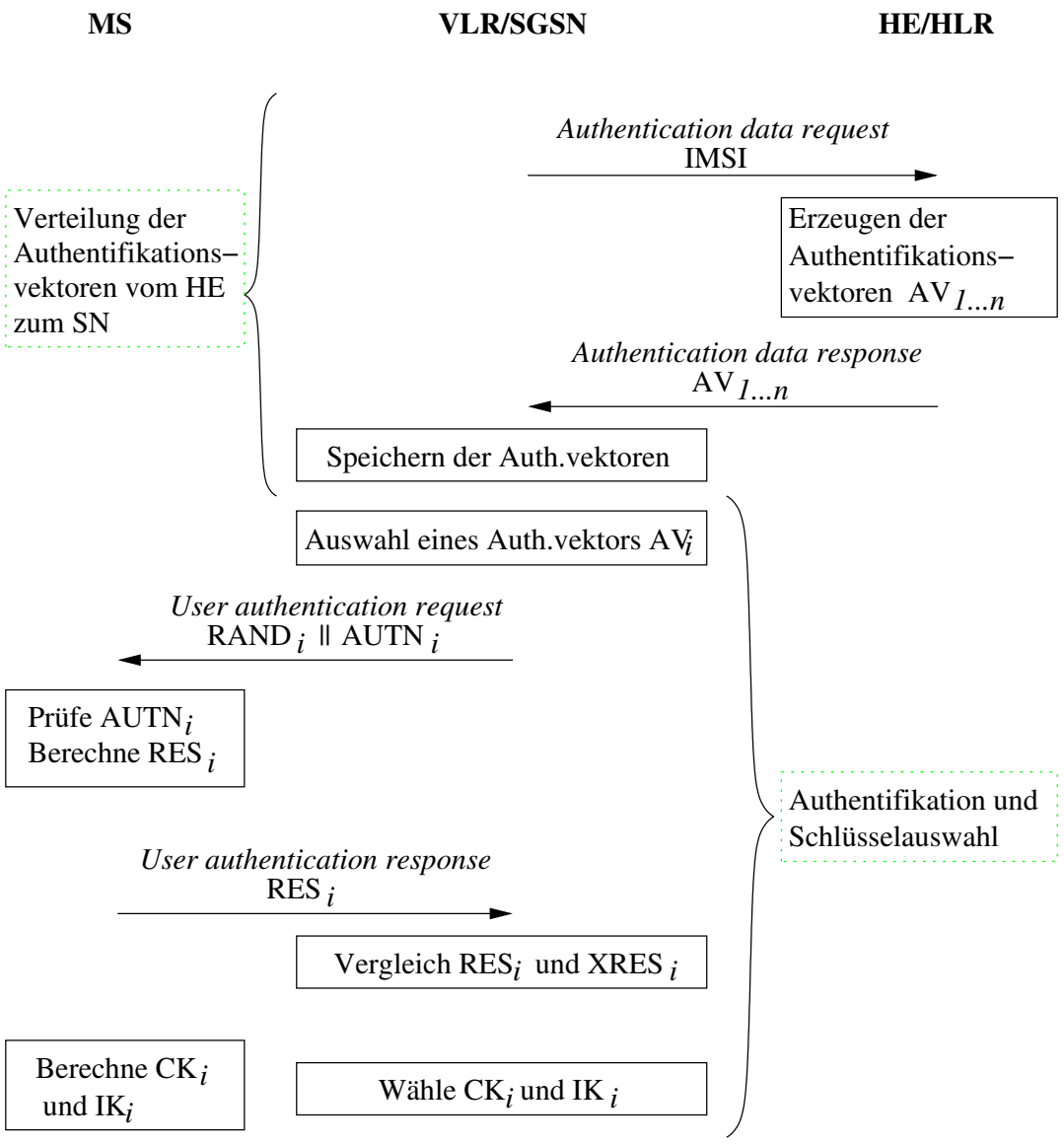

Abbildung 3.5: Authentifikation und Schlüssel-Vereinbarung (Quelle: Eigene Darstellung)

Auf eine Anfrage (*Authentication data request*, die die IMSI enthält) durch das VLR/SGSN schickt das HE/AuC ein nach der Sequenznummer sortiertes Array mit n Authentifikationsvektoren (siehe Abbildung 3.6) zurück. Hierbei ist zu beachten, dass diese kritischen Daten nicht über die Funkschnittstelle übertragen werden, sondern im verdrahteten CN. Damit die gesamte Authentifiaktions- und Schlüsselvereinbarungsprozedur als sicher gelten kann, muss hier angenommen werden, dass die Leitungen zwischen den einzelnen Knoten im CN als „angemessen sicher“ betrachtet werden können (s. Abschnitt 3.2). Außerdem muss das VLR/SGSN vom HE/AuC als vertrauenswürdig im Umgang mit geheimen Authentifikationsinformationen betrachtet werden können.

Ein Authentifikationsvektor (auch Quintett genannt) kann einmal[4] für die Authentifikation und Schlüsselvereinbarung zwischen VLR/SGSN und USIM verwendet werden und besteht aus:

- einer Zufallszahl RAND (128 Bit),
- einer vom Teilnehmer erwarteten Antwort XRES,
- einem Chiffrierschlüssel CK (128 Bit),
- einem Integritätsschlüssel IK (128 Bit),
- und einem Authentifikationstoken AUTN.

Das VLR/SGSN wählt den nächsten Authentifikationsvektor AV_i aus und sendet daraus die Parameter $RAND_i$ und $AUTN_i$ an den Teilnehmer. Auf der USIM wird das $AUTN_i$ und die Sequenznummer SQN[5] geprüft; bei Gültigkeit wird eine Antwort RES_i (32-128 Bit) generiert und zurückgeschickt. Die USIM berechnet zusätzlich noch die beiden Schlüssel CK und IK (siehe Abbildung 3.7). Wurde RES_i, nachdem es mit $XRES_i$ vom VLR/SGSN verglichen worden ist, akzeptiert, dann wird die Authentifikation und Schlüsselvereinbarung als gültig betrachtet und die vereinbarten Schlüssel CK und IK werden von USIM und VLR/SGSN jeweils an die Komponenten weitergegeben, die die Verschlüsselungs- und Integritätsfunktionen durchführen. Auf Netzwerkseite ist dies der bedienende RNC.[6]

Abbildung 3.6 zeigt wie die Authentifikationsvektoren erzeugt werden. Die jeweils vom HE/AuC generierte Sequenznummer SQN (48 Bit) und die Zufallszahl RAND, zusammen mit dem geheimen Schlüssel K (128 Bit), der nur in der USIM des Teilnehmers sowie im HE/AuC verfügbar ist und dem Feld AMF (*Authentication Management Field*) liefern die Eingabeparameter für die Algorithmen *f1* bis *f5*. Es werden folgende Werte berechnet:

- Ein *Message Authentication Code* MAC (siehe [Ert01]) zur Prüfung der Integrität des Authentifikationsvektors (64 Bit).
- Eine erwartete Authentifikations-Antwort des Teilnehmers XRES.
- Ein Chiffrierschlüssel CK zur Verschlüsselung der Datenströme.
- Ein Integritätsschlüssel IK zur Sicherstellung der Datenintegrität.
- Ein Schlüssel AK (48 Bit) zur Sicherstellung der Anonymität des Teilnehmers. AK wird hier benutzt, um die Sequenznummer zu verbergen, da diese bei einem passiven Angriff über einen längeren Zeitraum Aufschluss über die Identität und den Aufenthaltsort eines Teilnehmers geben könnte. Dies hängt davon ab, wie die SQN erzeugt wird (siehe [3GP02b], Anhang C). Wird ein Verbergen der SQN nicht benötigt gilt $f5 \equiv 0$, $(AK = 0)$.

[4] Versucht ein VLR/SGSN ein Quintett wiederzuverwenden, lehnt dies das MS ab.

[5] Sequenznummern sind zeitvariante Parameter. Sie begrenzen die Lebensdauer eines Authentifikationsvektors und schützen so vor Wiederholungsangriffen. Siehe hierzu [ISO97], Anhang B.

[6] Nach den letzten UMTS-Spezifikationen, werden auch die Schlüssel CK und IK unverschlüsselt im CN und auf der Iu Schnittstelle übertragen. Vgl. [AFKL00].

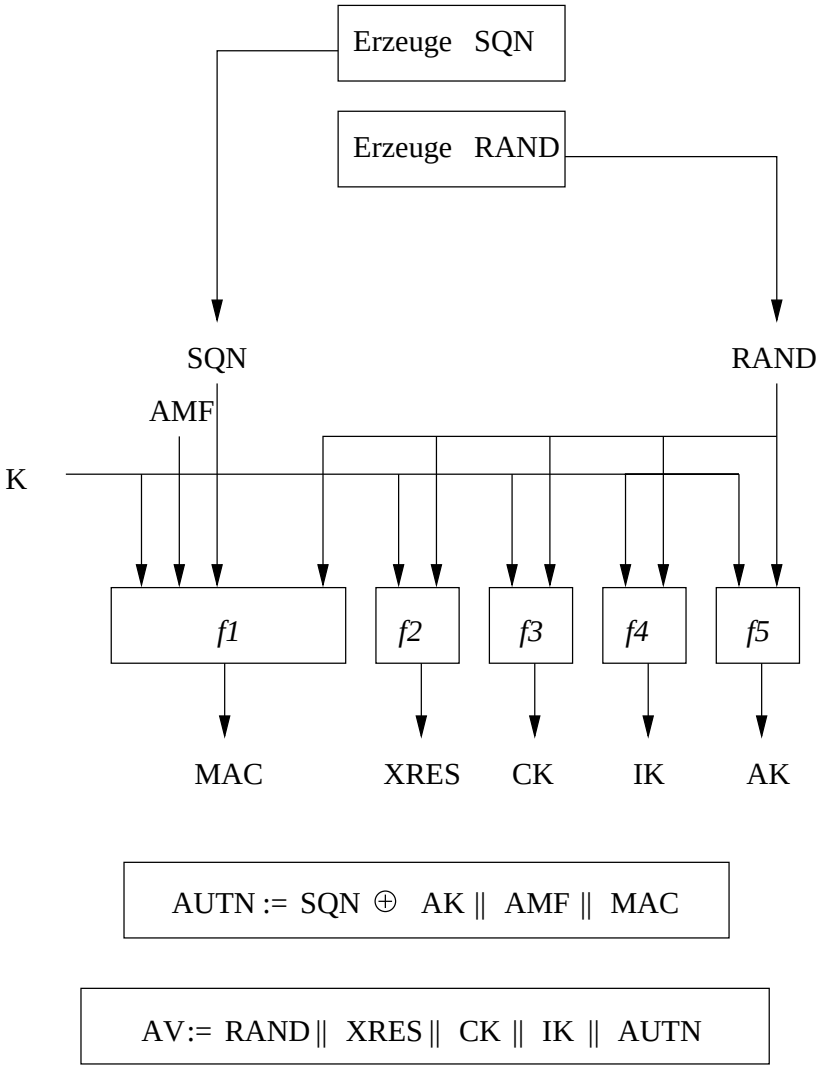

Abbildung 3.6: Erzeugen der Authentifikationsvektoren (Quelle: Eigene Darstellung)

Die einzelnen Werte werden nach folgenden Vorschriften berechnet, wobei *f1, f2* MAC-Funktionen sind, und *f3, f4, f5* Funktionen zur Schlüsselerzeugung sind[7]:

$$MAC = f1_K(SQN \| RAND \| AMF) \tag{3.1}$$

$$XRES = f2_K(RAND) \tag{3.2}$$

$$CK = f3_K(RAND) \tag{3.3}$$

$$IK = f4_K(RAND) \tag{3.4}$$

$$AK = f5_K(RAND) \tag{3.5}$$

Das Authentifikationstoken AUTN wird anschließend folgendermaßen konstruiert:

$$AUTN = (SQN \oplus AK) \| AMF \| MAC \tag{3.6}$$

Schließlich wird der Authentifikationsvektor konstruiert:

$$AV = RAND \| XRES \| CK \| IK \| AUTN \tag{3.7}$$

Die Gültigkeit des Quintetts überprüft der Teilnehmer anhand der MAC eines Vektors. Abbildung 3.7 gibt einen genaueren Einblick in die Abläufe bei der Authentifikation und Schlüsselberechnung auf der USIM, nach dem Erhalt der RAND und AUTN durch das VLR/SGSN.

Zur Ermittlung der Sequenznummer SEQ, welche Eingabeparameter von *f1* ist, muss zunächst der Schlüssel AK nach Gleichung 3.5 auf der USIM erzeugt werden. Die verschleierte SQN wird durch eine erneute XOR-Verknüpfung mit AK wie folgt berechnet:

$$SQN = (SQN \oplus AK) \oplus AK \tag{3.8}$$

Anschließend wird auf der USIM die Gültigkeit der empfangenen Authentifikationsdaten überprüft, indem die erhaltene MAC mit einer berechneten XMAC verglichen wird, die sich auf folgende Weise berechnen lässt:

$$XMAC = f1_K(SQN \| RAND \| AMF) \tag{3.9}$$

[7]Die Algorithmen *f1, f2, f3, f4, f5* sind nicht standardisiert, in den 3GPP Spezifikationen werden hierfür nur Empfehlungen bzw. Vorschläge gegeben. Jeder Hersteller kann aber seine eigenen Implementierungen verwenden, da die Algorithmen jeweils nur auf der USIM und im AuC des Heimnetzes verwendet werden und keinen Einfluss auf andere Netze haben. [3GP00]

Ist $MAC \neq XMAC$ so schickt der Teilnehmer eine *User Authentication Reject*-Nachricht an das VLR/SGSN zurück und die aktuelle Authentifiaktions-Prozedur wird von Teilnehmerseite abgebrochen.[8] Wurde die SEQ als ungültig erwiesen, wird die Prozedur ebenfalls abgebrochen, indem der Teilnehmer eine Nachricht zurückschickt und damit einen Synchronisationsvorgang einleitet (siehe [3GP02b]).

Können die Authentifikationsdaten als gültig betrachtet werden, berechnet die USIM die Antwort RES wie XRES nach Gleichung 3.2 und schickt sie in der *User Authentication Response*-Nachricht an das VLR/SGSN. Schließlich werden auf der USIM die Schlüssel CK und IK nach Gleichung 3.3 und 3.4 berechnet. Diese Schlüssel werden so lange auf der USIM gespeichert, bis die nächste erfolgreiche Authentifikation und Schlüsselvereinbarung durchgeführt wurde.

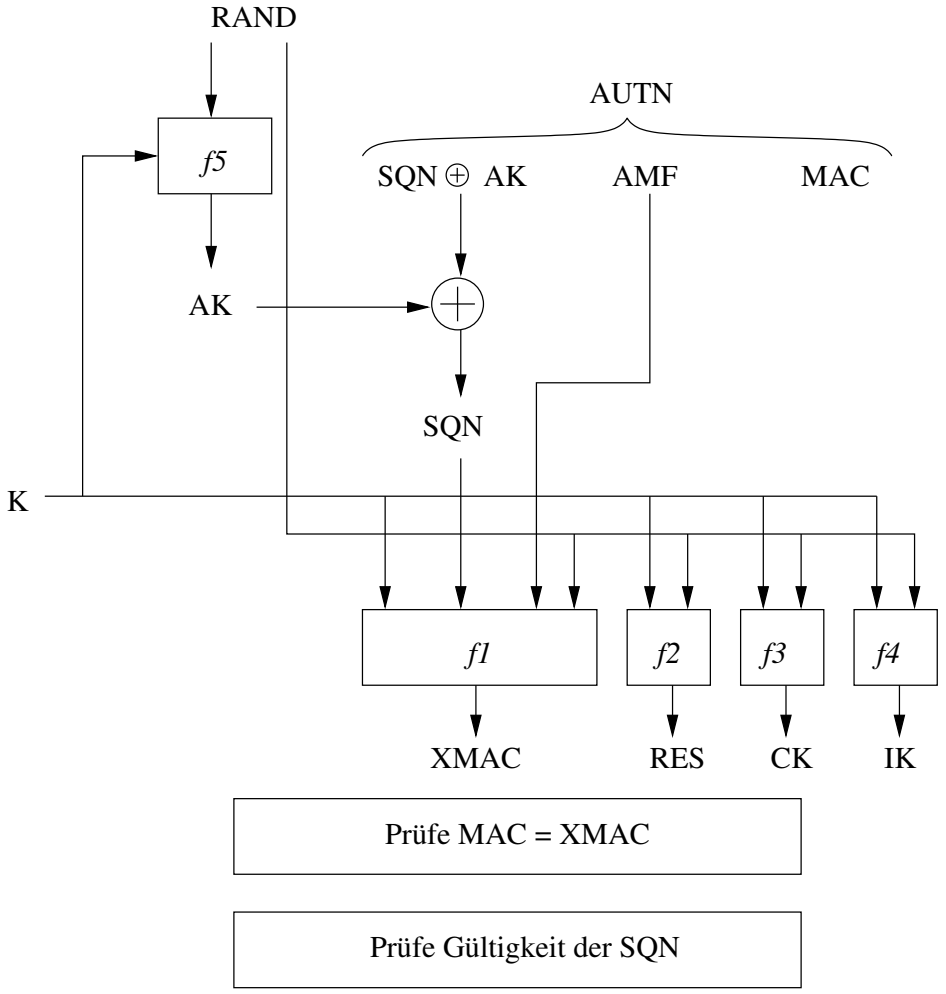

Abbildung 3.7: Teilnehmer-Authentifikation in der USIM (Quelle: Eigene Darstellung)

Zusammengefasst gilt: Das Netzwerk wird durch die USIM authentifiziert, indem $MAC = XMAC$ geprüft wird. Der Teilnehmer wird durch das VLR/SGSN authentifiziert, indem $RES = XRES$ geprüft wird.

[8] Es folgt hier eine Prozedur, auf die ich nicht eingehen werde.

3.1.3 Schlüsselverwaltung

Nach der AKA-Prozedur wird die Datenübertragung auf der Luftschnittstelle durch zwei Mechanismen gesichert, die jeweils über einen Schlüssel verfügen: Der Schlüssel CK (*Cypher Key*), der für die Ver- und Entschlüsselung der Daten verwendet wird und der Schlüssel IK (*Integrity Key*), der für die Sicherstellung der Integrität, also Unverletztheit der Daten dient. Da die AKA-Prozedur nicht verpflichtend vor jedem Verbindungsaufbau durchgeführt werden muss und somit keine neuen Schlüssel erzeugt werden, muss die Lebensdauer der Schlüssel begrenzt werden, um den dauerhaften Einsatz und möglicherweise böswilligen Wiedergebrauch von Schlüsseln bzw. verschlüsselten Daten (z.B. böswillige Replay-Angriffe [Sch96]) vorzubeugen. Aus diesem Grund ist auf der USIM ein Mechanismus implementiert, mit welchem die Datenmenge, auf die ein Schlüsselsatz angewandt wird, vom Netzbetreiber begrenzt werden kann.

Der Schlüsselsatz wird über den Parameter KSI identifiziert. Über diese Kennung gibt das Netzwerk den zu verwenden Schlüsselsatz vor, was eine – begrenzte – Wiederverwendung eines Schlüsselsatzes ermöglicht.

3.1.4 Integrität

Nach [ISO97] kann die Authentizität einer Partei nur für den Moment der Authentifikation bestimmt werden. Um auch die Authentizität anschließend übertragener Daten zu sichern, muss die Authentifikation zusammen mit Mechanismen zur sicheren Kommunikation von Daten stattfinden. Dazu ist in UMTS folgendes gegeben:

- **Vereinbarung des Integritätssalgorithmus**: MS und SN können den zur Prüfung der Integrität verwendeten Algorithmus sicher untereinander aushandeln.
- **Vereinbarung des Schlüssels zur Integritätsprüfung**: MS und SN einigen sich auf einen Schlüssel zur Prüfung der Integrität.
- **Datenintegrität und Herkunftsauthentifikation von Signalisierungsdaten**: Der Empfänger (MS oder SN) hat die Möglichkeit sicherzustellen, dass die empfangenen Signalisierungsdaten tatsächlich von dem angegebenen Sender stammen und dass diese Daten nicht unbefugt verändert wurden.

In UMTS werden die Signalisierungsdaten, die über die Luftschnittstelle übertragen werden, auf Integrität geprüft. Zu diesem Zweck wird auf alle sensiblen Signalisierungsdaten, die zwischen UE und RNC ausgetauscht werden, eine *Message Authentication Code* (MAC) Funktion angewendet, die bei UMTS als *f9* bzw. *UMTS Integrity Algorithm* (UIA) bezeichnet wird. Der Algorithmus steht sowohl im UE, als auch im RNC zur Verfügung.

Die Eingangsparameter für *f9* zur Berechnung der Prüfsumme MAC-I (32 Bit) sind:

- IK (128 Bit): der Integritätsschlüssel. Es kann jeweils für die CS Domäne (IK_{CS}) und für die PS Domäne (IK_{PS}) einen anderen Schlüssel geben, der in der entsprechenden Verbindung verwendet wird. IK wird in der USIM und

eine Kopie im ME, also im Handy selbst gespeichert. Wird das Handy ausgeschalten oder die USIM entfernt, so wird IK aus dem Speicher des ME gelöscht. Bei *Handovers* (s. [BS02]) wird IK (im Klartext) netzintern vom „alten“ zum „neuen“ RNC übertragen.

- COUNT-I (32 Bit): die Sequenznummer zur Integritätssicherung. Für jeden Signalisierungskanal auf der Luftschnittstelle gibt es jeweils für Up- und Downlink einen Wert COUNT-I. Der Wert wird pro gesicherter Nachricht um eins erhöht, womit ein Replay-Angriff während einer Verbindung verhindert wird. Es wird zudem sichergestellt, dass kein Wert von COUNT-I mit dem gleichen IK wieder verwendet wird (s. auch [3GP01]).

- FRESH (32 Bit): eine im RNC generierte und vom Netzwerk an den Teilnehmer übermittelte Zufallszahl, deren Einsatz vor Replay-Angriffen schützen soll. FRESH wird über die gesamte Dauer einer bestehenden Verbindung zwischen einem Teilnehmer und dem Netzwerk verwendet.[9]

- DIRECTION (1 Bit): zur Kennzeichnung der Übertragungsrichtung. DIRECTION sorgt außerdem dafür, dass sich die Eingangsparameter zur Berechnung der MAC-I für Up- und Downlink unterscheiden.

- MESSAGE (LENGTH): die – beliebig lange – Signalisierungsnachricht selbst, deren Integrität gesichert werden soll. Ihr wird bei der Berechnung der MAC die ID des Signalisierungskanals (vgl. COUNT-I) vorangestellt. Diese ID wird nicht mit der Nachricht übertragen, wird aber dazu benötigt, um gleiche Eingangsparameter für verschiedene MAC-Instanzen zu vermeiden.[10]

Die Berechnung des MAC erfolgt im Sender identisch wie im Empfänger. Der Sender generiert den MAC-I wie in Abbildung 3.8 veranschaulicht und hängt diesen dann an die Signalisierungsnachricht an, bevor diese über die Luftschnittstelle gesendet wird. Der Empfänger berechnet aus der empfangenen Nachricht XMAC-I und überprüft die Datenintegrität der Nachricht, indem er den XMAC-I mit dem empfangenen MAC-I vergleicht.

Der UIA kann auf verschiedenen Algorithmen basieren, die bei Verwendung anhand einer Nummer selektiert werden können. Der bisher einzige standardisierte Algorithmus ist die, auch als UIA1 bezeichnete, Blockchiffre KASUMI.

[9] In bestimmten Situationen wird ein neuer Wert für FRESH erzeugt. S.[3GP03b].

[10] Dies ist z.B. der Fall, wenn auf zwei unterschiedlichen Signalisierungskanälen, für die COUNT-I gleich ist, innerhalb einer Verbindung die gleiche FRESH verwendet wird, die gleiche Nachricht jeweils vom Netzwerk mit ensprechender DIRECTION gesendet wird und innerhalb einer Lebensdauer von IK liegt.

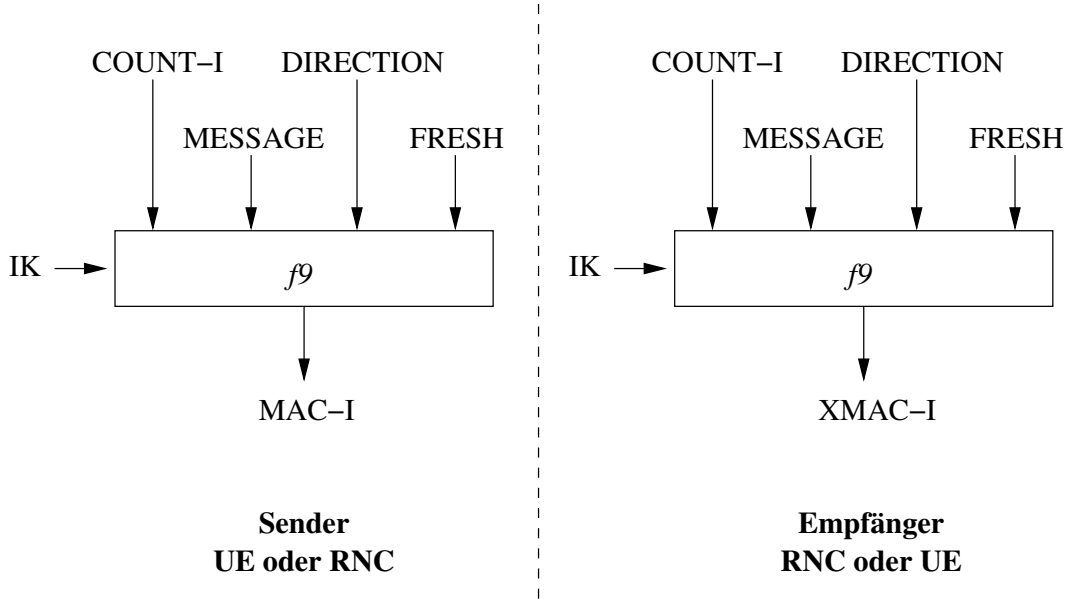

Abbildung 3.8: Erzeugung von MAC-I bzw. XMAC-I durch *f9* (Quelle: Eigene Darstellung)

KASUMI wird in *f9* im sog. „Chained Mode“ oder „Cipher Block Chaining“ (CBC) Modus [Sch96] verwendet und erzeugt einen 64 Bit langen Hash-Wert der Eingabenachricht, wovon schließlich die linken, also höherwertigen, 32 Bit als Ausgabewert für MAC-I dienen. Vor der eigentlichen Berechnung, deren Ablauf 3.9 visualisiert, wird die Funktion initialisiert, indem die Arbeitsvariablen A und B auf Null gesetzt werden und der Schlüssel-Modifikator (*Key Modifier*) KM auf hexadezimal '0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'. Die Eingangsparameter COUNT-I, FRESH, MESSAGE und DIRECTION werden aneinandergereiht und zusätzlich ein einzelnes '1'-Bit angefügt, gefolgt von z '0'-Bit, wobei $z \in \{0...63\}$, sodass die Gesamtlänge von PS (*Padded String*) ein ganzzahliges Vielfaches von 64 ist:

$$\begin{aligned} PS &= count - i_{0...31} || fresh_{0...31} || \\ &\quad message_{0...LENGTH-1} || \\ &\quad direction_0 || '1' || (z \cdot' 0') \end{aligned} \tag{3.10}$$

Die Berechnung des MAC erfolgt indem PS in BLOCKS 64-Bit Blöcke zerlegt wird, wobei gilt:

$$PS = PS_0 || PS_1 || PS_2 || ... || PS_{BLOCKS-1} \tag{3.11}$$

Anschließend wird für $0 \leq n \leq BLOCKS - 1$ der KASUMI Algorithmus mit dem Schlüssel IK auf jeden Block angewendet. Jeder Block wird mit dem Zwischenergebnis der Arbeitsvariablen A aus der vorigen Runde XOR-verknüpft. Die Arbeitsvariable B_{n+1} ergibt sich je Runde aus einer XOR-Verknüpfung von B_n mit A_n:

$$A_{n+1} = KASUMI_{IK}(A_n \oplus PS_n) \quad (3.12)$$
$$B_{n+1} = B_n \oplus A_{n+1} \quad (3.13)$$

Als letzte Berechnung erfolgt eine Anwendung des KASUMI auf B mit einem veränderten Schlüssel IK:

$$B = KASUMI_{IK \oplus KM}(B) \quad (3.14)$$

MAC-I ergibt sich schließlich aus den 32 Bit der linken Hälfte von B; die restlichen 32 Bit der rechten Hälfte $(b_{32}, b_{33}, b_{34}, ..., b_{64})$ werden verworfen. Für jedes $j \in \{0...31\}$ gilt:

$$mac - i_j = b_j \quad (3.15)$$

f9 ist keine standard CBC-MAC-Konstruktion, bei der alleine die vorletzte KASUMI-Ausgabe die Eingabe für die letzte KASUMI-Berechnung wäre, sondern stellt eine Variation dieser dar. Die Ausgaben aller vorigen KASUMI-Berechnungen werden miteinander XOR-verknüpft und liefern die Eingabe für die letzte KASUMI-Berechnung. Diese Konstruktion erreicht somit einen internen verketteten Zustand von 128 Bit, anstatt nur 64 Bit. Damit würde nach Lars Knudsen und Chris Mitchell ein Angreifer, der Nachahmungs- bzw. Fälschungsattacken durchführen will, in etwa 2^{48} Klartextnachrichten mit den zugehörigen MACs benötigen; mit der normalen Form bräuchte er 2^{32}. Der Rechenaufwand steigt außerdem nur sehr geringfügig um je eine binäre Addition eines 64-Bit-Blocks.

In welchen Protokoll-Schichten die Integritäts- und Verschlüsselungsmechanismen zur Anwendung kommen, kann [AFK+01] entnommen werden.

Die Integrität der Nutzdaten wird in UMTS nicht gesichert. Die über die Luftschnittstelle übertragenen Nutzdaten werden mit kryptographischen Algorithmen verschlüsselt. Da die Entropie der verschlüsselten Daten mit hoher Wahrscheinlichkeit ausreichend gering ist, wird davon ausgegangen, dass ein sinnvolles Ändern der Daten angemessen schwierig ist. Zumindest für Sprache ist dieser Ansatz wohl hinreichend; für sensible Daten allerdings, z.B. bei Geld-Transaktionen, reicht dies jedoch nicht aus und muss daher durch einen Ende-zu-Ende Dienst zur Integritätssicherung durchgeführt werden (z.B. SSL; s. [Ert01]). Auch die nur 32 Bit große Prüfsumme würde normalerweise als zu klein empfunden werden. Angesichts des Echtzeit-Charakters der Signalisierungsnachrichten dieses Systems, kann man die Größe jedoch als angemessen betrachten. So ist der gesamte Mechanismus für den beabsichtigten Einsatz in UMTS wohl als adequates Mittel anzusehen, um die Integrität zu schützen.

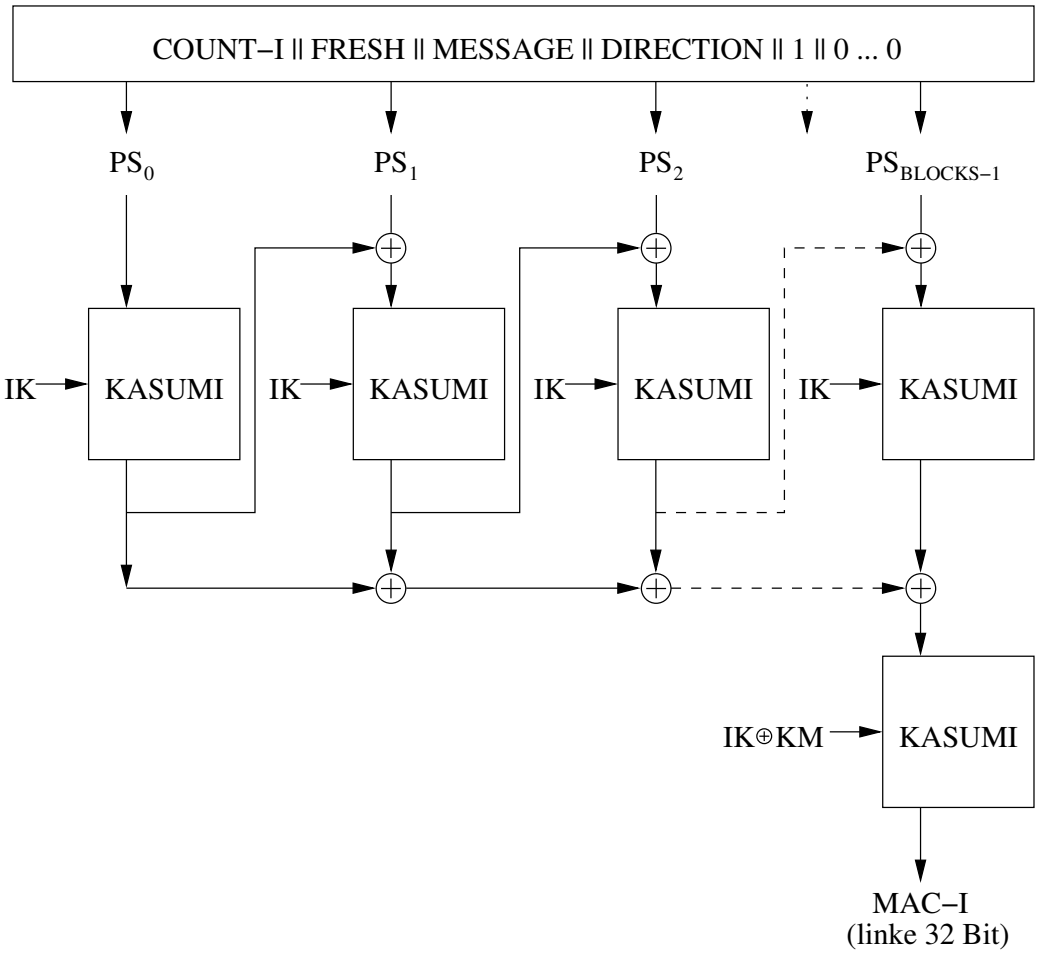

Abbildung 3.9: Die Integritätsfunktion *f9* (Quelle: Eigene Darstellung)

3.1.5 Verschlüsselung

Um den Forderungen aus Abschnitt 3.1.1 gerecht zu werden und die über die Funkschnittstelle übertragenen Nutz- und Signalisierungsdaten vor unerlaubtem Mithören zu schützen, werden Daten und Sprache vor der Übertragung mit einem Verschlüsselungsalgorithmus im RNC bzw. UE chiffriert. Dazu sind folgende Mechanismen implementiert:

- **Vereinbarung des Verschlüsselungsalgorithmus**: MS und SN können den zur Verschlüsselung verwendeten Algorithmus sicher untereinander aushandeln.
- **Vereinbarung des Schlüssels zur Datenverschlüsselung**: MS und SN einigen sich auf einen Schlüssel zur Verschlüsselung der über die Luftschnittstelle übertragenen Daten.
- **Geheimhaltung der Nutzdaten**: Die Teilnehmerdaten werden auf der Luftschnittstelle gegen Abhören geschützt.
- **Geheimhaltung der Signalisierungsdaten**: Die Signalisierungsdaten werden auf der Luftschnittstelle gegen Abhören geschützt.

Der in UMTS nach [3GP02c] standardisierte Verschlüsselungsalgorithmus *f8* hat auch die Bezeichnung UEA (*UMTS Encryption Algorithm*) und basiert auf einer Chiffre, die bei Verwendung anhand einer Kennnummer ausgewählt werden kann. Der hierfür bisher einzige standardisierte Algorithmus ist, ebenso wie bei *f9*, KASUMI mit der Kennung UEA1; die Kennung UEA0 bedeutet keine Verschlüsselung (im Falle, dass Verschlüsselung abgeschaltet werden muss). Die Funkton *f8* ist eine Stromchiffre, die Datenblöcke der Länge zwischen 1 und 20000 Bit ver-/entschlüsselt.

Für die Ver- bzw. Entschlüsselungsfunktion *f8* sind folgende Eingangsparameter nötig:

- CK (128 Bit): der Schlüssel zur Verschlüsselung. Es kann jeweils für die CS Domäne (CK_{CS}) und für die PS Domäne (CK_{PS}) einen anderen Schlüssel geben, der in der entsprechenden Verbindung verwendet wird. Die Speicherung und Weitergabe erfolgt wie bei IK. Sollte die effektive Länge von CK kleiner als 128 Bit sein, sollen die höchstwertigen Bit die effektive Information enthalten und als niederwertige Bit wiederholt werden, bis eine effektive Schlüssellänge von 128 Bit gegeben ist: $ck_n = ck_{n \bmod k}$, für alle n mit $k \leq n < 128$.
- COUNT-C (32 Bit): die Sequenznummer zur Verschlüsselung. Je Up- und Downlinkkanal gibt es einen COUNT-C. COUNT-C besteht eigentlich aus zwei Zählern, einem auf der physikalischen Ebene und einem, der die sog. Hyperrahmen zählt. Auf diesen Zählern basiert die Synchronisation (alle 10ms) des in *f9* erzeugten Schlüsselstroms und die Vermeidung der zyklischen Wiederverwendung eines Schlüsselstroms.
- BEARER (5 Bit): die eindeutige Kennung eines Funkkanals. Da der gleiche Schlüssel CK gleichzeitig auf verschiedenen Kanälen in Verbindung mit einem

Teilnehmer benutzt wird, könnte es passieren, dass aufgrund der selben Eingangsparameter der gleiche Schlüsselstrom für mehr als einen Kanal verwendet wird. Um dies zu verhindern, generiert der Algorithmus den Schlüsselstrom, basierend auf der Kennung des jeweiligen Funkkanals.

- DIRECTION (1 Bit): zur Kennzeichnung der Übertragungsrichtung. Der Parameter sorgt außerdem dafür, dass sich die Eingangsparameter zur Berechnung des Schlüsselstroms für Up- und Downlink unterscheiden.
- LENGTH (16 Bit): gibt die Länge des benötigten Schlüsselstromblocks (KEYSTREAM BLOCK) in Abgängigkeit des zu verschlüsselnden Klartextblocks (PLAINTEXT BLOCK) innerhalb eines 10ms-Rahmens an. Der Parameter wirkt sich nur auf die Länge des Blocks aus, nicht jedoch auf die eigentlichen Bit.

Abbildung 3.10 zeigt, wie *f8* verwendet wird, um einen Klartext zu ver- bzw. zu entschlüsseln. Zum Verschlüsseln wird der Klartextblock beim Sender Bit für Bit mit dem erzeugten Schlüsselstrom binär addiert, also XOR-verknüpft, und somit ein Chiffretext erzeugt. Auf der Seite des Empfängers wird dieser Chiffretext wiederum mit dem gleichen, zum Verschlüsseln verwendeten, Schlüsselstrom, der auf Empfängerseite mit den gleichen Eingabeparametern erzeugt wurde, ebenso bitweise XOR-verknüpft. Als Ergebnis der Verknüpfung erhält man wieder den Klartextblock wegen:

$$p_i \oplus k_i \oplus k_i \quad = \quad p_i \tag{3.16}$$

Die Sicherheit eines solchen Systems beruht vollständig auf dem Schlüsselstromgenerator, dessen Qualität seines Ausgabestroms über die Sicherheit des Chiffretextes entscheidet. Erzeugt er sich zyklisch wiederholende Bitmuster mit geringer Länge von z.B. 16 Bit, so würde der Algorithmus nur eine einfache XOR-Verknüpfung darstellen, was wiederum nichts anderes ist als eine polyalphabetische Vigenére-Chiffre [Sch96], die keine große Schutzwirkung bringt. Liefert der Generator dagegen echte Zufallsbits, hätte man ein „One-Time-Pad“ und damit eine Chiffre, die perfekte Sicherheit liefern würde. Den Schlüsselstrom, der in *f8* erzeugt wird, kann man irgendwo dazwischen einstufen. Der Schlüsselstromgenerator liefert einen Bitstrom, der in zumindest sehr großer Zeit keine Zyklen aufweist und zufällig erscheint. Trotzdem ist der Strom deterministisch und kann zuverlässig zur Entschlüsselung reproduziert werden. Wichtig ist vor allem, dass der Schlüsselstrom bei jedem Einsatz anders ist, weshalb die verschiedenen Eingabeparameter und vor allem ein Schlüssel verwendet wird.

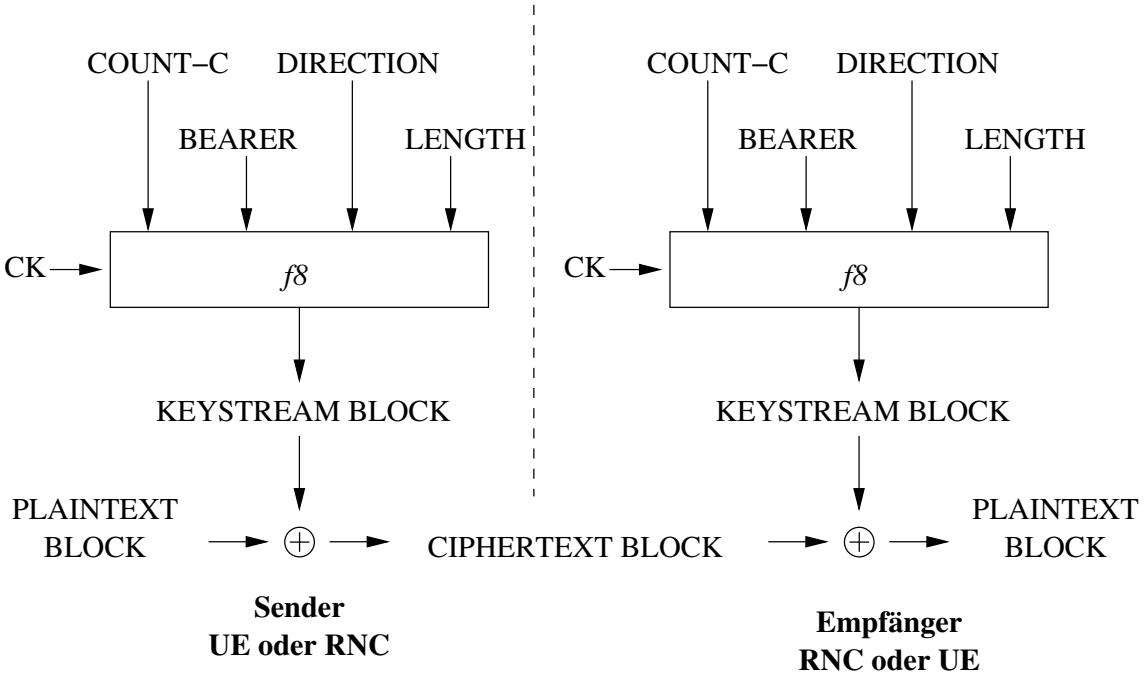

Abbildung 3.10: Verschlüsselung der Signalisierungs- und Nutzdaten auf der Luftschnittstelle (Quelle: Eigene Darstellung)

Die wichtigsten Anforderungen bei der Entwicklung von *f8* waren:

- Die Funktion *f8* soll vollständig standardisiert sein und soll eine symmetrische Stromchiffre sein.
- Die Funktion soll mindestens die nächsten 20 Jahre verwendet werden können.
- Erfolgreiche Angriffe mit wesentlich geringerem Aufwand, als dem Probieren aller möglichen Schlüssel des effektiven Schlüsselraums, sollen unmöglich sein.
- UEs, RNCs und AuCs, die solche kryptographischen Algorithmen implementieren, sollen ohne jegliche Einschränkungen zum Export und Verkauf nach dem Wassenaar-Abkommen freigegeben werden.
- Es sollen Verschlüsselungsgeschwindigkeiten bis 2Mbit/s realisierbar sein.

Der Schlüsselstromgenerator, der in Abbildung 3.11 dargestellt ist, basiert auf dem KASUMI-Algorithmus, der hier in einem sog. „Output Feedback“-Modus in Kombination mit einem „Counter“-Modus [Sch96] verwendet wird und den Schlüsselstrom in 64 Bit großen Blöcken erzeugt.[11] Vor der Berechnung wird die Funktion initialisiert. Ein 64 Bit großes Arbeitsregister A wird auf $A = COUNT - C||BEARER||DIRECTION||(26 \cdot '0')$ gesetzt und stellt somit den Initialisierungsvektor IV dar. Diese Konstante wird auch „pre-whitening“-Konstante genannt und schützt gegen „known plaintext“-Angriffe, da ein Abhörer die Eingaben bestimmter

[11] Durch den Einsatz der BLKCNT Variablen wird das frühzeitige Bilden von Zyklen verhindert.

KASUMI-Anwendungen nicht wissen kann. Die Zählervariable BLKCNT wird auf Null gesetzt, der Schlüssel-Modifikator (*Key Modifier*) KM auf '0x55555555555555-555555555555555555' und der Startblock KSB_0 des Schlüsselstromgenerators ebenfalls auf Null (Dies ist zur leichteren Implementierung da eine XOR-Operation mit Null keine Veränderung bringt). Anschließend wird KASUMI mit einem veränderten Schlüssel CK einmal auf das Register A angewendet:

$$A = KASUMI_{CK \oplus KM}(A) \tag{3.17}$$

Der Klar-/Chiffretext, der zu ver-/entschlüsseln ist, besteht aus LENGTH (1 - 20000) Bit, wärend sich der Schlüsselstrom aus 64 Bit großen Blöcken zusammensetzt, also ein Vielfaches von 64 Bit lang ist. Um die Länge des Schlüsselstroms der Länge der Daten anzupassen, werden die niederwertigsten Bit des letzten Blocks verworfen. Die Anzahl der Blöcke BLOCKS ergibt sich dann aus LENGTH/64, dessen Ergebnis auf die nächst größere ganze Zahl aufgerundet wird.

Ein Block KSB_n des Schlüsselstroms resultiert aus einer Anwendung des KASUMI mit dem Schlüssel CK auf das Ergebnis der XOR-Verknüpfung von A mit dem Zähler BLKCNT = (n-1) mit dem Schlüsselblock KSB_{n-1} der vorigen Runde. Es gilt $1 \leq n \leq BLOCKS$:

$$KSB_n = KASUMI_{CK}(A \oplus BLKCNT \oplus KSB_{n-1}) \tag{3.18}$$

Die einzelnen Blöcke KSB_1 bis KSB_{BLOCKS} werden jeweils mit dem höchstwertigen Bit zuerst aneinandergereiht und ergeben den Schlüsselstrom KS nach:

$$ks_{((n-1)\cdot 64)+i} = ksb_{n-1,i} \tag{3.19}$$

mit $1 \leq n \leq BLOCKS$ und $0 \leq i \leq 63$.

Die eigentliche Ver- und Entschlüsselung erfolgen zueinander identisch, indem der Eingangsdatenstrom (IBS) bitweise XOR-verknüpft wird mit dem Schlüsselstrom; der Ausgangsdatenstrom (OBS) wird erzeugt.

$$obs_i = ibs_i \oplus ks_i \tag{3.20}$$

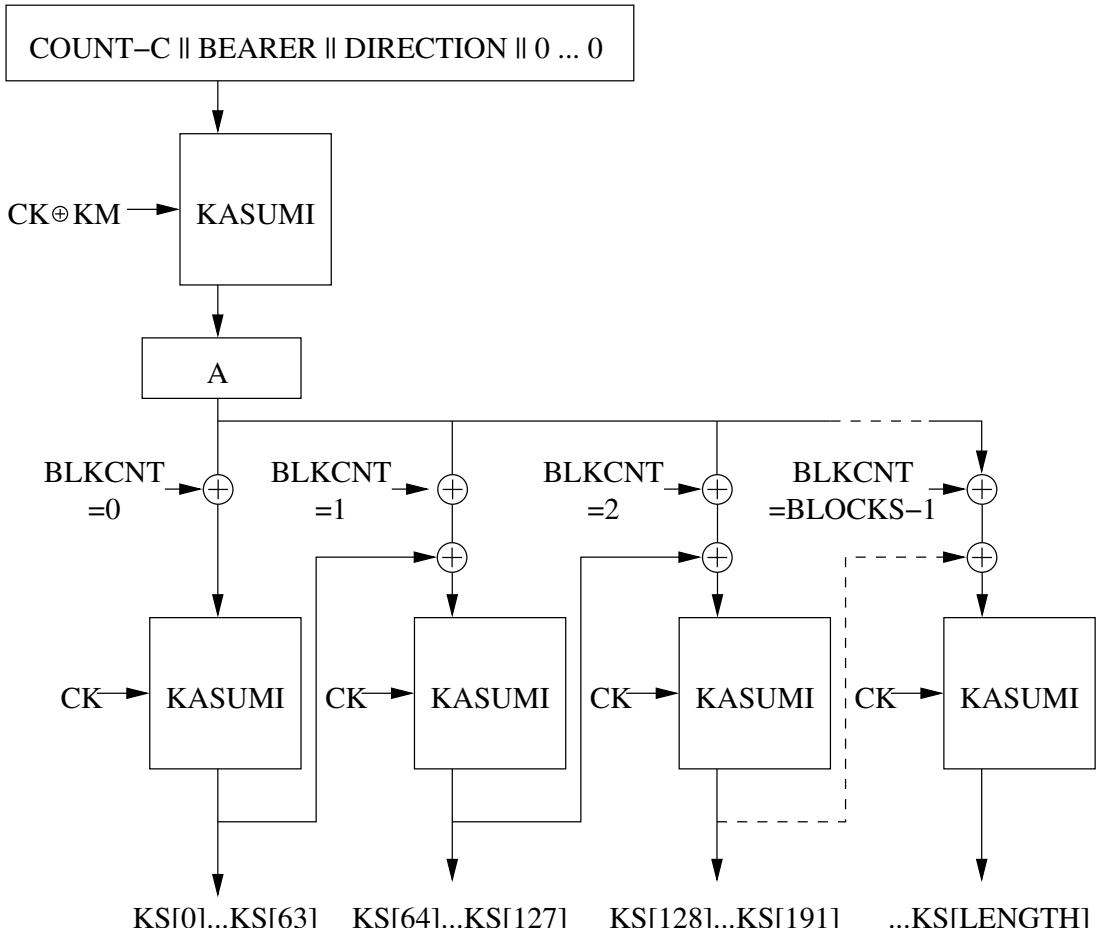

Abbildung 3.11: Die Verschlüsselungsfunktion *f8* (Quelle: Eigene Darstellung)

3.2 Netzwerksicherheit

Die Sicherheit des drahtgebundenen Netzwerks ist wie bei GSM auch in UMTS im System (noch) nicht integriert. Zwischen den einzelnen Knoten werden alle Daten im Klartext übertragen, sofern keine „Ende-zu-Ende“ Applikation die Daten sichert. So werden z.B. zwischen den RNCs und zwischen VLR/SGSN und RNC die während einer Verbindung verwendeten Schlüssel CK/IK im Klartext übertragen. Eine Authentifikation findet zwischen den einzelnen Knoten unbekannter Netze nicht statt. Eine Sicherung des Netzwerks ist bisher noch nicht spezifiziert worden, es sind allerdings Mechanismen für spätere Versionen geplant.

Auf bestimmten Übertragungsschichten, wie z.B. dem MAP (*Mobile Application Part*) oder IP (*Internet Protocol*) sind Protokollerweiterungen geplant, wie z.B. MAPsec oder IPsec, die die Daten innerhalb des Netzes sichern sollen (s. [3GP02a] u. [3GP03a]). Für die Schichten, in denen solche Sicherheitserweiterungen nicht implementiert werden, hängt die Sicherheit der übertragenen Daten somit alleine von der geographischen Lage der Knoten bzw. der Leitungen ab. Sämtliche Leitungen, auf denen unverschlüsselte Daten übertragen werden, sind private, unterirdische Backbones, deren Zugang nur authorisiertem Personal gestattet ist. Komponenten wie RNC, MSC/VLR, SGSN, GGSN oder HLR/AuC sind üblicher Weise in gesicherten Räumen in Gebäuden untergebracht.

3.3 Benutzersicherheit

Die Bentuzersicherheit, also die Sicherheit auf Benutzerbene der Endgeräte, ist bei UMTS hauptsächlich in zwei Arten vorhanden. Die eine ist die sog. „User-to-USIM“ Authentifikation und bietet Schutz vor unerlaubtem Zugriff auf die USIM. Benutzer und USIM kennen beide einen geheimen Schlüssel, die PIN (*Personal Identification Number*), der in der USIM sicher gespeichert ist und gegen Auslesen sowohl durch Software, als auch durch Hardwareimplementationen geschützt ist. Siehe hierzu auch die Spezifikationen [3GP02g] und [3GP02f].

Die zweite Art der Benutzersicherheit besteht zwischen USIM und Endgerät (*Terminal*). Hierfür kennen auch USIM und MS einen geheimen Schlüssel, der in beiden sicher gespeichert ist und nur bestimmten USIMs den Zugriff auf ein Terminal erlauben soll. Mehr Details zu diesem Thema können [3GP02e] entnommen werden.

3.4 Anwendungssicherheit

Die zwischen den einzelnen Applikationen auf der USIM, dem MS und dem Netzwerk ausgetauschten Informationen, können vom Netzbetreiber oder vom Anbieter der Applikation entsprechend ihrer Anforderungen geschützt werden. In [3GP03c] sind entsprechende Mechanismen beschrieben, auf die ich hier allerdings nicht näher eingehen werde.

Einen Überblick eines solchen Szenarios verschafft Abbildung 3.12 anhand von Finanztransaktionen über den SMS Dienst.

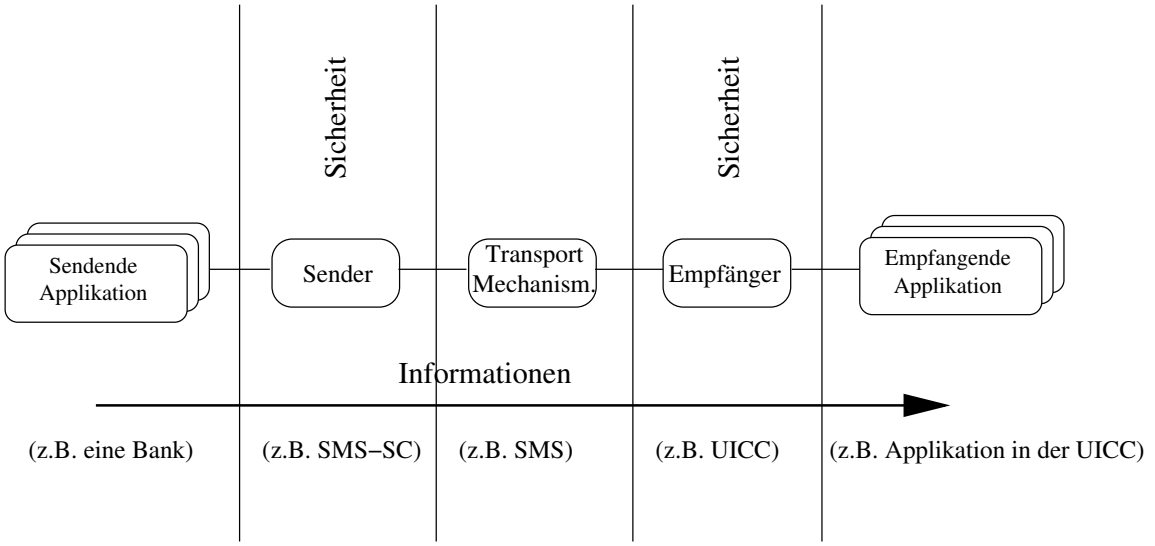

Abbildung 3.12: Sicherheit zwischen Anwendungen (Quelle: Eigene Darstellung)

3.5 Sichtbarkeit und Konfigurierbarkeit der Sicherheit

3.5.1 Sichtbarkeit

Üblicherweise sind die Sicherheitseigenschaften für den Benutzer transparent und er weiß nicht, welche Dienste derzeit aktiviert sind und welche nicht. In manchen Fällen jedoch kann es für den Benutzer durchaus von Vorteil sein, wenn er über den Sicherheitsstatus einer Verbindung informiert wird, z.B. bei Einkäufen im Internet über sein Handy oder Finanztransaktionen. Deshalb wird der Benutzer in UMTS darüber informiert, ob seine Nutzdaten der Verbindung verschlüsselt werden oder nicht. Gelangt der Benutzer in ein fremdes/anderes Netzwerk, so wird ihm der Sicherheitsgrad, den dieses Netz leisten kann, ebenfalls bekannt gegeben. Dies ist z.B. speziell der Fall, wenn ein Benutzer ein UMTS-Netz während einer Verbindung verlässt und über ein GSM-Netz weitertelefoniert.

3.5.2 Konfigurierbarkeit

In UMTS kann der Benutzer selbst entscheiden, ob die Benutzung eines bestimmten Dienstes davon abhängen soll, ob ein bestimmtes Sicherheitsmerkmal aktiv ist oder nicht. So kann der Benutzer seine Dienste so konfigurieren, dass er nur die Dienste zur Verfügung hat und sein Terminal nur die Dienste verwenden darf, die mit den vorhandenen, aktivierten Sicherheitsmerkmalen erlaubt sind. So können z.B. ankommende Anrufe, die nicht verschlüsselt sind, sofort abgelehnt werden oder abgehende Anrufe in einem Netz, dass keine Verschlüsselung anbietet, werden erst gar nicht erlaubt. Ebenso können nur bestimmte Sicherheits-Algorithmen für eine Verbindung akzeptiert werden.

4. KASUMI

Bei der Entwicklung der Sicherheitsalgorithmen für das 3GPP durch die ETSI SAGE (*Security Algorithms Group of Experts*) 1999 kam es darauf an, möglichst einfache Algorithmen mit einem hohen Grad an Sicherheit zu entwickeln, die sowohl in Software, als auch in Hardware sehr schnell (schneller als die Datenrate bei UMTS, 2Mbit/s) und effektiv (niedrige Leistungsaufnahme und weniger als 10.000 logische Gatter) implementierbar sein sollten. Man entschied sich, die beiden standardisierten Algorithmen *f8* und *f9* auf der selben Chiffre aufzubauen. Als absolut geeignet stellte sich die Blockchiffre MISTY1 [Mat97] heraus, die 1996 im Auftrag der Mitsubishi Electric Corporation von Mitsuru Matsui[1] und seinem Team entwickelt wurde und von Mitsubishi frei für das 3GPP Projekt zur Verfügung gestellt wurde. MISTY1 gehört mittlerweile zu den fünf geprüften und ausgewählten Block-Algorithmen des NESSIE-Projekts[2], wo sich unter anderen auch AES wiederfindet. MISTY1 ist industrieller Standard RFC-2994.

Die Entwickler entschieden sich also, eher einen schon bestehenden Algorithmus zu adaptieren, als einen komplett neuen zu schaffen, dessen Sicherheit erst noch erwiesen werden müsste. Es kam den Entwicklern darauf an, den neuen Algorithmus nicht über-zu-designen und komplizierter zu machen als er tatsächlich für seine Anforderungen sein müsste. Er sollte gegen alle praktischen Angriffe schützen und nicht gegen unrealistische, theoretische Angriffe.

Schließlich wurde eine modifizierte Version der MISTY1-Chiffre entwickelt namens KASUMI[3]. KASUMI ist eine symmetrische Blockchiffre, die sozusagen das Herzstück der Sicherheitsalgorithmen *f8* und *f9* darstellt und in [3GP02c] durch die 3GPP spezifiziert ist, womit er das Kerkhoffs-Prinzip voll erfüllt. Jeder kann den Algorithmus genau analysieren und Kryptographieexperten haben die Möglichkeit, die Sicherheit zu testen. Heute wird KASUMI nicht nur in UMTS eingesetzt, sondern ist auch als neuer Verschlüsselungsalgorithmus A5/3 in GSM-Endgeräten und Basisstationen implementiert.

[1] Erfinder der linearen Cryptanalyse

[2] New European Schemes for Signatures, Integrity and Encryption

[3] KASUMI ist japanisch und steht für das engl. Wort „misty“, was „dunstig, unklar“ bedeutet.

4.1 Übersicht

KASUMI ist eine reine 8-Runden Feistel-Chiffre mit einer Blocklänge von 64 Bit und einer Schlüssellänge von 128 Bit. MISTY1 wurde basierend auf der Theorie der beweisbaren Sicherheit gegen differentielle und linear Kryptanalysen [NK95] entwickelt und KASUMI sollte diese Eigenschaften natürlich übernehmen.

Die Funktion und ihre Unterfunktionen FO, FI, FL sind in Abbildung 4.1 dargestellt. Der Klartextblock wird in je zwei 32 Bit Hälften L_0 und R_0 aufgeteilt und durchläuft anschließend die acht Runden, bevor die zwei Hälften wieder zusammengesetzt werden.

4.2 Die Rundenfunktion

Die linke Hälfte des 64 Bit großen Eingabeblocks durchläuft die Rundenfunktion f_i mit dem Rundenschlüssel K_i (enthält die Teilschlüssel KL_i, KO_i, KI_i). Das Ergebnis wird mit der rechten Hälfte XOR-verknüpft und als linke Hälfte der nächsten Runde übernommen. Die unveränderte linke Hälfte wird in der jeweils folgenden Runde zur rechten Hälfte. Die i-te Runde von KASUMI ergibt sich somit nach

$$R_i = L_{i-1} \tag{4.1}$$
$$L_i = R_{i-1} \oplus f_i(L_{i-1}, K_i) \tag{4.2}$$

Wobei sich nach der achten Runde der Chiffretext durch das Zusammensetzen der beiden Hälften ergibt als $C = L_8||R_8$.

Die Rundenfunktion f_i stellt sich für gerade und ungerade Rundennummern unterschiedlich dar. Als Eingabe erhält sie einen 32-Bit-Block X und einen 128-Bit-Rundenschlüssel K_i. Die Funktion selbst besteht aus den Unterfunktionen FL_i und FO_i. Die 128 Bit des Rundenschlüssels verteilen sich wie folgt auf die Unterfunktionen:

- 32 Bit für FL (Rundenteilschlüssel KL_i)
- 2×48 Bit für FO (Rundenteilschlüssel KO_i und KI_i)

f_i ergibt sich dann je nach Rundennummer aus der Fallunterscheidung:

$$f_i = \begin{cases} FO(FL(X, KL_i), KO_i, KI_i), & \text{falls } i \text{ ungerade} \\ FL(FO(X, KO_i, KI_i), KL_i), & \text{falls } i \text{ gerade} \end{cases} \tag{4.3}$$

4.2.1 Die Unterfunktionen *FL, FO* und *FI*

Runden mit ungeraden Nummern berechnen zuerst FL und anschließend FO; bei geraden Rundennummern genau anders herum. Die Funktion FL dient dazu, den 32-Bit-Eingabeblock L_i und den 32-Bit-Teilschlüssel KL_i jeweils in zwei 16 Bit große Hälften aufzuteilen; L, R und $KL_{i,1}$, $KL_{i,2}$. Die Ausgabe resultiert dann aus der Berechnung:

$$FL(L_i, KL_i) = L'||R', \qquad \text{mit} \tag{4.4}$$
$$R' = R \oplus \lll (L \cap KL_{i,1}), \qquad \text{und} \tag{4.5}$$
$$L' = L \oplus \lll (R \cup KL_{i,2}) \tag{4.6}$$

Die Funktion FO ist selbst wieder ein Feistel-Netzwerk mit drei Runden. Die 32-Bit-Eingabe wird in zwei 16-Bit-Hälften L_j, R_j geteilt. L_j durchläuft anschließend in drei Runden die XOR-Verknüpfung mit dem Teilschlüssel $KO_{i,j}$, die Funktion FI mit dem jeweiligen Teilschlüssel $KI_{i,j}$ und eine XOR-Verknüpfung mit R_j. Die jeweiligen 48-Bit-Teilschlüssel KO_i und KI_i (vgl. Abschnitt 4.2.2) werden in 16-Bit-Blöcke aufgeteilt. L_j und R_j berechnen sich wie

$$L_j = R_{j-1} \tag{4.7}$$
$$R_j = FI(L_{j-1} \oplus KO_{i,j}, KI_{i,j}) \oplus R_{j-1} \tag{4.8}$$

Das Resultat ergibt sich dann aus $(L_j||R_j)$.

In der Funktion FI wird der 16 Bit große Eingangsblock in einen linken 9-Bit- und einen rechten 7-Bit-Block aufgeteilt. Beide Blöcke durchlaufen dann je zweimal eine Substitution (S-Box), nämlich S9 bzw. S7. Die Funktion „zero-extend“, wie sie in Abbildung 4.1 bezeichnet ist, erweitert einen 7-Bit- zu einem 9-Bit-Block, indem an die höchstwertige Stelle zwei Null-Bit vorangesetzt werden. „truncate“ verkürzt einen 9-Bit-Block zu einem 7-Bit-Block, indem die zwei höchstwertigen Bit verworfen werden.

Die S-Boxen S9 und S7 sind so entwickelt worden, dass sie sowohl in kombinatorischer Logik, wie auch als Lookup-Tabellen möglichst einfach und effizient implementierbar sind. In der Algebraischen Normalform (ANF) erkennt man sehr schön, dass die KASUMI-S-Boxen für eine einfache Hardware-Implementierung optimiert sind; die Anzahl der Terme ist im Vergleich zu einer zufällig gewählten S-Box sehr klein. Beide Formen der S-Boxen finden sich im Anhang B.

4.2.2 Die Rundenteilschlüssel

Die 128 Bit des Schlüssels K dienen bei KASUMI dazu, um die Rundenteilschlüssel KL_i, KO_i, KI_i davon abzuleiten. Der Hauptschlüssel K wird dazu zunächst in acht 16-Bit-Blöcke K_j aufgeteilt, nämlich $K = K_1||K_2||K_3||K_4||K_5||K_6||K_7||K_8$.

Die acht 16-Bit-Blöcke K'_j entstehen aus einer XOR-Verknüpfung der Blöcke K_j mit den Konstanten C_j (s. Anhang C).

$$K'_j = K_j \oplus C_j \tag{4.9}$$

Die Rundenteilschlüssel ergeben sich dann aus K_j und K'_j mit der Tabelle C.1 im Anhang C.

Beispiel: Der Teilschlüssel $KO_{5,3}$, also der Schlüssel für die fünfte Runde von KASUMI und für die dritte Runde der Funktion FO, ergäbe sich dann aus der „Left-Shift“-Operation $K_3 \lll 13$.

4.3 Sicherheit

KASUMI, *f8* und *f9* wurden entwickelt, um in erster Linie die Sicherheit im UMTS-Kontext zu gewähren. Mögliche Risiken außerhalb dieses Kontexts wurden sehr wohl

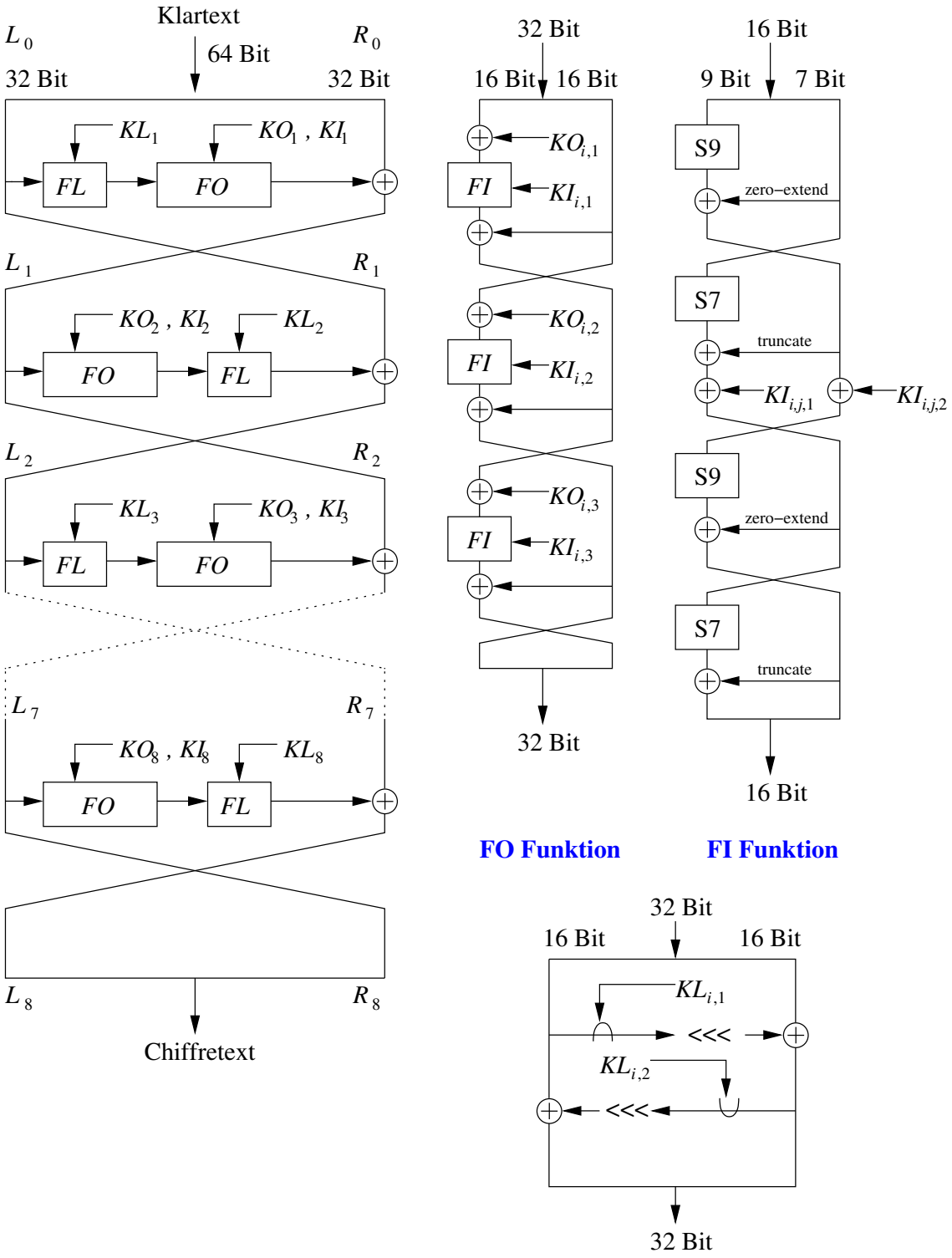

Abbildung 4.1: Die Blockchiffre KASUMI (Quelle: Eigene Darstellung)

betrachtet und Veränderungen am Design überdacht, jedoch nicht notwendiger Weise umgesetzt. Denn steigende Komplexität durch neue Veränderungen wurde meist kritischer bewertet, als die Risiken nur theoretisch realistischer Angriffe.

Die tatsächliche Sicherheit von KASUMI basiert auf der nichtlinearen Funktion FO bzw. FI und der Qualität ihrer S-Boxen, die das eigentliche nichtlineare Element darstellen. Alle Funktionen in KASUMI sind schlüsselabhängig; die Sicherheit basiert natürlich hauptsächlich auf der Geheimhaltung des Schlüssels!

Bei der Entwicklung wurden bestimmte Angriffe gegen KASUMI besonders berücksichtigt (vgl. [SAG01]):

- lineare Kryptanalyse
- differentielle Kryptanalyse und Varianten, wie „miss in the middle“ etc.
- „higher order“ differentielle Kryptanalyse mit Differentialen höherer Ordung und Interpolationen
- schwache Schlüssel

Allgemein kann gesagt werden, dass derzeit keine praktischen Angriffe gegen KASUMI bestehen, die erfolgreich wären. Es wurden keine schwachen Schlüssel gefunden und Angriffe, die gegen KASUMI erfolgreich sein können, sind dies nur mit einer reduzierten Funktion von maximal fünf bis sechs Runden. Als Kern der Funktionen *f8* und *f9* und in dieser Kombination im UMTS-Kontext sind keine der bekannten starken Angriffe praktikabel mit verhältnismäßigen Mitteln anwendbar.

Bei der Prüfung des Algorithmus durch mehrere unabhängige Experten, wurden Verbesserungsvorschläge gemacht, die bisher aufgrund oben genannter Gründe nicht implementiert worden sind. Diese waren z.B., die Anzahl der Runden zu erhöhen, FO um eine vierte Runde zu erweitern und die Schlüsseltabelle komplizierter zu gestalten.

4.3.1 Nichtlinearität der S-Boxen

Die Sicherheit des Algorithmus basiert, wie schon erwähnt, auf der Nichtlinearität der Substitutionsfunktionen; der S-Boxen S7 und S9 (Anhang B). Die Lookup-Tabellen beider S-Boxen können als Funktionen $S7 : GF(2^7) \mapsto GF(2^7)$ bzw. $S9 : GF(2^9) \mapsto GF(2^9)$ betrachtet weren, die wiederum als Vektoren gesehen werden können

$$S7((x_6, ..., x_0)) = (u_1(x_6, ..., x_0), ..., u_7(x_6, ..., x_0)) = (y_6, ..., y_0) \quad (4.10)$$

$$S9((x_8, ..., x_0)) = (v_1(x_8, ..., x_0), ..., v_9(x_8, ..., x_0)) = (y_8, ..., y_0) \quad (4.11)$$

der Funktionen $u_i : GF(2^7) \mapsto GF(2)$ bzw. $v_j : GF(2^9) \mapsto GF(2)$, mit $1 \leq i \leq 7$ und $1 \leq j \leq 9$.

S7 wurde speziell für den Hardwareeinsatz mit kombinatorischer Logik entwickelt. Als Folge ist der Grad der Nichtlinearität drei, was in Anhang B.1 als Darstellung in der algebraischen Normalform schön zu sehen ist. S7 ist eine lineare Transformation

$$x \mapsto x^{81} \qquad \text{in } GF(2^7) \quad (4.12)$$

mit optimalen nichtlinearen Eigenschaften. Der Exponent 81 ist ein sog. Kasami-Exponent, wodurch nach [Dob99] bewiesen werden kann, dass er diese optimalen Eigenschaften besitzt.

S9 ist ebenfalls relativ leicht in Hardware zu implementieren und ihr Grad der Nichtlinearität ist zwei. S9 kann als eine lineare Transformation der Potenzfunktion

$$x \mapsto x^5 \qquad \text{in } GF(2^9) \tag{4.13}$$

gesehen werden und erreicht nur fast perfekte Nichtlinearität.

Hinsichtlich linearer Strukturen [SAG01] weist S9 aufgrund ihres quadratischen Grades folgende Eigenschaft auf: Ist $GF(2^n) \mapsto GF(2^n)$ eine fast perfekte nichtlineare Permutation, derer Ausgabe-Bit quadratische Funktionen der Eingabe-Bit sind und $f_{\vec{u}}(\vec{x}) = s(\vec{x}) \bullet \vec{u}$ eine nichtlineare Kombination der Ausgabe-Bit von s, dann gibt es genau eine lineare Struktur von $f_{\vec{u}}$. D.h. es gibt einen Null-Vektor $\vec{w}$ mit $f_{\vec{u}}(\vec{x}) \oplus f_{\vec{u}}(\vec{x} \oplus \vec{w}) = konstant$. Dies kann man eine lineare Struktur $(\vec{u}, \vec{w})$ der Permutation s nennen. Damit besitzt die Tabelle S9 511 lineare Strukturen $(\vec{u}, \vec{w})$, die aber aufgrund der speziellen Konstruktion nicht zu linearen Strukturen in FI und somit in FO führen.

Die S7 und S9 sind keine zufälligen S-Boxen. Jedes Ausgabe-Bit ist von jedem Eingabe-Bit abhängig. Die S-Boxen sind für den sog. Lawineneffekt mit verantwortlich, was bedeutet, dass eine minimale Änderung im Klartext-Block eine dramatische Änderung im zugehörigen Block des Chiffretextes zur Folge hat. Diese Eigenschaft erfüllen S7 und S9 alleine allerdings nicht.

4.3.2 Mathematische Analyse

Die mathematische Analyse zeigt, wie die Sicherheit der einzelnen Komponenten aus mathematischer Sicht zu bewerten sind.

Die Funktion FL ist eine lineare Funktion und trägt nicht maßgeblich zur Sicherheit des Algorithmus bei. Die Hauptaufgabe dieser Funktion ist es, durch die Shift- und Binäroperationen mit den Teilschlüsseln, die einzelnen Bit über die Runden schwieriger nachvollziehen zu können und dies mit sehr einfachen Methoden mit sehr wenig Rechenaufwand.

Die Funktion FO mit ihrer Unterfunktion FI stellen die Zufallsfunktion des Algorithmus dar und sind aufgrund der Substitutions-Boxen nichtlinear. Durch die ungerade Division des 16-Bit-Eingabeblocks in 9 Bit und 7 Bit kann mit Theorem 4 in [Mat97] gezeigt werden, dass die durchschnittliche lineare und differentielle Wahrscheinlichkeit DP^{FI} und LP^{FI} von FI kleiner ist als $2^{-9+1}2^{-7+1} = 2^{-14}$, wenn alle Bit der Teilschlüssel unabhängig sind. DP^{FO} bzw. LP^{FO} mit einer geraden Division in zwei 16-Bit-Hälften ist damit kleiner als $2^{-16+2}2^{-16+2} = 2^{-28}$. Die differentielle/lineare Wahrscheinlichkeit bestimmt die Sicherheit eines Feistel-Netzwerks gegen differentielle/lineare Kryptanalysen und gibt die Wahrscheinlichkeit an, differentielle/lineare Merkmale und Eigenschaften einer Funktion finden zu können. Für sichere Chiffren muss die maximale differentielle/lineare Wahrscheinlichkeit ausreichend klein sein [Mat97].

Die DP bzw. LP von drei Runden des KASUMI Algorithmus ist nicht größer als 2^{-56} und Methoden der differentiellen und linearen Kryptanalyse haben sich gegen acht Runden als erfolglos erwiesen. Angriffe mit bis zu sechs Runden sind zum Teil mit sehr großem Aufwand möglich [ETS99].

4.3.3 Pragmatische Sicherheit

Wie schon erwähnt wurde bei der Entwicklung sehr darauf geachtet, die Komplexität des Algorithmus nicht unnötig zugunsten der Sicherheit zu erhöhen, ohne auch einen praktischen Nutzen davon zu haben, der sich im eingesetzten Umfeld bewährt.

Ein gutes Beispiel für diesen sehr pragmatische Ansatz ist die Verschlüsselungsfunktion *f8*. Müsste ein sehr langer Schlüsselstrom erzeugt werden – Größenordnung 2^{38} Bit –, wäre es möglich, eine wiederkehrende Struktur im Schlüsselstrom zu erkennen. Dies würde man normalerweise und in einer anderen Anwendung sicherlich als unerwünscht bewerten und mögliche Angriffe erwarten. Im UMTS-Kontext kann dies jedoch getrost vernachlässigt werden, da (zumindest in der jetzigen Version der Spezifikation) die maximale Länge des Klartextstroms und somit des benötigten Schlüsselstroms 20 kBit beträgt.

4.3.4 Seiteneffekte

Angriffe aufgrund von Seiteneffekten gegen KASUMI wurden in verschiedensten Variationen getestet und es wurde keine Eigenschaft des Algorithmus gefunden, die ihn besonders gefährdet gegen eine solche Art von Angriffen machen würde. Besonders die Schlüsseltabelle hat sich als effizient gegen Methoden gezeigt, bei denen der Stromverbrauch gemessen wird und daraus Rückschlüsse auf das Hamming-Gewicht[4] der Teilschlüssel-Bytes gemacht werden können (z.B. „differential power analysis“, „simple power analysis“, s. [Ert01]). Manche Angriffe dieser Art können aber vernachlässigt werden, da die beschränkte Benutzung der Geräte und deren Lokalität im UMTS-Kontext einige Angriffe aufgrund von Seiteneffekten ausschließen.

4.4 Komplexität

Aufgrund der sehr guten Eigenschaften zur Parallelisierbarkeit einzelner Komponenten von KASUMI, wie z.B. der Substitutionen S7 und S9, und der einfachen Operationen, ist die Komplexität sehr gering gehalten. Auch wurde auf überflüssige, komplexitätssteigernde Erweiterungen, wie im vorigen Abschnitt beschrieben, verzichtet. Die Verschlüsselungsgeschwindigkeit liegt über 2MBit/s und die Schaltkreisgröße unter 3000 Gattern.

Verschiedene Geschwindigkeitstests, die in [BP03] dokumentiert sind, zeigen, dass KASUMI durchschnittlich – je nach Implementierung – nahezu die Geschwindigkeit von DES erreicht und damit sicherlich nicht zu den schnellsten Blockchiffren zählt aber trotzdem den Ansprüchen entspricht.

[4] Das Hamming-Gewicht ist die Anzahl der von Null verschiedenen Stellen in einer Sequenz. Hier sind es die „1“-Bit in einem Byte.

Primitive Name	**Key Size**		
		PIII/Linux Coppermine gcc 2.95.2	**PIII/Linux Coppermine gcc 3.1.1**
Idea	128	61/61/878	57/56/450
Khazad	128	45/46/834	39/40/765
Khazad-tweak	128	49/49/841	39/40/766
Misty1	128	47/48/257	47/47/209
Safer++	128	152/196/1835	178/169/1504
Cs-cipher	128	163/150/2851	184/191/2691
Hierocrypt-L1	128	54/57/36K	44/47/34K
Nimbus	128	34/34/12K	38/41/13K
Nush	128	95/66/1438	48/46/1422
CAST-128	128	37/38/1021	31/30/933
DES	56	60/60/922	59/59/886
Triple-DES	168	161/160/2753	155/156/2656
Kasumi	128	90/88/410	74/75/339
RC5 (12 Rounds)	64	24/24/1508	20/19/1235
Skipjack	80	160/149/18	115/121/16

Abbildung 4.2: Geschwindigkeiten verschiedener Blockchiffren mit zwei Compilerversionen. Die Ergebnisse sind in der Form Verschlüsselungs-/Entschlüsselungs-/Schlüsselerzeugungs-Zeiten, gemessen in Zyklen/Byte bzw. Zyklen/Schlüsselerzeugung. (Quelle: [BP03])

5. Implementierung in web*Mathematica*

In diesem Kapitel werde ich die Implementierungsphasen der besprochenen Algorithmen in *Mathematica* und web*Mathematica* beschreiben.

5.1 *Mathematica* und web*Mathematica*

Mathematica ist eine Software-Umgebung vor allem für Ingenieure und Mathematiker. Sie integriert einen Kernel zur numerischen und symbolischen Berechnung komplexer mathematischer Probleme, ein grafisches Ausgabesystem, eine eigene interpretierte Programmiersprache, eine Dokumentationsumgebung und Schnittstellen zu anderen Systemen bzw. Programmiersprachen, wie z.B. Java. Mehr zu *Mathematica* kann entweder auf der Wolfram Research Homepage [WR03] oder im *Mathematica* Buch [Wol96] nachgelesen werden.

Für die Implementierung von Algorithmen eignet sich *Mathematica* als Programmiersprache sehr gut, da die Sprache zum einen relativ einfach und intuitiv aufgebaut ist und zum anderen eine umfassende Funktionsbibliothek bereits vorhanden ist. Die geschriebenen Programme und Funktionen sind sehr gut nachvollziehbar und bei der Entwicklung kommt einem die Integration in eine voll interpretierte Umgebung sehr entgegen. Man programmiert also schon im Probieren. Eine für meinen Zweck sehr entgegenkommende Eigenschaft von *Mathematica* war außerdem die starke Ausrichtung der Funktionsbibliothek auf Listen bzw. Arrays und Matrizen. Zu beachten ist, dass die geschriebenen Funktionen und Programme in der Regel deutlich langsamer sind als Implementierungen in Programmiersprachen wie z.B. C oder PASCAL.

web*Mathematica* ist sozusagen die Internetversion von *Mathematica* . Mit web*Mathematica* kann man interaktive Berechnungen und Visualisierungen in eine Webseite integrieren. Dynamische Inhalte, die mit *Mathematica* berechnet wurden, können über das Internet in einem üblichen Browser dargestellt werden. Die eigentliche Verwendung von *Mathematica* kann vor dem Benutzer völlig verschleiert werden. Mit web*Mathematica* kann man auf sämtliche Funktionen und Eigenschaften von *Mathematica* zurückgreifen. Auch Bilder, Plots, Matrizen, die sog. *Mathematica* Notebooks und MathML-Code kann ausgegeben werden.

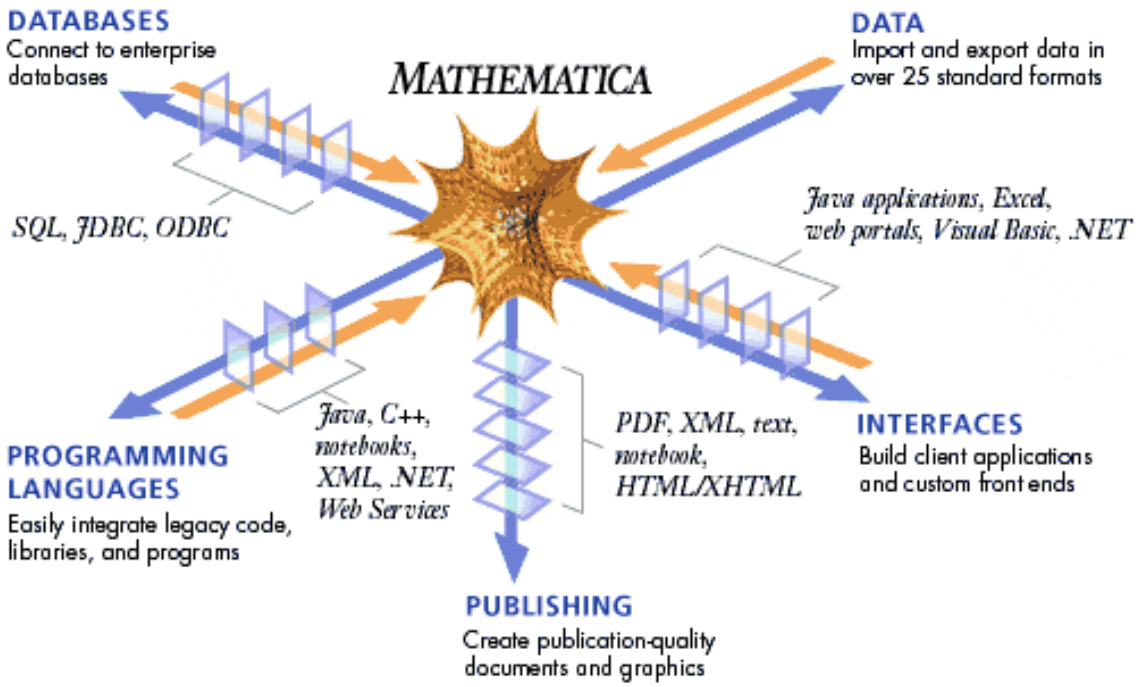

Abbildung 5.1: *Mathematica* Entwicklungsumgebung (Quelle: [WR03])

Die Technologie von web*Mathematica* arbeitet mit den sog. *Mathematica* Server Pages (MSPs), die wiederum auf der Java-Technologie, den sog. Servlets basieren. Java Servlets sind, im Unterschied zu Applets, serverseitige Anwendungen, die die Funktionsweise eines Webservers erweitern können. Servlets bleiben nach dem ersten Aufruf geladen und laufen später in Threads ab und nicht in Prozessen, wodurch das zeitaufwändige Laden des Bytecode-Interpreters, der *Java Virtual Machine* entfällt. Mit dem sog. Servlet-Alias können Dateien auf dem Server, die von einem Webclient aufgerufen werden und bestimmte Endungen haben (z.B. .msp) oder in bestimmten Verzeichnissen liegen, direkt einem bestimmten Servlet zur Ausführung zugeordnet werden [Glo03]. Üblicherweise laufen Servlets in sog. *Servlet Containern* ab, die an den Webserver gekoppelt sind. *Servlet Container* sind z.B. Tomcat oder JRun, die beide auch als eigenständige Webserver laufen können. Abbildung 5.2 veranschaulicht die Zusammenarbeit der einzelnen Komponenten.

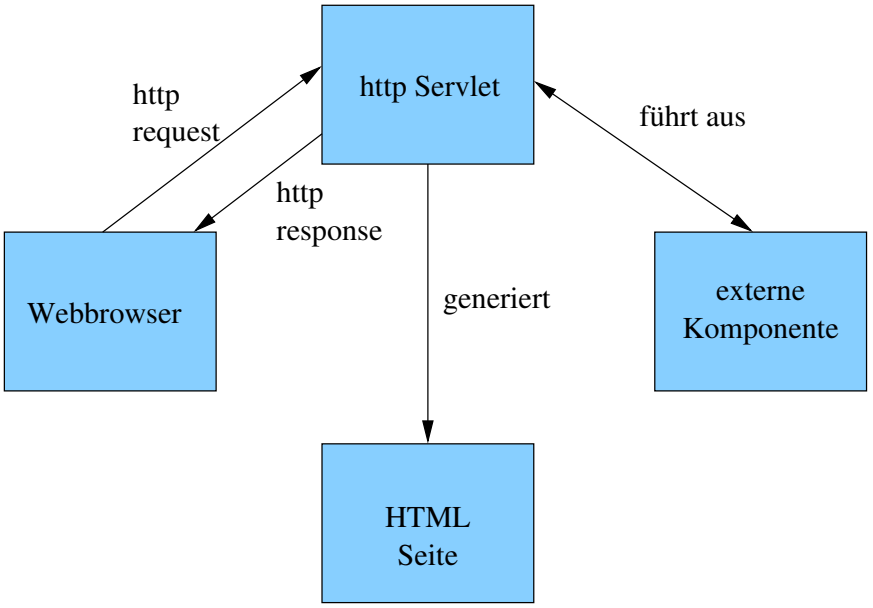

Abbildung 5.2: *Mathematica* Java Servlets Architektur (Quelle: Eigene Darstellung)

Mit der von web*Mathematica* verwendeten MSP Technologie können *Mathematica* Kommandos in eine HTML Seite innherhalb bestimmter Tags integriert werden. Diese MSP Scripts werden dann bei einem Client-Request von einem Servlet auf dem Webserver verarbeitet; der *Mathematica* Kernel wird gestartet, die enthaltenen *Mathematica* Befehle werden ausgeführt und die berechnete *Mathematica* Ausgabe wird als HTML Seite generiert an den Webserver und dann an den Client zurückgeliefert. Um Seiten mit web*Mathematica* zu erstellen, muss man also keine Java Kenntnisse anwenden, sondern man programmiert in HTML und fügt an den entsprechenden Stellen die sog. Mathlet-Tags (<%Mathlet *Ausdruck* %>) ein, deren Inhalt als *Mathematica* Ausdrücke vom Kernel verarbeitet werden. Beispiele und ausführlichere Erläuterungen zu web*Mathematica* finden Sie entweder auf [WR03] oder in der web*Mathematica* Dokumentation [WJ01].

Das folgende Beispiel enthält zwei Eingabefelder innerhalb eines HTML-Formulars namens *expr* und *num*, die als Variablen der *Mathematica* Funktion Expand übergeben werden. Durch Klicken des submit-Buttons wird die Seite mit den Inhalten des Formulars an den Server übertragen und vom Servlet verarbeitet. Ein freier *Mathematica* Kernel wird gestartet, die Berechnung wird durchgeführt und eine HTML-Seite wird generiert, mit dem Ergebnis der Funktion Expand[ ] an der Stelle des Mathlet-Tags. Die Funktion MSPBlock[ ] ist eine *AddOn* Funktion des web*Mathematica* Pakets und überprüft die Inhalte der Variablen, bevor Sie an den Kernel weitergegeben werden. So wird verhindert, dass sicherheitskritische Kommandos vom *Mathematica* Kernel über den Client ausgeführt werden. Variablen eines Formulars werden immer mit dem entsprechenden Namen und zwei vorangestellten Dollar-Zeichen ($$) innerhalb eines Mathlet-Tags verwendet. Konform dazu sollten alle definierten Variablen in web*Mathematica* dieses Format haben.

```
<!--
    webMathematica source code (c) 1999-2001,
    Wolfram Research, Inc. All rights reserved.
-->

<html>
<head>
<title>Expanding Polynomials</title>
</head>
<h1>Expanding Polynomials</h1>

<form action="Expand" method="post">

Enter a polynomial (eg x+y):
<input type="text" name="expr" align="left" size="10" >

Enter a positive integer (eg 4):
<input type="text" name="num" align="left" size="3" >
<br>

<\%Mathlet
        MSPBlock[ {$$expr, $$num},
                  Expand[$$expr^$$num]] %>

<br>
<input type="submit" name="submitButton" value="Evaluate">
</form>

</body>
</html>
```

Bei web*Mathematica* ist zu beachten, dass man in der Regel nur eine *Mathematica* Session zur Verfügung hat, um die gewünschten Berechungen durchzuführen. Da man üblicherweise nur eine Lizenz für *Mathematica* auf dem Server hat, hat man auch nur einen Kernel zur Verfügung, den sich alle Clients teilen müssen. Die Variablen können zwar eine bestimmte festgelegte Zeit in der *Mathematica* Session gespeichert werden, ein anderer Client kann diese aber überschreiben. Allerdings kann man das System auf dem Server so konfigurieren, dass man sich eine private Session in dem sog. Session-Pool anlegt und sich für diese einen Kernel reserviert. Bei nur einer Lizenz ist somit aber kein Kernel mehr frei für weitere Sessions. Das Speichern von Werten ist ein generelles Problem, vor allem weil HTTP ein zustandsloses Protokoll ist, es also nicht möglich ist, sich über das Protokoll Zustände zu „merken“. Diese Eigenschaft muss man mit entsprechenden Methoden umgehen und sich z.B. mit Cookies oder mehreren, sich gegenseitig aufrufenden Seiten Abhilfe verschaffen.[1]

5.2 KASUMI in *Mathematica*

Die Vorüberlegungen zur Implementierung des KASUMI Algorithmus in *Mathematica* beschäftigten sich hauptsächlich mit der Art der Eingabe und Ausgabe bzw. deren

[1] Während des Verfassens dieser Arbeit ist eine neue Version von web*Mathematica* erschienen, die neue Funktionen bereitstellt, um Daten über HTTP Sessions in sog. Session-Variablen zu speichern.

Formate, um möglichst viele bereits vorhandene Funktionen verwenden zu können. So wurde darauf geachtet, dass sämtliche Binärzahlen als *Mathematica* Listen dargestellt wurden, da dies der verbreiteteste Datentyp in *Mathematica* ist und viele Funktionen darauf arbeiten. Außerdem sollte es bei der Entwicklung nicht darauf ankommen, möglichst effizient zu programmieren, sondern der Code sollte gut lesbar und verständlich sein und somit die Zusammenfassung und Optimierung des Codes auf ein entsprechendes Limit begrenzt sein. Daraus ergab sich auch der Anspruch, möglichst modular zu arbeiten und in sich geschlossene Komponenten auszugliedern in eigene Funktionen oder Module.

Der gesamte Algorithmus teilt sich als *Mathematica* Implementierung in folgende Module auf:

- **Kasumi[in_, K_]** realisiert die KASUMI Blockchiffre. Eingabeblock in_ ist eine Liste mit 64 binären Elementen und der 128 Bit Schlüssel K_ ebenfalls als Liste binärer Elemente. Ausgegeben wird der Chiffretext als eine Liste im selben Format wie der Eingabeblock. (Z.B. {1,0,1,...,0})

- **Kasumi[in_, K_, roundout_]** realisiert ebenfalls die KASUMI Blockchiffre. Es kann noch ein zusätzlicher Parameter roundout_ (0 oder 1) angegeben werden, der bestimmt, ob die Rundenergebnisse mit ausgegeben werden. Die Ausgabe ist eine Liste von Listen mit 8 mal 64 binären Elementen. (Z.B. {{1,0,1,...,0},{0,1,1,...,1}})

- **FL[in_, round_]** realisiert die KASUMI-Unterfunktion FL. Der 32 Bit Eingabeblock als Liste in_ wird mit dem globalen 32 Bit Teilschlüssel KL[round_] verknüpft. Der Parameter round_ stellt dabei die aktuelle Rundenzahl dar (1 bis 8). Als Ausgabe erhält man einen 32 Bit Ausgabeblock als Liste.

- **FO[in_, round_]** realisiert die KASUMI-Unterfunktion FO. Eingabeblock in_ ist eine Liste mit 32 binären Elementen und die Rundenzahl round_. Intern werden die zwei globalen Schlüsselblöcke KO_ und KI_ je als Liste mit 48 binären Elementen verwendet. Ausgegeben wird eine Liste im selben Format wie der Eingabeblock.

- **FI[in_, round_, roundFO_]** realisiert die KASUMI-Unterfunktion FI. Eingabeblock in_ ist eine Liste mit 16 binären Elementen, die KASUMI-Rundennummer round_ und die Rundennummer von FO roundFO_ (1 bis 3). Der intern verwendete Teilschlüsselblock KI ist als globale Liste durch die KeySchedule[ ]-Funktion vorhanden. Ausgegeben wird eine Liste im selben Format wie die Eingabe.

- **KeySchedule[K_]** realisiert die KASUMI KeySchedule zur Teilschlüsselgenerierung aus dem 128 Bit Eingabeschlüssel K_. Es werden globale Variablen angelegt, die die entsprechenden Teilschlüssel enthalten: KL (32 Bit), KO (48 Bit), KI (48 Bit).

- **S7[xlist_]** macht einen Table-Lookup in der KASUMI S-Box S7 und liefert den Wert an der Stelle xlist_ zurück. Als Eingabe wird eine binäre Liste mit sieben Elementen erwartet; ebenso groß ist die Ausgabe.

- **S9[xlist_]** macht einen Table-Lookup in der KASUMI S-Box S9 und liefert den Wert an der Stelle xlist_ zurück. Als Eingabe wird eine binäre Liste mit neun Elementen erwartet; ebenso groß ist die Ausgabe.

Abbildung 5.3 zeigt die verschiedenen Aufrufebenen der einzelnen Module zueinander.

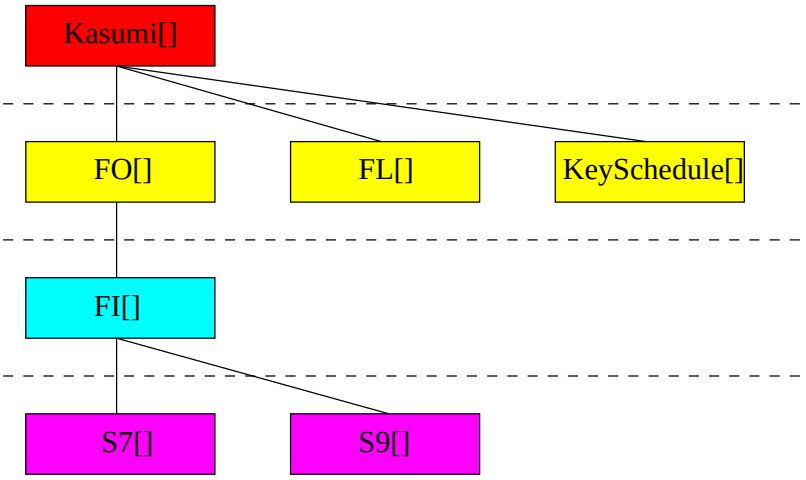

Abbildung 5.3: Modul-Hierarchie der Implementierung von KASUMI (Quelle: Eigene Darstellung)

Außerdem habe ich die Verschlüsselungsfunktion f8[ ] und die Integritätsfunktion f9[ ] jeweils in verschiedenen Versionen implementiert mit unterschiedlicher Parameterzahl.

Alle Module wurden von mir in einer Paket-Datei *kasumi_pack.m* zusammengefasst. Sämtliche zugehörigen *::usage* Kommentare wurden in der Datei *kasumi_use.m* zusammengefasst. Den Inhalt dieser Dateien finden Sie im Anhang D.1. Im Folgenden werden die einzelnen Module genauer beschrieben.

5.2.1 Kasumi[in_, K_]

Zu Anfang des Moduls werden die Benutzereingaben auf korrekte Länge und Inhalt überprüft. Die Länge wird mit der *Mathematica* Funktion Length[ ][2] überprüft und der Inhalt mit der *Mathematica* Funktion Select[ ], die jedes Element einer Liste prüft; hier wird geprüft, ob ein Element ungleich 0 und 1 ist. Ist diese Aussage bei einem Element war, so wird dies in die resultierende Liste eingetragen. Bei korrektem Inhalt muss die Länge dieser Liste also 0 bleiben, anderenfalls existiert ein unzulässiges Zeichen in der Eingabeliste. Sollte Inhalt oder Länge der Eingabeparameter nicht richtig sein, so wird eine definierte *Mathematica* Message in den Message-Ausgabekanal geschrieben.

Bei syntaktisch richtigen Eingabewerten, werden die Teilschlüssel durch das Aufrufen der Funktion KeySchedule[ ] mit dem Eingabeschlüssel K erzeugt. Die Teilschlüssel KL, KO und KI stehen nun global als Variablen im Kernel zur Verfügung.

[2]Eine ausführliche Dokumentation aller *Mathematica* Funktionen finden Sie in [Wol96]

Anschließend wird der Klartext-Eingabeblock mit der von mir geschriebenen Hilfsfunktion SplitBlock[ ] (siehe Anhang D.3) in zwei gleiche Hälften geteilt, die jeweils in einer Variable gespeichert werden. Diese beiden Listen durchlaufen nun acht mal eine Schleife, in der die Berechnungen einer KASUMI-Runde durchgeführt werden wie im vorigen Kapitel beschrieben. Die intern definierte Rundenfunktion f[ ] prüft anhand der Rundenzahl, ob es sich um eine gerade oder ungerade Runde handelt und führt dann in der entsprechenden Reihenfolge die jeweiligen Unterfunktionsaufrufe mit der aktuellen linken Hälfte aus. Das Ergebnis wird mit der rechten Hälfte XOR-verknüpft und damit die neue linke Hälfte für die nächste Runde erzeugt. Zur Ausgabe werden nach acht Runden die zwei Listen-Hälften wieder mit der *Mathematica* Funktion Join[ ] zu einer Liste zusammengefügt.

5.2.2 Kasumi[in_, K_, roundout_]

Die Kasumi[ ] Version mit drei Parametern kann zusätzlich die Rundenergebnisse ausgeben. Vor den Berechnungen und nach der Syntaxprüfung wird die Ausgabeliste als eine Liste mit acht Listen als Elemente formatiert. Dies ist deshalb notwendig, um später direkt über den Index auf diese Position zu schreiben.

Die Schleife läuft hier etwas anders ab; nach der Berechnung der neuen Hälften, wird geprüft ob der dritte Parameter roundout_ gleich Null ist. Ist das der Fall, so wird geprüft ob wir uns in der achten Runde befinden und wenn auch das zutrifft, werden die zusammengefügten Hälften als Liste in eine Liste eingetragen. Dies geschieht deshalb, da das Ausgabeformat konform zu dem mit allen acht Rundenergebnissen sein soll und dieses ist ein Liste von Listen.

Ist der Parameter roundout_ nicht Null, so wird jedes Rundenergebnis als Liste in die Ausgabeliste an die der Rundenzahl entsprechende Stelle eingetragen. Dabei werden die Ausgabehälften der ungeraden Runden vor dem zusammenfügen vertauscht, um bei der Berechnung der Differenz zu anderen Ergebnissen oder zum Klartext, keine Verfälschungen zu erhalten. Diese Eigenschaft kann jedoch leicht geändert werden, indem die zwei Variablen *inright* und *inleft* in dem entsprechenden Join[ ]-Aufruf gegeneinander ersetzt werden.

5.2.3 FL[in_, round_]

Die Unterfunktion FL[ ] teilt zunächst die Eingabeliste in zwei 16 Bit Hälften auf. Die Ausgabe berechnet sich wie im vorigen Kapitel beschrieben, dabei werden für die bitweise UND-, ODER- bzw. XOR-Verknüpfung die *Mathematica* Funktionen BitAnd[ ], BitOr[ ] bzw. BitXor[ ] verwendet und die Leftshift Operation führt die *Mathematica* Funktion RotateLeft[ ] aus. Der Aufruf *KL[round, 1]* gibt das Listenelement der globalen Teilschlüssel-Variablen *KL* an der ersten Stelle, der *round*-ten Unterliste. Dieses Element ist selbst wiederum eine Liste mit 16 binären Elementen. Als Ausgabe werden die zwei neu berechneten Hälften wieder zusammengefügt.

5.2.4 FO[in_, round_]

Auch hier werden die gleich große Hälften aus dem 32 Bit Eingabeblock erzeugt. Die Hälften durchlaufen dann dreimal eine Schleife, die die Rundenfunktion von *FO* realisiert. Hierfür ist intern die Funktion FOround[ ] definiert, die pro Runde einmal aufgerufen wird. FOround[ ] arbeitet mit den globalen Teilschlüssel-Listen

KO und *KI* und den beiden Texthälften. Sie ruft die Unterfunktion FI[ ] auf, deren Ergebnis mit der aktuellen rechten Texthälfte XOR-verknüpft, die Ausgabe von FOround[ ] ist. Auch hier ergeben die zusammengefügten Hälften der letzten Runde die Ausgabeliste.

5.2.5 FI[in_, round_, roundFO_]

Die Unterfunktion FI[ ] teilt die Eingabeliste in_ mit SplitBlock[ ] in eine Liste mit den neun linken Elementen und eine Liste mit den sieben rechten Elementen. Der entsprechende Rundenschlüssel KI wird in eine Liste mit den sieben linken Elementen und eine mit den neun rechten Elementen aufgeteilt.

In der ersten Runde ergibt sich der neue rechte Block aus einer XOR-Verknüpfung des Resultats der S-Box Funktion S9[ ] und des mit Nullen auf neun Stellen aufgefüllten rechten Blocks durch ZeroExtend[ ]. Der neue linke Block ergibt sich aus dem alten rechten. In ähnlicher Weise, mit leichten Abweichungen, berechnen sich die anderen Zwischenergebnisse konform nach dem KASUMI-Standard. Die Ausgabe ergibt sich wieder durch den Join[ ]-Aufruf der beiden letzten Blöcke.

5.2.6 KeySchedule[K_]

Die KeySchedule[ ] Funktion definiert zunächst die interne Liste der Konstanten C_j (vgl. Tabelle C.2). Mit dem Aufruf der *Mathematica* Funktion Partition[ ] wird der eingegebene Schlüssel K_ in acht Listen mit je 16 Elementen aufgeteilt, die in K_j gespeichert werden. Die Schlüsselvariablen, die es später zu belegen gilt, werden als leere Listen formatiert. Die Listen *KL, KO* und *KI* sind dreifach verschachtelt (Kasumi-Runde, FX-Runde, Bit). Die Variable Kc stellt K' dar und enthält somit die acht Schlüsselteile, die mit den Konstanten C_j XOR-verknüpft wurden. Die einzelnen Teilschlüssel ergeben sich dann für die acht Runden in einer Schleife durch die Operationen, wie sie in der KASUMI-Keyschedule (Tabelle C.1) definiert sind.

5.2.7 S7[xlist_]

Die S-Box S7 ist als Liste s7list initialisiert. Beim Aufruf der Funktion S7[ ] mit einer binären 7 Bit Zahl, wird diese Zahl mit Hilfe der *Mathematica* Funktion FromDigits[ ] in eine Dezimalzahl gewandelt. Diese Zahl plus eins ergibt den Index für den Tabellen-Lookup. (Der Index wird deshalb um eins erhöht, da *Mathematica* Listen bei eins anfangen und nicht bei null, wie die Dezimalzahlen der S7.) Die an der resultierenden Stelle vorhandene Dezimalzahl wird anschließend mit der *Mathematica* Funktion IntegerDigits[ ] wieder in eine siebenstellige Binärzahl gewandelt und ausgegeben.

5.2.8 S9[xlist_]

Die Funktion S9[ ] funktioniert genau gleich wie S7[ ], außer dass hier die Liste mit den Zahlen der S-Box S9 initialisiert wird und anstatt mit 7 mit 9 Bit Zahlen gearbeitet wird.

5.2.9 f8[plain_, CK_]

Die UMTS Verschlüsselungsfunktion *f8* wurde von mir in drei verschiedenen Versionen mit jeweils unterschiedlicher Parameterzahl realisiert. Hier werde ich die Grundversion mit zwei Parametern erläutern. Eingaben sind der beliebig große Klartext als binäre Liste und der Schlüsselblock mit 128 Bit ebenfalls als binäre Liste. Nach der Syntaxprüfung wird die Anzahl der zu erzeugenden Schlüsselstromblöcke berechnet. Hierbei wird die *Mathematica* Funktion Ceiling[ ] verwendet, um auf die nächst größere ganze Zahl aufzurunden.

Nach der Initialisierung der restlichen Eingabeparameter der *f8* Funktion (COUNT-C etc.) mit festen, zum Teil zufällig gewählten, Werten, wird die Ausgabeliste des Schlüsselblockstroms KSB mit der Anzahl der Blöcke plus eins formatiert. Deshalb mit einem Feld mehr, da der erste Block der Initialisierungsblock ist, der später nicht mit ausgegeben wird.

Nach einer ersten KASUMI-Anwendung auf die Initialisierungsparameter mit einem veränderten Schlüssel, werden die Schlüsselstromblöcke in einer Schleife berechnet, wie in Abschnitt 3.1.5 beschrieben. Die erzeugten Schlüsselstromblöcke werden letztlich zu einer Liste mit der *Mathematica* Funktion Flatten[ ] zusammengefügt und die Differenz-Elemente zum Klartextblock werden am Ende der Liste entfernt.

5.2.10 f9[msg_, IK_]

Die UMTS Integritätsfunktion *f9* wurde von mir ebenfalls in drei verschiedenen Versionen mit jeweils unterschiedlicher Parameterzahl realisiert. Hier werde ich die Grundversion mit zwei Parametern erläutern. Eingaben sind der beliebig große Klartext als binäre Liste und der Schlüsselblock mit 128 Bit, ebenfalls als binäre Liste.

Der Anfang bis zur Inititialisierung der Parameter läuft sehr ähnlich ab, wie bei f8[ ]. Um die Variable PS mit Nullen zu einem 64 Bit Block aufzufüllen, wird eine Modulo 64 Funktion auf die Länge der verketteten Parameter minus eins angewendet. Zu dem Ergebnis wird eins addiert und anschließend von 64 abgezogen (zur besseren Lesbarkeit dieser Umweg).

Der Ablauf des Algorithmus ist wie in Abschnitt 3.1.4 beschrieben. Die linken 32 Bit des Schlussblocks stellen dann den MAC dar und werden ausgegeben.

5.2.11 Das Crypto Paket

Um alle Funktionen und Hilfsfunktionen in einem Paket zu vereinen, ohne dabei die Übersicht zu verlieren, entschied ich mich dafür zwar ein Crypto Paket in der *Mathematica* Datei crypto.m (siehe Anhang D.3) zu definieren, die Inhalte der Datei aber zum Teil aus anderen Dateien einzulesen. So werden z.B. alle Kasumi Funktionen mit dem Befehl $\ll$ *kasumi_pack.m* eingelesen und alle Kasumi *::usage* Texte mit dem Befehl $\ll$ *kasumi_use.m*. Dies hält die Paket Datei relativ klein und lagert zusammengehörige Funktionen in eigenen Dateien aus. Ebenso habe ich die DES Funktionen in das Paket eingebunden. Den Quellcode der DES Funktionen und deren Beschreibung finden Sie im Anhang D.4 und Anhang D.5.

Allgemeine Hilfsfunktionen, die auch in späteren Entwicklungen nützlich sein könnten wurden intern definiert. Diese sind:

- **BinPlus[bin_, num_, size_]**
- **SplitBlock[nlist_, hleft_, hright_]**
- **BlockRead[b_, nfile_]**
- **Truncate[nlist_, nbit_, pos_]**
- **ZeroExtend[nlist_, nbit_, pos_]**
- **HammingD[list1_, list2_]**
- **ListToString[list_]**
- **BinToHex[in_]**
- **StringToBin[str_]**
- **BinToString[lst_]**

Die Erläuterungen dazu finden Sie jeweils im Anhang D.3.

5.2.12 Analyse-Berechnungen

Unter Verwendung der entwickelten Algorithmen, habe ich verschiedene Matrizen und Graphen zur Analyse berechnet:

Dependence Matrix [Mur99] mit 10000 verschiedenen Klartextpaaren:

```
<< crypto.m
<< testdata.m
<< Statistics`DescriptiveStatistics`;

depmat = Table[0, {8}, {64}, {64}];
plainlista = {}; plainlistb = {};
cyphlista = {}; cyphlistb = {};
(** verschiedene Klartexte **)
Timing[
  For[z = 1, z <= 10000, z++,
    plainlista = Table[Random[Integer], {64}];
    (** alle 64 Bit je Klartext einmal an i - ter Stelle verändern **)
    For[i = 1, i <= 64, i++,
      plainlistb = BitXor[plainlista, compl[[i]]];
      cyphlista = Kasumi[plainlista, key, 1];
      cyphlistb = Kasumi[plainlistb, key, 1];
      (** nach Rundenzahl auswerten **)
      For[k = 1, k <= 8, k++,
        (** jedes Bit auf Veränderung an j - ter Stelle prüfen **)
        For[j = 1, j <= 64, j++,
          depmat[[k, i, j]] += BitXor[cyphlista[[k]], cyphlistb[[k]]][[j]]
          ]
        ]
      ]
    ]
  ]
For[r = 1, r <= 8, r++,
  Print["Runde: ", r , "\n", MatrixForm[depmat[[r]]],
    "\n-----------------------------------------------------\n"]
  ]
```

Die Ausgabe-Matrizen mit 64 mal 64 Elementen sind sehr groß und deshalb schlecht darstellbar. Jede Matrix steht für eine Runde und beschreibt in der Reihe i und der Spalte j wie oft sich das Ändern des i-ten Eingabe-Bits zur Änderung von j Ausgabe-Bits führt. An der Matrix der zweiten Runde ist schön zu erkennen, dass KASUMI nach nur zwei Runden noch große Schwächen aufweist. Dies ist aber leicht durch die Feistel-Struktur zu erkennen, da sich für die rechte Ausgabehälfte der zweiten Runde folgendes ergibt: $R_{out} = R_0 \oplus FO_1(FL_1(L_0, KL_1), KO_1, KI)$. Werden die rechten Bit der Eingabe verändert, sind die Bit der linken Eingabehälfte fest und damit $FO_1(FL_1(L_0, KL_1), KO_1, KI_1)$ konstant. Ändern wir also das i-te Bit von R_0 dann ergibt sich: $R'_{out} = R_0 \oplus (0...1..0) \oplus konstant$, wobei die 1 an der i-ten Stelle des zweiten Vektors steht. R'_{out} ist somit gleich R_{out} mit der i-ten Stelle verändert. Zur Veranschaulichung ist hier die fünfte Zeile der Matrix dargestellt (zu beachten ist, dass das erste Bit bei der Berechnung das höchstwertige, also linke ist).

4950	5061	5102	5052	4938	4877	5010	4974
5012	5073	5033	4974	5089	5002	4965	5059
5014	4980	5005	5023	4919	5020	4941	5014
4947	5008	5009	4953	4879	5056	4991	4984
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	10000	0	0	0	0

Die folgende Berechnung zeigt die Erfüllung des Lawineneffekts. 10000 veschiedene Klartextpaare mit jeweils der gleichen Differenz, werden mit KASUMI chiffriert; die einzelnen Chiffretextpaare der Runden werden auf ihre Hamming-Distanz untersucht. Als Graph wird der Mittelwert der Hamming-Distanzen je Runde ausgegeben. Man erkennt, dass bei dieser Version, bei der immer das niederwertigste Bit geändert wird, KASUMI schon nach der dritten Runde den Lawineneffekt erfüllt. Es werden nämlich im Mittel 32 Bit geändert. Diese Berechnung ist mit beliebig anderen Differenzstellen durchführbar.

```
plainlista = {}; plainlistb = {};  cyphlista = {}; cyphlistb = {};
hamlist = Table[{}, {8}];

(** Mit 10000 Klartextpaaren, je 1. Bit Unterschied **)
For[i = 1, i <= 10000, i++,
    plainlista = Table[Random[Integer], {64}];
    plainlistb = BinPlus[plainlista, 1, 64];
    cyphlista = Kasumi[plainlista, key, 1];
    cyphlistb = Kasumi[plainlistb, key, 1];

    (** Hamming-Distanz der Chiffretextpaare pro Runde berechnen **)
    For[j = 1, j <= 8, j++,
      AppendTo[hamlist[[j]], HammingD[cyphlista[[j]], cyphlistb[[j]]]];
      ]
    ];

(** Graphen des Mittelwerts der Hamming-Distanz pro Runde ausgeben **)
PrependTo[Table[{z, N[Mean[hamlist[[z]]]]}, {z, 8} ], {0, 1}];
ListPlot[%, PlotLabel -> "Mittelwert(HammingDist)",  PlotJoined -> True,
```

```
  AxesLabel -> {"Runde", ""},
  PlotRange -> {0, Automatic}
  ]
```

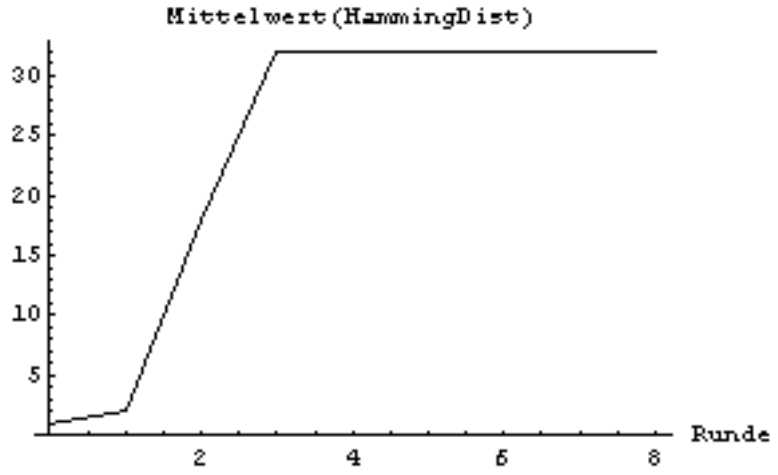

Abbildung 5.4: *Mathematica* Plot der Mittelwerte der Hamming-Distanzen (Quelle: *Mathematica* Plot)

Die nächste Berechnung erzeugt eine Korrelationsmatrix über 10000 verschiedene Klartextpaare mit 64 mal 64 Elementen, wobei das Element der Zeile i und der Spalte j die Korrelation zwischen dem i-ten Bit des ersten Chiffretextes und dem j-ten Bit des zweiten Chiffretextes darstellt. Geändert wird hier immer das niederwertigste Bit.

```
(*** Korrelationsmatrix, 1. Bit = höchstwertiges **)

korrmat = Table[0, {8}, {64}, {64}]; tempmat = Table[0, {8}, {64}, {64}, {3}];
plainlista = {}; plainlistb = {}; cyphlista = {}; cyphlistb = {};
(**verschiedene Klartexte**)
Timing[
  For[z = 1, z <= 10000, z++,
    plainlista = Table[Random[Integer], {64}];
    (** das niederwertigste Bit verändern **)
    plainlistb = BitXor[plainlista, compl[[64]]];
    cyphlista = Kasumi[plainlista, key, 1];
    cyphlistb = Kasumi[plainlistb, key, 1];
    (** nach Rundenzahl k auswerten - Der (i, j)-te Wert der Matrix
        korrmat ist die Korrelation des i-ten Bit des original
        Klartextes mit dem j-ten Bit des modifizierten Klartextes **)
    For[k = 1, k <= 8, k++,
      For[i = 1, i <= 64, i++,
        For[j = 1, j <= 64, j++,

          tempmat[[k, i, j,
                1]] += (cyphlista[[k, i]] - 0.5) * (cyphlistb[[k, j]] - 0.5);
          tempmat[[k, i, j, 2]] += (cyphlista[[k, i]] - 0.5)^2;
          tempmat[[k, i, j, 3]] += (cyphlistb[[k, j]] - 0.5)^2
```

```
          ]
        ]
      ]
    ];
  (** Das Element korrmat[1, 64] ist die Korrlation des ersten Bit
      (hier höchstwertiges!!) des orig. Klartextes mit dem 64. Bit
      (hier niederwertiges!!) des modif. Klartextes. **)
  For[k = 1, k <= 8, k++,
    For[i = 1, i <= 64, i++,
      For[j = 1, j <= 64, j++,
        korrmat[[k, i,
              j]] = (tempmat[[k, i, j, 1]])/(Sqrt[
                tempmat[[k, i, j, 2]] * tempmat[[k, i, j, 3]]])
        ]
      ]
    ]
  ]

For[r = 1, r <= 8, r++,
  Print["Runde: ", r, "\n", MatrixForm[korrmat[[r]]],
    "\n------------------------------------------------------\n"]]
```

Alle Ergebnismatrizen und Berechnungen sind als Notebooks vorhanden und auf der CD zu dieser Arbeit verfügbar.

5.3 Erstellung der web*Mathematica* Seiten

Ziel war es, basierend auf dem verfassten *Mathematica* Crypto-Paket, eine web*Mathematica* Webseite zu erstellen, die Demo-Tools zu den entwickelten Algorithmen bereitstellt. Die Testumgebung war dabei der Server des Rechenzentrums der FH Ravensburg-Weingarten, auf dem Tomcat 4 und web*Mathematica* v1.0 installiert sind. Die URL des Servers ist:

`http://www.webmath.fh-weingarten.de:8080/webMathematica/`

Clientseitig benutzte ich den Netscape Navigator 7.0, für den die Formatierung auch optimiert ist. Die Seiten sollten möglichst übersichtlich und einfach zu bedienen sein. Im Folgenden sind die einzelnen Seiten als .msp Dateien beschrieben.[3]

Ein Link zu den von mir erstellten Seiten findet sich auf der Homepage von Herrn Prof. Dr. Ertel:

`http://www.fh-weingarten.de/~ertel`

5.3.1 index.msp

Jede Datei, die zu der Seitensammlung gehört, besitzt die gleichen CSS (*Cascading Style Sheet*) Definitionen, um allen Dateien ein gleiches Aussehen der Designkriterien wie Schrift, Farbe etc. zu geben. Diese Definition sollte in eventuellen Erweiterungen mit übernommen werden.

[3] Aufgrund der Größe dieses Dokuments, ist der Quelltext der MSP-Scripts nicht abgedruckt und wird separat als Softcopy auf einem Datenmedium abgegeben.

index.msp stellt die Hauptseite dar, die zuerst aufgerufen werden sollte. Jede Seite baut sich auf wie eine normale HTML-Seite in Head und Body (siehe dazu [Mue03]). Der erste Teil des HTML-Bodys ist für die Darstellung des Menüs zuständig. Anschließend erfolgt die Überschrift des Hauptanzeigebereiches und darunter die entsprechend formatierten Texte und Bilder. (Die roten Schriftzeichen auf der Startseite bedeuten übrigens KRYPTOGRAPHIE dargestellt im Freimaurercode [Spe03].)

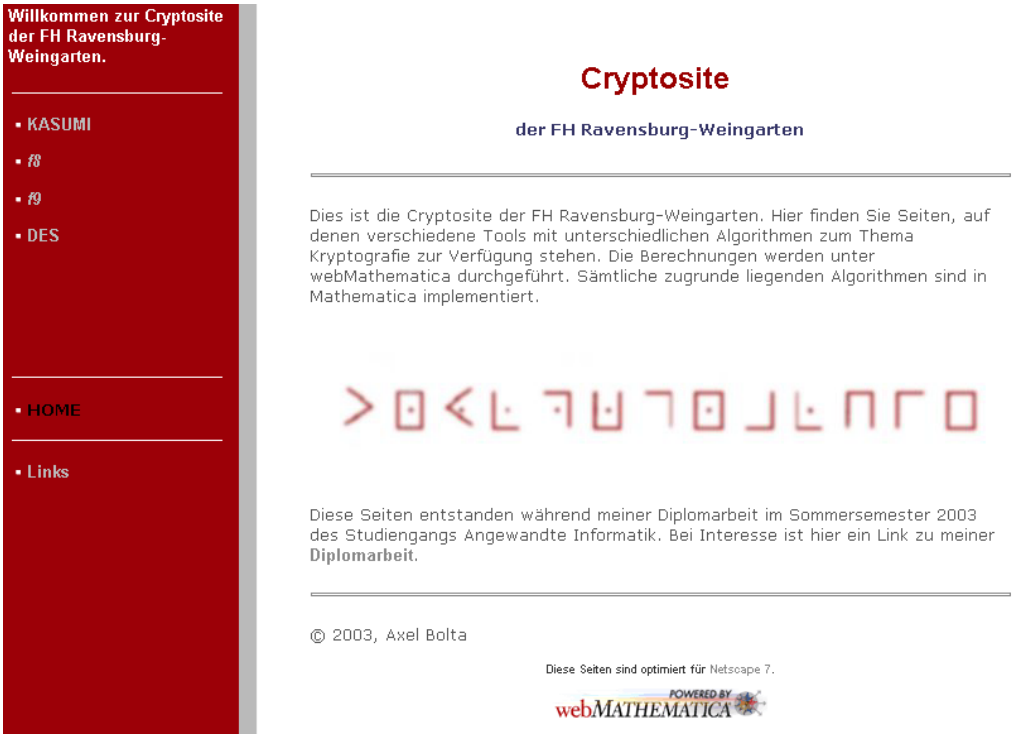

Abbildung 5.5: Screenshot der Seite index.msp

5.3.2 kasumi_index.msp, f8_index.msp, f9_index.msp, des_index.msp

Alle diese Index-Seiten stellen den jeweiligen Algorithmus in Text und Bild kurz vor und öffnen im Menü eine Unterauswahl mit verschiedenen Versionen von Demo-Tools zu den Algorithmen. Die Version V1 ist dabei meistens die einfachste, die am leichtesten anzuwenden bzw. zu verstehen ist und Version V3 die komplizierteste. Über die Links im Menü gelangt man zu den einzelnen Versionen.

5.3.3 kasumiv1.msp, kasumiv1_1.msp

Diese zwei MSP-Scripts stellen die Version 1 des Demo-Tools zu KASUMI dar. kasumiv1.msp ist die Hauptseite, die das Menü aktualisiert und das Eingabeformular bereitstellt.

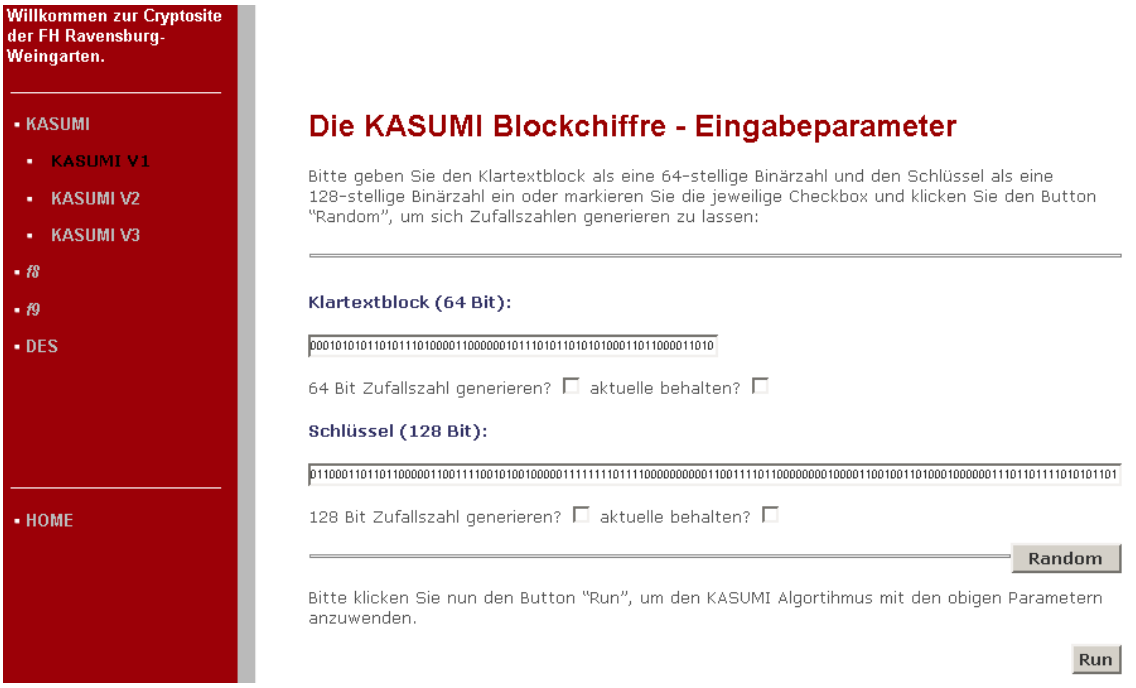

Abbildung 5.6: Screenshot der Seite kasumiv1.msp

Im Head werden einige JavaScript Funktionen definiert. URLescape() wurde von einem MSP Beispiel Script übernommen und sorgt dafür, dass Strings aus Eingabefeldern richtig formatiert weitergegeben werden. Die Funktion run() prüft mit regulären Ausdrücken die Syntax der Eingabewerte und gibt bei falschen Werten ein JavaScript-Alert aus. Bei richtigen Werten wird ein neues Fenster mit window.open() erzeugt und in dieses wird die Datei kasumiv1_1.msp geladen. Beim Aufruf werden dem neuen window-Objekt die eingegebenen Werte als Variablen übergeben. Außerdem werden bestimmte Objekteigenschaften des neuen window-Objekts festgelegt (Größe, Anzeige des Browsermenüs etc.).

Unmittelbar nach dem JavaScript-Teil, steht noch im Head das erste Mathlet-Tag mit der Anweisung $\ll$ *crypto.m*. Hier wird das *Mathematica* Paket Crypto in den Kernel geladen. Die Paket-Datei muss sich hierfür entweder in einem Pfad befinden, der in der *Mathematica* Variablen $Path definiert ist, oder im selben Verzeichnis wie die .msp-Datei in welcher der Aufruf erzeugt wird.

Im Hauptfeld des Bodys wird ein HTML-Formular mit zwei Eingabefeldern angelegt. Hier hat der Benutzer die Möglichkeit den Klartext und den Schlüssel als Binärzahlen in der entsprechenden Größe einzugeben. Alternativ dazu kann er die zwei angelegten Checkboxen check1 und check2 markieren, um anschließend mit dem submit-Button „Random“ Zufallszahlen in die Formularfelder zu laden. Dies geschieht indem die Daten des Formulars an den web*Mathematica* Server übertragen und mit den Mathlet-Tags ausgewertet werden. Die Werte der Variablen namens plain und key sind als Mathlet-Tag definiert. D.h. die Rückgabe des web*Mathematica* Servers nach der Ausführung der Anweisungen im *Mathematica* Kernel liefern die Werte der Variablen. In den Anweisungen wird zunächst mit der web*Mathematica* Funktion MSPValueQ[ ] geprüft ob die Variable check1 bzw. check2 einen Wert hat, also ob die Checkbox markiert ist. Wenn ja, wird eine binäre Zufallszahl der entsprechenden Größe als Liste mit der *Mathematica* Funktion Random[Integer] erzeugt.

Diese Liste wird mit ListToString[ ] in einen String gewandelt und der Variablen des Feldes zugewiesen. Ist check1 bzw. check2 nicht markiert, so wird geprüft ob die Checkbox check11 bzw. check22 markiert ist, um den aktuellen Wert des Eingabefeldes und somit der Variablen zu speichern. Dies ist dehalb nötig, da andernfalls bei jedem Laden der Seite die Informationen verloren gehen würden. Wenn keine der Checkboxen, also check1, check11 bzw. check2, check22 markiert ist, so wird ein leerer String in das Feld geschrieben.

Hat der Benutzer seine Eingaben gewählt, so kann er mit dem Button „Run“, der innerhalb vom Formular definiert ist, die JavaScript-Funktion run() aufrufen und somit das neue Fenster erzeugen, dem die Eingaben mitgeliefert werden.

Abbildung 5.7: Screenshot der Seite kasumiv1_1.msp

Das neue Fenster lädt die Datei kasumiv1_1.msp. Auch hier wird zu Anfang das Crypto Paket in den Kernel geladen und zusätzlich die Zeilenlänge der *Mathematica* Ausgaben auf 150 Zeichen gesetzt, damit die 128-stelligen Bit-Strings nicht umgebrochen dargestellt werden.

Im Body wird zunächst in einem Mathlet-Tag die Berechnung des Algorithmus mit den übernommenen Variablen plain und key als Parameter definiert. Die Variablen, die als String übergeben wurden, werden mit der *Mathematica* Funktion ToExpression[ ] in eine *Mathematica* Anweisung formatiert, um anschließend mit IntegerDigits[ ] eine binäre *Mathematica* Liste darzustellen. Diese Listen werden dann in der *Mathematica* Session neu definierten Variablen $$plainlist und $$keylist zugewiesen, die jetzt nur im laufenden *Mathematica* Kernel existieren. Mit diesen Listen wird die Kasumi[ ] Funktion aufgerufen, die den Chiffretext als *Mathematica* Liste erzeugt.

Anschließend werden in verschiedenen Mathlet-Tags die Werte der Variablen ausgegeben. Dabei müssen die Listen mit ListToString[ ] konvertiert werden.

5.3.4 kasumiv2.msp, kasumiv2_1.msp

Die zweite Version der KASUMI Demo stellt das gleiche Eingabefenster dar, wie V1. Der Unterschied liegt in der Datei kasumiv2_1.msp, die in das neue Fenster geladen wird. Die *Mathematica* Berechnungen zu Beginn des Bodys führen nicht die Standardversion der Kasumi[ ] Funktion aus, sondern die Version mit der Rundenausgabe.

Die Ergebnissse werden in einer Variablen gespeichert und anschließend wird in einer Schleife von jedem Rundenergebnis die Hamming-Distanz zum Klartext berechnet. Die Hamming-Distanzen werden mit der *Mathematica* Funktion AppendTo[ ] in die Liste $$hamlist geschrieben. Schließlich wird der Liste noch ein Element vorangestellt, dass die Hamming-Distanz des Klartext zu sich selbst enthält, also null, um später beim plotten der Liste diesen Wert mit auszugeben.

Nach der Berechnung wird der Klartext und der Chiffretext der letzten Runde ausgegeben, wie schon in V1 beschrieben. Darunter wird eine HTML Tabelle mit neun Zeilen und vier Spalten angelegt. Spalte 1 enthält pro Runde (1 bis 8) eine bildliche Darstellung des Algorithmus je in Form eines GIF-Bildes. Spalte 2 enthält die jeweilige Rundenzahl, Spalte 3 das binäre Rundenergebnis und Spalte 4 die jeweilige Hamming-Distanz.

Unter der Tabelle wird in einem Mathlet-Tag mit der web*Mathematica* Funktion MSPShow[ ][4] ein Plot der Liste der Hamming-Distanzen zur Rundenzahl durchgeführt. Zuvor wird mit Table[ ] jedem der neun Elemente der Liste $$hamlist seine entsprechene Rundenzahl vorangestellt, sodass sich die Elemente der Liste $$pllst in der Form { *Rundenzahl, Hamming-Distanz* } ergeben. Dieses Format ist nötig für die ListPlot[ ] Funktion. Das resultierende GIF-Bild wird an der Stelle des Mathlet-Tags im Browser angezeigt.

5.3.5 kasumiv3.msp, kasumiv3_1.msp

Version 3 der KASUMI Demo realisiert eine parallele Rundendarstellung von einem Klartextpaar. Die Eingabefläche stellt, zusätzlich zu den bereits aus den zwei anderen Versionen bekannten Feldern, zwei Felder dar, die eine zweite Klartext- und Schlüsseleingabe ermöglichen. Diese Felder sind im selben HTML Formular definiert, wie die anderen zwei und übernehmen auch deren Zufallszahlen. Dies wird erreicht, indem der Wert des entsprechenden Feldes durch ein Mathlet-Tag gesetzt wird, dass nur die zugehörige Variable (z.B. $$plaina) des anderen Feldes ausgibt. Die einzelnen Bit dieser zwei zusätzlichen Felder kann bzw. soll der Benutzer verändern, ohne aber die Anzahl der Bit zu ändern. Das Ziel ist, die Basisoperation der differentiellen Kryptanalyse beispielhaft mit einem Klartextpaar und einer bestimmten Differenz der Klartexte durchzuführen. Hierfür werden alle vier Variablen und deren aktuelle Werte an das neue Fenster und das MSP Script kasumiv3_1.msp übergeben.

[4] MSPShow[ ] wird zum anzeigen einer grafischen *Mathematica* Ausgabe in einer HTML Seite verwendet.

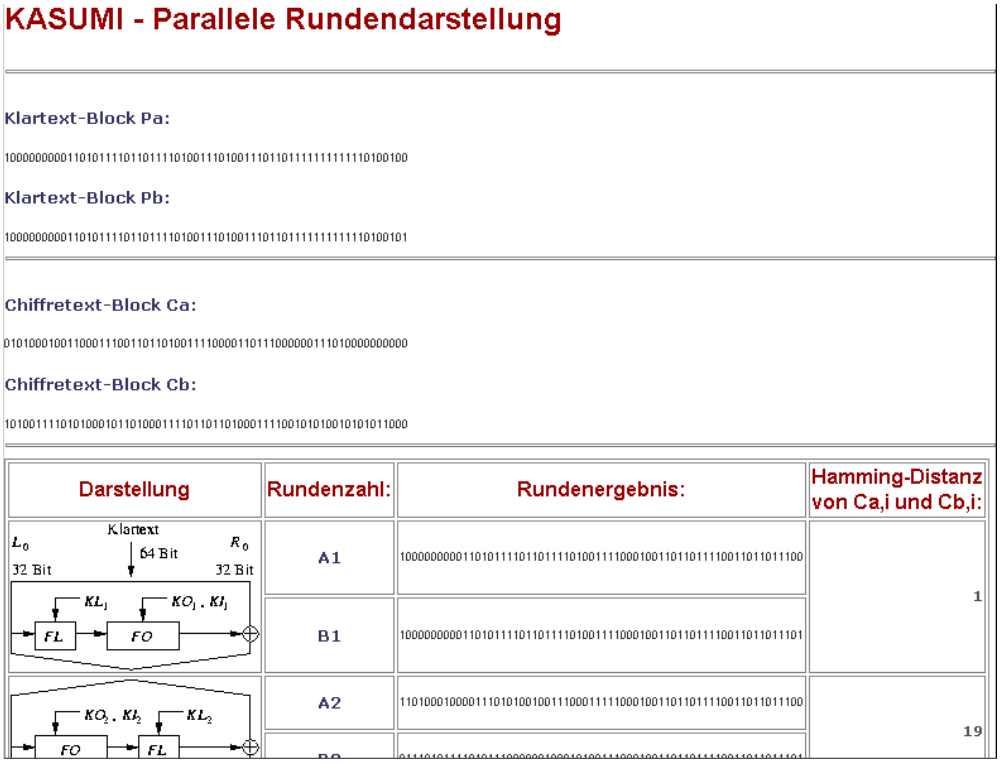

Abbildung 5.8: Screenshot der Seite kasumiv3_1.msp

In dem neuen Fenster wird dann die parallele Rundendarstellung angezeigt. Zu Beginn des Bodys werden dafür folgende Berechnungen durchgeführt: Auf beide Klartexte wird mit dem zugehörigen Schlüssel die erweiterte Kasumi[ ] Funktion mit der Rundenausgabe angewendet. Die Hamming-Distanz wird jetzt zwischen den jeweiligen Chiffretexten der einzelnen Runden berechnet und in eine Liste $$hamlist geschrieben. Der folgende Teil erfolgt wie bei Version 2, mit dem Unterschied, dass pro Rundenzeile beide Zwischenergebnisse ausgegeben werden. Anhand des Plots kann der Benutzer schön den Lawineneffekt erkennen und z.B. sehen, welche Auswirkungen die Veränderung einzelner Bits hat; z.B. Veränderung der linken oder rechten Hälfte des Klartextblocks.

5.3.6 f8v1.msp, f8v1_1.msp

Die Version 1 der Verschlüsselungsfunktion *f8*-Demo, stellt zunächst zwei Eingabefelder dar, wobei es sich bei einem um eine sog. Textarea handelt. Wir können mit einer Textarea, bezüglich der Verwendung als *Mathematica* Variable, gleich umgehen, wie mit Input Feldern. Der Benutzer hat die Möglichkeit, den Kartext als beliebige Zeichenfolge einzugeben. Den Schlüssel kann er sich erzeugen lassen, oder selbst binär eingeben. Mit dem Mathlet in der Textarea und der web*Mathematica* Funktion MSPSetDefault[ ] wird der Wert der Textarea-Variablen $$plain auf einen Standardwert gesetzt, falls die Variable leer ist; z.B. beim ersten Laden der Seite. Mit der folgenden Zeile wird der aktuelle Wert der Variablen $$plain ausgegeben und in der Textarea angezeigt.

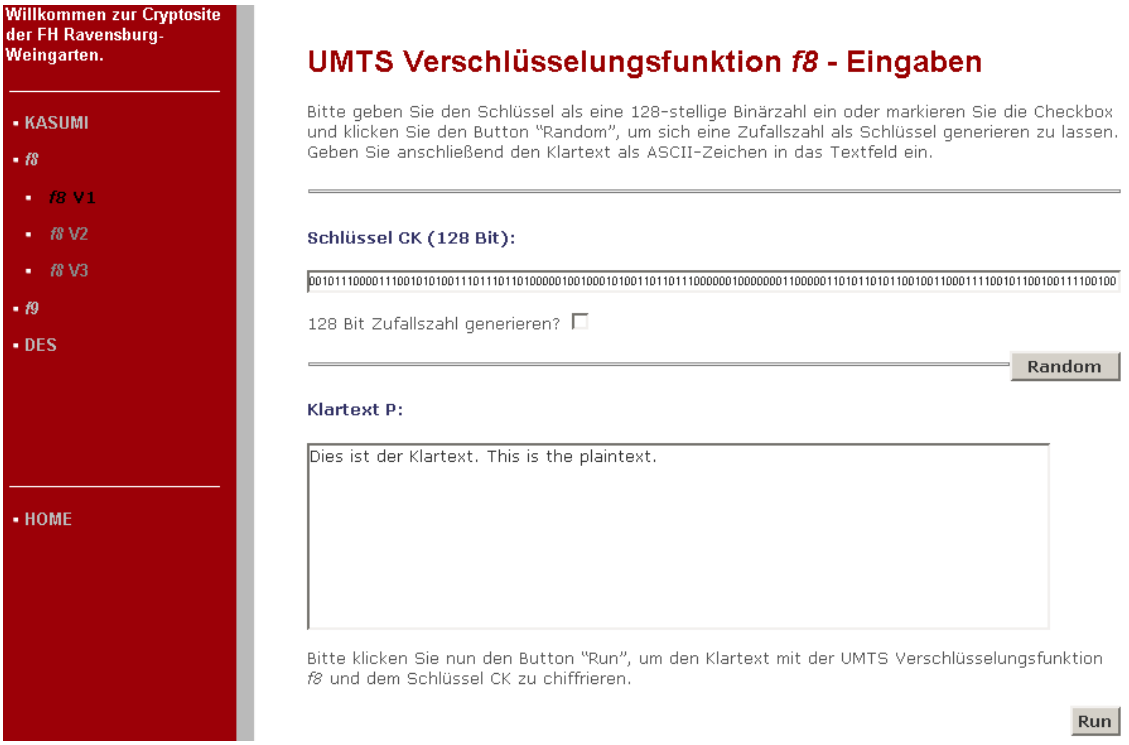

Abbildung 5.9: Screenshot der Seite f8v1.msp

Durch klicken des Buttons „Run“, wird auch hier über ein JavaScript ein neues Fenster erzeugt, dem die beiden Variablen übergeben werden. In dem MSP Script f8v1_1.msp, das dabei geladen wird, wird in einem Mathlet der eingegebene Klartext mit der Hilfsfunktion ListToBin[ ] aus einem String byteweise in eine binäre Liste gewandelt. Auch der String aus Nullen und Einsen des Schlüsselblocks wird in eine Liste zerlegt. Es folgt die Erzeugung des Schlüsselstroms mit f8[ ] und den Parametern und eine anschließende XOR-Verknüpfung mit dem Klartext. Zusätzlich wird der erzeugte Chiffretext mit dem selben Schlüsselstrom ebenfalls XOR-verknüpft und man erhält wieder den ursprünglichen Klartext. Mit der Hilfsfunktion BinToString[ ] werden die binären Listen wieder byteweise in Charactercode-Strings konvertiert und ausgegeben. Außerdem wird über ein, in den Text eingebettetes, Mathlet die aktuelle Zeichenkodierung des *Mathematica* Kernels ausgegeben.

Abbildung 5.10: Screenshot der Seite f8v1_1.msp

Leider ist es aufgrund der vielen verschiedenen erzeugten Zeichen im Chiffretext, die auch Steuerzeichen enthalten, und den unterschiedlichen Zeichenkodierungen der Browser und des *Mathematica* Kernels nicht möglich, den erzeugten Chiffretext aus dem Ergebnisfenster manuell zu kopieren und im Hauptfenster wieder als Klartext einzusetzen. Mit dem identischen Schlüssel würde der berechnete Chiffretext dem ursprünglichen Klartext entsprechen. Um dies zu erreichen, müsste man eine Konvertierungsfunktion in *Mathematica* schreiben, die z.B. nur den ASCII Zeichensatz verwendet wodurch mit ISO Zeichensätzen homogen gearbeitet werden könnte.

5.3.7 f8v2.msp, f8v2_1.msp

Die zweite Version der *f8*-Darstellung arbeitet ähnlich wie die erste. Der Unterschied liegt hier in der zusätzlichen Eingabemöglichkeit des Initialisierungsvektors IV der Funktion.

Bei der Ergebnisberechnung wird daher die Version 2 der f8[ ] Funktion aufgerufen.

5.3.8 f8v3.msp, f8v3_1.msp

Die dritte Version stellt die volle Anzahl der Parameter bereit zur Eingabe. Dementsprechend wird auch die Version 3 von f8[ ] bei der Berechnung verwendet. Der Rest läuft gleich ab, wie bei den anderen beiden Versionen.

5.3.9 f9vX.msp, f9vX_Y.msp

Die Demo-Seiten der Integritätsfunktion *f9* funktionieren an sich genau gleich, wie die von *f8* mit zum Teil unterschiedlichen Parametern.

Im Ergebnisfenster wird dann die entsprechende Version der f9[ ] Funktion aufgerufen und der erzeugte MAC wird als Binärzahl und als Hexadezimalzahl ausgegeben. Mit der web*Mathematica* Funktion MSPFormat[ ] wird das *Mathematica* BaseForm-Format als GIF-Bild exportiert, um es in HTML darstellen zu können.

6. Zusammenfassung und Ausblick

Abschließend lässt sich sagen, dass die in UMTS verwendeten Sicherheitsmechanismen auf jeden Fall der Zeit entsprechen und dass das UMTS-Netz, mit allen spezifizierten und vorgesehenen Sicherheitskonzepten implementiert, als eines der sichersten Funknetze betrachtet werden kann. Gerade im Vergleich zu WLAN (*Wireless LAN*), das bei IP-Netzen als scharfer Konkurent für UMTS gilt, sind schon einige elementare Sicherheitsmängel bekannt und das WLAN-Netz mit dem verwendeten WEP (*Wireless Eqivalent Privacy Protocol*) wird von Experten nicht als sicher gewertet. Die WLAN-Netze sind zwar relativ einfach aufzubauen und zu betreiben – auch für den privaten Nutzer – aber gerade dies führt oft dazu, dass selbst die geringsten Sicherheitsvorkehrungen vernachlässigt werden und damit das Netz potentiellen Angreifern nahezu vollständig offengelegt wird. Vor allem für Firmen, deren Netzwerk-Administratoren evtl. zu wenig auf Sicherheit sensibilisiert sind, kann die Integration eines WLANs in das Firmennetz zu erheblichen Risiken führen [Kla02].

Wie in den meisten Fällen ist auch der Mensch das größte Sicherheitsrisiko, wenn es um die Sicherheit eines UMTS-Netzes geht. Durch fahrlässigen Umgang mit Passwörtern, die einem Zugang zur Administrationsebene des Netzes verschaffen oder nicht adequates Sichern der Zugangsschnittstellen für Abhörzwecke, können leicht alle technisch implementierten Sicherheitsvorkehrungen umgangen werden.

Die in UMTS verwendeten Algorithmen und Mechanismen zur Datensicherheit wurden von unhabhängigene Kryptographieexperten entwickelt und evaluiert. Kein sinnvoller, praktisch durchführbarer Ansatz eines Angriffes wurde aufgezeigt.

Selbstverständlich hört die Entwicklung nie auf und während 3G gerade in der Anlaufphase ist, finden bereits Entwicklungen für 4G statt und dazu auch für andere Sicherheitskozepte. Ein Ausblick auf Mobilfunksysteme der vierten Generation (4G) lässt erwarten, dass es zukünftig nur noch paketvermittelten und keinen leitungsvermittelten Datenverkehr mehr geben wird. Ein Vorteil davon sind Bandbreiten, die im Bereich von 20 bis 100 Mbit/s diskutiert werden [Pea03]. Durch die reine Paketvermittlung, wird 4G mehr mit einem IP-Netzwerk zu vergleichen sein und wesentlich effizienter als 3G sein. Die Vision von 4G ist die Integration der verschiedenen Zugangsnetzwerk-Technologien; einer der ersten Punkte hierbei ist sicherlich

die Integration der verschiedenen Sicherheitsmechanismen [Bem02]. Es ist anzunehmen, dass viele Sicherheitsmechnismen auf IP-Ebene (OSI Layer 3), wie z.B. IPSec oder IPSec basierendes VPN (*Virtual Private Network*), angewendet werden. Abbildung 6.1 zeigt ein Referenzmodell einer 4G Architektur.

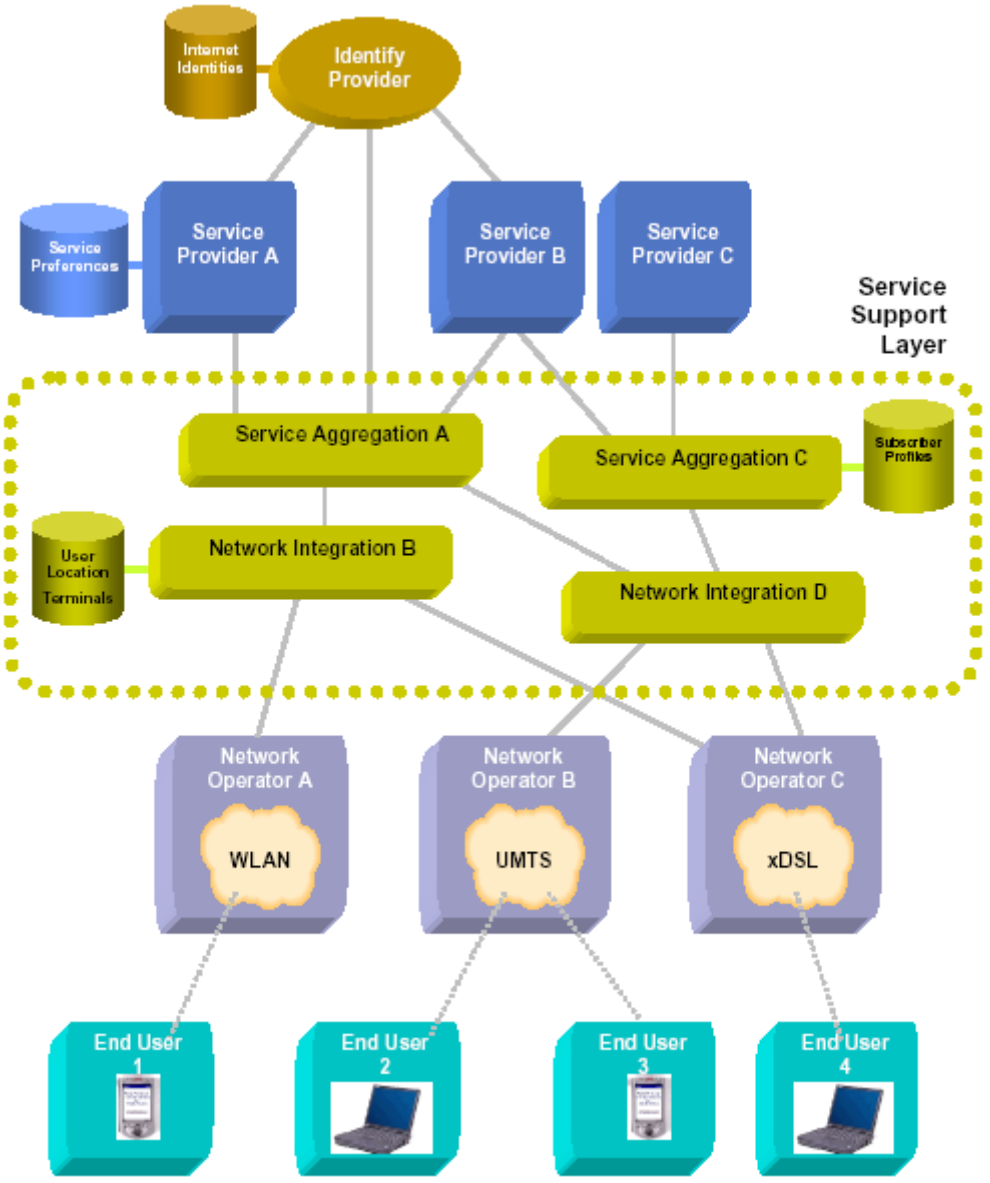

Abbildung 6.1: 4G-Dienste-Architektur (Quelle: [Bem02])

Die Sicherheitsmodelle der 3G Netze werden also wahrscheinlich teilweise auch in 4G Netzen wiederzufinden sein, auch wenn sich die gesamte 4G-Netzstruktur als umfassender darstellen wird. Die 3G Sicherheitsarchitektur wird auf jeden Fall in den nächsten Jahren ausreichenden Schutz für die Mobilfunknetze der dritten Generation geben und sicherheitskritischen Diensten, wie z.B. M-Commerce eine sichere Kommunikationsplattform sein.

A. Abkürzungen, Symbole

A.1 Abkürzungsverzeichnis

2G	2nd Generation
3G	3rd Generation
3GPP	Third Generation Partnership Project
A5/3	Encryption algorithm A5/3
AK	Anonymity Key
AKA	Authentication and key agreement
AMF	Authentication management field
AN	Access Network
ANF	Algebraische Normalform
ATM	Asynchronous Transfer Mode
AuC	Authentication Centre
AUTN	Authentication Token
AV	Authentication Vector
BTS	Base Transceiver Station
CBC	Cipher Block Chaining
CDMA	Code Division Multiple Access
CK	Cipher Key
CN	Core Network
CPU	Central Processing Unit
CS	Circuit Switched
DP	Differential Probability
EDGE	Enhanced Data rates for GSM Evolution
ETSI	European Telecommunications Standards Institute
GGSN	Gateway GPRS Support Node
GMSC	Gateway MSC
GPRS	General Packet Radio Service
HE	Home Environment

HLR	Home Location Register
ICC	Integrated Circuit Card
ID	Identifier
IK	Integrity Key
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity
IMT-2000	International Mobile Telecommunications 2000
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ISO	International Organisation for Standardisation
ITU	International Telecommunication Union
K	USIM Individual key
KSI	Key Set Identifier
LAI	Location Area Identity
DP	Linear Probability
MAC	Message Authentication Code
MAC-I	MAC used for data integrity of signalling messages
MAP	Mobile Application Part
ME	Mobile Equipment
MS	Mobile Station
MSC	Mobile Services Switching Centre
OFB	Output Feedback
PIN	Personal Identification Number
PS	Packet Switched
PSTN	Public Switched Telephone Network
RAN	Radio Access Network
RAND	Random challenge
REQ	REQuest
RES	user RESponse
RNC	Radio Network Controller
SC	Service Centre (used for SMS)
SGSN	Serving GPRS Support Node
SIM	(GSM) Subscriber Identity Module
SMS	Short Message Service
SN	Serving Network
SQN	Sequence number
SRNC	Serving Radio Network Controller
TMSI	Temporary Mobile Subscriber Identity
TRAU	Transcoder and Rate Adapter Unit
TS	Technical Specification
UE	User Equipment
UEA	UMTS Encryption Algorithm
UIA	UMTS Integrity Algorithm
UICC	UMTS IC Card

UMTS	Universal Mobile Telecommunication System
USIM	UMTS Subscriber Identity Module
UTRAN	Universal Terrestrial Radio Access Network
VLR	Visitor Location Register
WLAN	Wireless Local Area Network
XMAC	Expected Message Authentication Code (calculated by the USIM)
XRES	Expected Response

A.2 Symbole

$\|\|$	Verkettung
$\oplus$	Exklusiv-ODER (XOR)
$\lll$	Bitweise Links-Rotation
$\cap$	Logisches UND (AND)
$\cup$	Logisches ODER (OR)
$GF(2^n)$	Galois-Feld

B. KASUMIs S-Boxen

Beide S-Boxen sind jeweils als Dezimal-Tabelle und in Gate-Logik-Form bzw. Algebraischer Normalform (ANF) dargestellt. Die Eingabe X besteht aus $n = 7$ bzw. $n = 9$ Bit in der Form $X = x_n||x_{n-1}||...||x_0$, ebenso wie die Ausgabe Y. x_0 bzw. y_0 sind die jeweils niederwertigsten Bit.

In den Logik-Gleichungen gilt: z.B. $x_0x_1x_2 \Leftrightarrow x_0 \cap x_1 \cap x_2$.

B.1 S7

Dezimal-Tabelle:

54	50	62	56	22	34	94	96	38	6	63	93	2	18	123	33
55	113	39	114	21	67	65	12	47	73	46	27	25	111	124	81
53	9	121	79	52	60	58	48	101	127	40	120	104	70	71	43
20	122	72	61	23	109	13	100	77	1	16	7	82	10	105	98
117	116	76	11	89	106	0	125	118	99	86	69	30	57	126	87
112	51	17	5	95	14	90	84	91	8	35	103	32	97	28	66
102	31	26	45	75	4	85	92	37	74	80	49	68	29	115	44
64	107	108	24	110	83	36	78	42	19	15	41	88	119	59	3

Gate-Logik (ANF):

$$
\begin{aligned}
y_0 &= x_1x_3 \oplus x_4 \oplus x_0x_1x_4 \oplus x_5 \oplus x_2x_5 \oplus x_3x_4x_5 \oplus x_6 \oplus x_0x_6 \oplus x_1x_6 \\
&\quad \oplus x_3x_6 \oplus x_2x_4x_6 \oplus x_1x_5x_6 \oplus x_4x_5x_6 \\
y_1 &= x_0x_1 \oplus x_0x_4 \oplus x_2x_4 \oplus x_5 \oplus x_1x_2x_5 \oplus x_0x_3x_5 \oplus x_6 \oplus x_0x_2x_6 \\
&\quad \oplus x_3x_6 \oplus x_4x_5x_6 \oplus 1 \\
y_2 &= x_0 \oplus x_0x_3 \oplus x_2x_3 \oplus x_1x_2x_4 \oplus x_0x_3x_4 \oplus x_1x_5 \oplus x_0x_2x_5 \oplus x_0x_6 \\
&\quad \oplus x_0x_1x_6 \oplus x_2x_6 \oplus x_4x_6 \oplus 1 \\
y_3 &= x_1 \oplus x_0x_1x_2 \oplus x_1x_4 \oplus x_3x_4 \oplus x_0x_5 \oplus x_0x_1x_5 \oplus x_2x_3x_5 \oplus x_1x_4x_5 \\
&\quad \oplus x_2x_6 \oplus x_1x_3x_6 \\
y_4 &= x_0x_2 \oplus x_3 \oplus x_1x_3 \oplus x_1x_4 \oplus x_0x_1x_4 \oplus x_2x_3x_4 \oplus x_0x_5 \oplus x_1x_3x_5 \oplus x_0x_4x_5 \\
&\quad \oplus x_1x_6 \oplus x_3x_6 \oplus x_0x_3x_6 \oplus x_5x_6 \oplus 1 \\
y_5 &= x_2 \oplus x_0x_2 \oplus x_0x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_4 \oplus x_0x_5 \oplus x_2x_5 \oplus x_4x_5 \oplus x_1x_6 \\
&\quad \oplus x_1x_2x_6 \oplus x_0x_3x_6 \oplus x_3x_4x_6 \oplus x_2x_5x_6 \oplus 1 \\
y_6 &= x_1x_2 \oplus x_0x_1x_3 \oplus x_0x_4 \oplus x_1x_5 \oplus x_3x_5 \oplus x_6 \oplus x_0x_1x_6 \oplus x_2x_3x_6 \\
&\quad \oplus x_1x_4x_6 \oplus x_0x_5x_6
\end{aligned}
$$

Beispiel: Angenommen, die Eingabe der S-Box ist dezimal 61. Die Dezimal-Tabelle ist von links nach rechts und von oben nach unten, von 0 aufsteigend durchgehend nummeriert. Die Anzahl der Felder beträgt $16 \times 8 = 2^7$. Die zugeordnete Zahl ist dann $S7_{61} = 10$.

Binär erhält man:
$61 = \text{‘0111101‘} \Rightarrow x_6 = 0, x_5 = 1, x_4 = 1, x_3 = 1, x_2 = 1, x_1 = 0, x_0 = 1$
daraus ergibt sich:

$$
\begin{aligned}
y_0 &= 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 = 0 \\
y_1 &= 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 1 \\
y_2 &= 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0 \\
y_3 &= 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 = 1 \\
y_4 &= 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0 \\
y_5 &= 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0 \\
y_6 &= 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 = 0
\end{aligned}
$$

damit ist $Y = \text{‘0001010‘} = 10$

B.2 S9

Dezimal-Tabelle:

167	239	161	379	391	334	9	338	38	226	48	358	452	385	90	397
183	253	147	331	415	340	51	362	306	500	262	82	216	159	356	177
175	241	489	37	206	17	0	333	44	254	378	58	143	220	81	400
95	3	315	245	54	235	218	405	472	264	172	494	371	290	399	76
165	197	395	121	257	480	423	212	240	28	462	176	406	507	288	223
501	407	249	265	89	186	221	428	164	74	440	196	458	421	350	163
232	158	134	354	13	250	491	142	191	69	193	425	152	227	366	135
344	300	276	242	437	320	113	278	11	243	87	317	36	93	496	27
487	446	482	41	68	156	457	131	326	403	339	20	39	115	442	124
475	384	508	53	112	170	479	151	126	169	73	268	279	321	168	364
363	292	46	499	393	327	324	24	456	267	157	460	488	426	309	229
439	506	208	271	349	401	434	236	16	209	359	52	56	120	199	277
465	416	252	287	246	6	83	305	420	345	153	502	65	61	244	282
173	222	418	67	386	368	261	101	476	291	195	430	49	79	166	330
280	383	373	128	382	408	155	495	367	388	274	107	459	417	62	454
132	225	203	316	234	14	301	91	503	286	424	211	347	307	140	374
35	103	125	427	19	214	453	146	498	314	444	230	256	329	198	285
50	116	78	410	10	205	510	171	231	45	139	467	29	86	505	32
72	26	342	150	313	490	431	238	411	325	149	473	40	119	174	355
185	233	389	71	448	273	372	55	110	178	322	12	469	392	369	190
1	109	375	137	181	88	75	308	260	484	98	272	370	275	412	111
336	318	4	504	492	259	304	77	337	435	21	357	303	332	483	18
47	85	25	497	474	289	100	269	296	478	270	106	31	104	433	84
414	486	394	96	99	154	511	148	413	361	409	255	162	215	302	201
266	351	343	144	441	365	108	298	251	34	182	509	138	210	335	133
311	352	328	141	396	346	123	319	450	281	429	228	443	481	92	404
485	422	248	297	23	213	130	466	22	217	283	70	294	360	419	127
312	377	7	468	194	2	117	295	463	258	224	447	247	187	80	398
284	353	105	390	299	471	470	184	57	200	348	63	204	188	33	451
97	30	310	219	94	160	129	493	64	179	263	102	189	207	114	402
438	477	387	122	192	42	381	5	145	118	180	449	293	323	136	380
43	66	60	455	341	445	202	432	8	237	15	376	436	464	59	461

Gate-Logik (ANF):

$$
\begin{aligned}
y_0 &= x_0x_2 \oplus x_3 \oplus x_2x_5 \oplus x_5x_6 \oplus x_0x_7 \oplus x_1x_7 \oplus x_2x_7 \oplus x_4x_8 \\
&\quad \oplus x_5x_8 \oplus x_7x_8 \oplus 1 \\
y_1 &= x_1 \oplus x_0x_1 \oplus x_2x_3 \oplus x_0x_4 \oplus x_1x_4 \oplus x_0x_5 \oplus x_3x_5 \oplus x_6 \oplus x_1x_7 \\
&\quad \oplus x_2x_7 \oplus x_5x_8 \oplus 1 \\
y_2 &= x_1 \oplus x_0x_3 \oplus x_3x_4 \oplus x_0x_5 \oplus x_2x_6 \oplus x_3x_6 \oplus x_5x_6 \oplus x_4x_7 \oplus x_5x_7 \\
&\quad \oplus x_6x_7 \oplus x_8 \oplus x_0x_8 \oplus 1 \\
y_3 &= x_0 \oplus x_1x_2 \oplus x_0x_3 \oplus x_2x_4 \oplus x_5 \oplus x_0x_6 \oplus x_1x_6 \oplus x_4x_7 \oplus x_0x_8 \\
&\quad \oplus x_1x_8 \oplus x_7x_8 \\
y_4 &= x_0x_1 \oplus x_1x_3 \oplus x_4 \oplus x_0x_5 \oplus x_3x_6 \oplus x_0x_7 \oplus x_6x_7 \oplus x_1x_8 \oplus x_2x_8 \oplus x_3x_8 \\
y_5 &= x_2 \oplus x_1x_4 \oplus x_4x_5 \oplus x_0x_6 \oplus x_1x_6 \oplus x_3x_7 \oplus x_4x_7 \oplus x_6x_7 \\
&\quad \oplus x_5x_8 \oplus x_6x_8 \oplus x_7x_8 \oplus 1 \\
y_6 &= x_0 \oplus x_2x_3 \oplus x_1x_5 \oplus x_2x_5 \oplus x_4x_5 \oplus x_3x_6 \oplus x_4x_6 \oplus x_5x_6 \\
&\quad \oplus x_7 \oplus x_1x_8 \oplus x_3x_8 \oplus x_5x_8 \oplus x_7x_8 \\
y_7 &= x_0x_1 \oplus x_0x_2 \oplus x_1x_2 \oplus x_3 \oplus x_0x_3 \oplus x_2x_3 \oplus x_4x_5 \oplus x_2x_6 \\
&\quad \oplus x_3x_6 \oplus x_2x_7 \oplus x_5x_7 \oplus x_8 \oplus 1 \\
y_8 &= x_0x_1 \oplus x_2 \oplus x_1x_2 \oplus x_3x_4 \oplus x_1x_5 \oplus x_2x_5 \oplus x_1x_6 \oplus x_4x_6 \\
&\quad \oplus x_7 \oplus x_2x_8 \oplus x_3x_8
\end{aligned}
$$

Beispiel: Angenommen, die Eingabe der S-Box ist dezimal 179. Die Dezimal-Tabelle ist von links nach rechts und von oben nach unten, von 0 aufsteigend durchgehend nummeriert. Die Anzahl der Felder beträgt $16 \times 32 = 2^9$. Die zugeordnete Zahl ist dann $S9_{179} = 271$.

Binär erhält man:
$179 =$ ‘010110011‘ $\Rightarrow$
$x_8 = 0, x_7 = 1, x_6 = 0, x_5 = 1, x_4 = 1, x_3 = 0, x_2 = 0, x_1 = 1, x_0 = 1$
daraus ergibt sich:

$$
\begin{aligned}
y_0 &= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 1 \\
y_1 &= 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1 \\
y_2 &= 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 1 \\
y_3 &= 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 1 \\
y_4 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus = 0 \\
y_5 &= 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0 \\
y_6 &= 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 = 0 \\
y_7 &= 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 0 \\
y_8 &= 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 1
\end{aligned}
$$

damit ist $Y =$ ‘100001111‘ $= 271$

C. Tabellen

	1	2	3	4	5	6	7	8
$KL_{i,1}$	$K1 \lll 1$	$K2 \lll 1$	$K3 \lll 1$	$K4 \lll 1$	$K5 \lll 1$	$K6 \lll 1$	$K7 \lll 1$	$K8 \lll 1$
$KL_{i,2}$	$K3'$	$K4'$	$K5'$	$K6'$	$K7'$	$K8'$	$K1'$	$K2'$
$KO_{i,1}$	$K2 \lll 5$	$K3 \lll 5$	$K4 \lll 5$	$K5 \lll 5$	$K6 \lll 5$	$K7 \lll 5$	$K8 \lll 5$	$K1 \lll 5$
$KO_{i,2}$	$K6 \lll 8$	$K7 \lll 8$	$K8 \lll 8$	$K1 \lll 8$	$K2 \lll 8$	$K3 \lll 8$	$K4 \lll 8$	$K5 \lll 8$
$KO_{i,3}$	$K7 \lll 13$	$K8 \lll 13$	$K1 \lll 13$	$K2 \lll 13$	$K3 \lll 13$	$K4 \lll 13$	$K5 \lll 13$	$K6 \lll 13$
$KI_{i,1}$	$K5'$	$K6'$	$K7'$	$K8'$	$K1'$	$K2'$	$K3'$	$K4'$
$KI_{i,2}$	$K4'$	$K5'$	$K6'$	$K7'$	$K8'$	$K1'$	$K2$’	$K3'$
$KI_{i,3}$	$K8'$	$K1'$	$K2'$	$K3'$	$K4'$	$K5'$	$K6'$	$K7'$

Tabelle C.1: Die Rundenteilschlüssel

C_1	0x0123
C_2	0x4567
C_3	0x89AB
C_4	0xCDEF
C_5	0xFEDC
C_6	0xBA98
C_7	0x7654
C_8	0x3210

Tabelle C.2: Die Konstanten C_j

D. *Mathematica* Quellcode

D.1 kasumi_pack.m

```
(***************************** Kasumi Functions ************************)


(*************************************************************************)
(*************************************************************************)
(*************************************************************************)
(** f8 Version 1 mit zwei Parameter. A ist fest vorbelegt.**)
(*************************************************************************)

f8[plain_, CK_] := Module[
    {
        KM, A, BLKCNT, KSB, blocks,
        length, count, bearer, direction, padding,
        out
    },

    (*********************************************************************)

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[Length[CK]!= 128
      || (Length[Select[plain, # != 1 && # !=0 &]] != 0)
      || (Length[Select[CK, # != 1 && # !=0 &]] != 0),

      Message[f8::sizeerror],

      (** Else **)
      (**
       Berechnung der Anzahl der Blöcke aus der Länge des Klartextstroms
       dividiert durch 64 und falls nötig aufgerundet auf die nächst größere
       ganze Zahl.
       **)
      length = Length[plain];
      blocks = Ceiling[length/64];
```

```
      (**
       Initialisierung der Parameter
       **)

      (** Parameter COUNT-C (32 Bit), BEARER (5 Bit), DIRECTION (1 Bit) **)
      count = {0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0,
               1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1};
      bearer = {0, 1, 1, 0, 0};
      direction = {1};
      padding = Table[0, {26}];

      (**Arbeitsregister A **)
      A = Join[count, bearer, direction, padding];
      (** Blockzähler-Variable **)
      BLKCNT = Table[0, {64}];
      (** Key Modifier,  Hex Zahl in binäre Liste wandeln **)
      KM = IntegerDigits[16^^55555555555555555555555555555555, 2, 128];
      (** Blockliste formatieren mit blocks Unterlisten plus dem Startblock **)
      KSB = Table[{}, {blocks+1}];

      (** Startblock auf Null gesetzt **)
      KSB[[1]] = Table[0, {64}];
      (*****************************************************************)

      (**
       Ablauf des Algorithmus
       **)

      (** Erste Anwendung von Kasumi mit modifiziertem Schlüssel **)
      A = Kasumi[A, BitXor[CK, KM]];

      (** Berechnung der Schlüsselstromblöcke im Output-Feedback-Mode **)
      For[n = 1, n <= blocks, n++,
         KSB[[n+1]] = Kasumi[BitXor[BitXor[A, BLKCNT], KSB[[n]]], CK];
         BLKCNT = BinPlus[BLKCNT, 1, 64]
          ];

      (** Entfernen des Startblocks **)
      KSB = Drop[KSB, 1];

      (** Verketten der Blöcke zu einer Liste **)
      KS = Flatten[KSB];

      (**
       Verwerfen der letzten Bit, sodass die Länge des Schlüsselstroms
       gleich der Länge des Klartextstroms ist **)
      KS = Drop[KS, -(blocks * 64 - length)]

    ]      (** If! **)
](** Module! **)

(*************************************************************************)
(*************************************************************************)
(** f8 Version 2 mit drei Parameter. IV bzw. A wird übergeben.**)
(*************************************************************************)

f8[plain_, CK_, IV_] := Module[
    {
    KM, A, BLKCNT, KSB, blocks, length,
```

```
out
},

(*********************************************************************)

(**
 Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
 **)
If[(Length[CK]!= 128) || (Length[IV] != 64)
  || (Length[Select[plain, # != 1 && # !=0 &]] != 0)
  || (Length[Select[CK, # != 1 && # !=0 &]] != 0)
  || (Length[Select[IV, # != 1 && # !=0 &]] != 0),

  Message[f8::sizeerror],

  (** Else **)
  (**
   Berechnung der Anzahl der Blöcke aus der Länge des Klartextstroms
   dividiert durch 64 und falls nötig aufgerundet auf die nächst größere
   ganze Zahl.
   **)
  length = Length[plain];
  blocks = Ceiling[length/64];

  (**
   Initialisierung der Parameter
   **)

  (** Arbeitsregister **)
  A = IV;
  (** Blockzähler-Variable **)
  BLKCNT = Table[0, {64}];
  (** Key Modifier,  Hex Zahl in binäre Liste wandeln **)
  KM = IntegerDigits[16^^55555555555555555555555555555555, 2, 128];
  (** Blockliste formatieren mit blocks Unterlisten plus dem Startblock **)
  KSB = Table[{}, {blocks+1}];

  (** Startblock auf Null gesetzt **)
  KSB[[1]] = Table[0, {64}];
  (*********************************************************************)

  (**
   Ablauf des Algorithmus
   **)

  (** Erste Anwendung von Kasumi mit modifiziertem Schlüssel **)
  A = Kasumi[A, BitXor[CK, KM]];

  (** Berechnung der Schlüsselstromblöcke im Output-Feedback-Mode **)
  For[n = 1, n <= blocks, n++,
     KSB[[n+1]] = Kasumi[BitXor[BitXor[A, BLKCNT], KSB[[n]]], CK];
     BLKCNT = BinPlus[BLKCNT, 1, 64]
      ];

  (** Entfernen des Startblocks **)
  KSB = Drop[KSB, 1];

  (** Verketten der Blöcke zu einer Liste **)
  KS = Flatten[KSB];
```

```
      (**
       Verwerfen der letzten Bit, sodass die Länge des Schlüsselstroms
       gleich der Länge des Klartextstroms ist **)
      KS = Drop[KS, -(blocks * 64 - length)]
   ] (** If! **)
](** Module! **)

(*****************************************************************************)
(*****************************************************************************)
(** f8 Ver 3 mit fünf Parametern. COUNT, BEARER, DIRECTION wird übergeben.**)
(*****************************************************************************)

f8[plain_, CK_, COUNT_, BEARER_, DIRECTION_] := Module[
    {
        KM, A, BLKCNT, KSB, blocks,
        length, padding,
        out
    },

    (*************************************************************************)

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[CK]!= 128) || (Length[COUNT] != 32)
      || (Length[BEARER] != 5) || (Length[DIRECTION] != 1)
      || (Length[Select[plain, # != 1 && # !=0 &]] != 0)
      || (Length[Select[CK, # != 1 && # !=0 &]] != 0)
      || (Length[Select[COUNT, # != 1 && # !=0 &]] != 0)
      || (Length[Select[BEARER, # != 1 && # !=0 &]] != 0)
      || (Length[Select[DIRECTION, # != 1 && # !=0 &]] != 0),

      Message[f8::sizeerror],

      (** Else **)
      (**
       Berechnung der Anzahl der Blöcke aus der Länge des Klartextstroms
       dividiert durch 64 und falls nötig aufgerundet auf die nächst größere
       ganze Zahl.
       **)
      length = Length[plain];
      blocks = Ceiling[length/64];

      (**
       Initialisierung der Parameter
       **)
      (** 26 Nullen zum auffüllen von A **)
      padding = Table[0, {26}];
      (** Arbeitsregister **)
      A = Join[COUNT, BEARER, DIRECTION, padding];
      (** Blockzähler-Variable **)
      BLKCNT = Table[0, {64}];
      (** Key Modifier,  Hex Zahl in binäre Liste wandeln **)
      KM = IntegerDigits[16^^55555555555555555555555555555555, 2, 128];
      (** Blockliste formatieren mit blocks Unterlisten plus dem Startblock **)
      KSB = Table[{}, {blocks+1}];

      (** Startblock auf Null gesetzt **)
```

```
      KSB[[1]] = Table[0, {64}];
      (*****************************************************************)

      (**
       Ablauf des Algorithmus
       **)

      (** Erste Anwendung von Kasumi mit modifiziertem Schlüssel **)
      A = Kasumi[A, BitXor[CK, KM]];

      (** Berechnung der Schlüsselstromblöcke im Output-Feedback-Mode **)
      For[n = 1, n <= blocks, n++,
         KSB[[n+1]] = Kasumi[BitXor[BitXor[A, BLKCNT], KSB[[n]]], CK];
         BLKCNT = BinPlus[BLKCNT, 1, 64]
          ];

      (** Entfernen des Startblocks **)
      KSB = Drop[KSB, 1];

      (** Verketten der Blöcke zu einer Liste **)
      KS = Flatten[KSB];

      (**
       Verwerfen der letzten Bit, sodass die Länge des Schlüsselstroms
       gleich der Länge des Klartextstroms ist **)
      KS = Drop[KS, -(blocks * 64 - length)]
    ] (** If! **)
](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** f9 Version 1 mit zwei Parametern. COUNT_, DIRECTION_, FRESH_ sind fix.**)
(*****************************************************************************)

f9[msg_, IK_] := Module[
    {
        COUNT, FRESH, DIRECTION, length,
        PS, pslen, padding, z, blocks,
        A, B,
        out
    },

    (*****************************************************************)

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[IK]!= 128)
      || (Length[Select[msg, # != 1 && # !=0 &]] != 0)
      || (Length[Select[IK, # != 1 && # !=0 &]] != 0),

      Message[f9::sizeerror],

      (** Else **)
      (**
       Initialisierung der Variablen
       **)
```

```
      COUNT = {0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0,
               1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0};
      FRESH = {0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0,
               1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1};
      DIRECTION = {0};

      A = Table[0, {64}];
      B = Table[0, {64}];
      KM = IntegerDigits[16^^AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, 2, 128];

      (**
       Bestimmung der z Füll-Nullen, die die verketteten Parameter
       in PS zu einem Vielfachen von 64 Bit erweitern.
       **)
      pslen = Length[Join[COUNT, FRESH, msg, DIRECTION, {1}]];
      z = 64 - (Mod[(pslen - 1), 64] + 1);

      (**
       PS ergibt sich dann aus der Verkettung und wird anschließend
       in 64 Bit Blöcke aufgeteilt
       **)
      PS = Join[COUNT, FRESH, msg, DIRECTION, {1}, Table[0, {z}]];
      PS = Partition[PS, 64];
      blocks = Length[PS];

      (*********************************************************************)

      (**
       Ablauf des Algorithmus
       **)

      For[n = 1, n <= blocks, n++,
         A = Kasumi[BitXor[A, PS[[n]]], IK];
         B = BitXor[B, A]
         ];

      (** Schlussanwendung von Kasumi auf B mit verändertem Schlüssel **)
      B = Kasumi[B, BitXor[IK, KM]];

      (** Nur die linken 32 Bit von B dienen als MAC **)
      out = Take[B, 32]

   ] (** If! **)
] (** Module! **)

(*************************************************************************)
(*************************************************************************)
(** f9 Version 2 mit vier  Parametern. FRESH_ wird randomisiert.**)
(*************************************************************************)

f9[msg_, IK_, COUNT_, DIRECTION_] := Module[
    {
        FRESH, length,
        PS, pslen, padding, z, blocks,
        A, B,
        out
    },

    (*****************************************************************)
```

```
    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[IK]!= 128) || (Length[COUNT]!= 32) || (Length[DIRECTION]!= 1)
      || (Length[Select[msg, # != 1 && # !=0 &]] != 0)
      || (Length[Select[IK, # != 1 && # !=0 &]] != 0)
      || (Length[Select[COUNT, # != 1 && # !=0 &]] != 0)
      || (Length[Select[DIRECTION, # != 1 && # !=0 &]] != 0),

      Message[f9::sizeerror],

      (** Else **)
      (**
       Initialisierung der Variablen
       **)
      (** 32 Bit Pseudozufallszahl **)
      FRESH = Table[Random[Integer], {32}];

      A = Table[0, {64}];
      B = Table[0, {64}];
      KM = IntegerDigits[16^^AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, 2, 128];

      (**
       Bestimmung der z Füll-Nullen, die die verketteten Parameter
       in PS zu einem Vielfachen von 64 Bit erweitern.
       **)
      pslen = Length[Join[COUNT, FRESH, msg, DIRECTION, {1}]];
      z = 64 - (Mod[(pslen - 1), 64] + 1);

      (**
       PS ergibt sich dann aus der Verkettung und wird anschließend
       in 64 Bit Blöcke aufgeteilt
       **)
      PS = Join[COUNT, FRESH, msg, DIRECTION, {1}, Table[0, {z}]];
      PS = Partition[PS, 64];
      blocks = Length[PS];

      (**********************************************************************)

      (**
       Ablauf des Algorithmus
       **)

      For[n = 1, n <= blocks, n++,
         A = Kasumi[BitXor[A, PS[[n]]], IK];
         B = BitXor[B, A]
         ];

      (** Schlussanwendung von Kasumi auf B mit verändertem Schlüssel **)
      B = Kasumi[B, BitXor[IK, KM]];

      (** Nur die linken 32 Bit von B dienen als MAC **)
      out = Take[B, 32]

   ] (** If! **)
] (** Module! **)

(************************************************************************)
```

```
(*****************************************************************************)
(** f9 Version 3 mit fünf Parametern. **)
(*****************************************************************************)

f9[msg_, IK_, COUNT_, DIRECTION_, FRESH_] := Module[
    {
        length, PS, pslen, padding, z, blocks,
        A, B,
        out
    },

    (*********************************************************************)

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[Length[IK]!= 128 || (Length[COUNT]!= 32)
      || (Length[DIRECTION]!= 1) || (Length[FRESH]!= 32)
      || (Length[Select[msg, # != 1 && # !=0 &]] != 0)
      || (Length[Select[IK, # != 1 && # !=0 &]] != 0)
      || (Length[Select[COUNT, # != 1 && # !=0 &]] != 0)
      || (Length[Select[DIRECTION, # != 1 && # !=0 &]] != 0)
      || (Length[Select[FRESH, # != 1 && # !=0 &]] != 0),

      Message[f9::sizeerror],

      (** Else **)
      (**
       Initialisierung der Variablen
       **)
      A = Table[0, {64}];
      B = Table[0, {64}];
      KM = IntegerDigits[16^^AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, 2, 128];

      (**
       Bestimmung der z Füll-Nullen, die die verketteten Parameter
       in PS zu einem Vielfachen von 64 Bit erweitern.
       **)
      pslen = Length[Join[COUNT, FRESH, msg, DIRECTION, {1}]];
      z = 64 - (Mod[(pslen - 1), 64] + 1);

      (**
       PS ergibt sich dann aus der Verkettung und wird anschließend
       in 64 Bit Blöcke aufgeteilt
       **)
      PS = Join[COUNT, FRESH, msg, DIRECTION, {1}, Table[0, {z}]];
      PS = Partition[PS, 64];
      blocks = Length[PS];

      (*******************************************************************)

      (**
       Ablauf des Algorithmus
       **)

      For[n = 1, n <= blocks, n++,
         A = Kasumi[BitXor[A, PS[[n]]], IK];
         B = BitXor[B, A]
         ];
```

```
      (** Schlussanwendung von Kasumi auf B mit verändertem Schlüssel **)
      B = Kasumi[B, BitXor[IK, KM]];

      (** Nur die linken 32 Bit von B dienen als MAC **)
      out = Take[B, 32]

   ] (** If! **)
] (** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** Kasumi V1 mit zwei Parametern. **)
(*****************************************************************************)

Kasumi[in_, K_] := Module[
    {
     insplit, inleft, inright,
     newright, i, round,
     out
    },

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[in]!= 64) || (Length[K]!= 128)
      || (Length[Select[in, # != 1 && # !=0 &]] != 0)
      || (Length[Select[K, # != 1 && # !=0 &]] != 0),

      Message[Kasumi::sizeerror],

      (** Else **)
      (********* f Rundenfunktion für gerade und ungerade Runden ************)
      (********* erzeugt die neue linke Hälfte für die nächste Runde *********)
      f[left_, right_] := Module[
        {},

        If[Mod[round, 2] == 1,
          (** ungerade Runden **)
          BitXor[FO[FL[left, round], round], right],
          (** else, gerade Runden **)
          BitXor[FL[FO[left, round], round], right]
        ]
      ];
      (*****************************************)

      (** Erzeugen der Teilschlüssel mit der KeySchedule **)
      KeySchedule[K];

      (**
       Aufteilen des Eingabeblocks in zwei 32-Bit-Hälften.
       **)
      insplit = SplitBlock[in, 32, 32];
      inleft = insplit[[1]];
      inright = insplit[[2]];

      (**
```

```
       Acht mal die Rundenfunktion  mit den jeweiligen Rundenschlüseln
       durchlaufen und die Hälften vertauschen.
       **)
      For[i=1, i<=8, i++,
         round = i;
         newright = inleft;
         (** neue linke Hälfte **)
         inleft = f[inleft, inright];
         (** neue rechte Hälfte **)
         inright = newright
      ];

      (** Ausgabe der zwei verketteten Hälften **)
      out = Join[inleft, inright]

    ]      (** If! **)
](** Module! **)

(*****************************************************************************)
(*****************************************************************************)
(** Kasumi V2 mit drei Parametern. **)
(*****************************************************************************)

Kasumi[in_, K_, roundout_] := Module[
    {
     insplit, inleft, inright,
     newright, i, round, odd,
     out
    },

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[in]!= 64) || (Length[K]!= 128)
      || (Length[Select[in, # != 1 && # !=0 &]] != 0)
      || (Length[Select[K, # != 1 && # !=0 &]] != 0),

      Message[Kasumi::sizeerror],

      (** Else **)

      (** out formatieren **)
      out = Table[{}, {8}];

      (********* f Rundenfunktion für gerade und ungerade Runden ************)
      (********* erzeugt die neue linke Hälfte für die nächste Runde *********)
      f[left_, right_] := Module[
          {out},

          If[Mod[round, 2] == 1,
            (** ungerade Runden **)
            out =BitXor[FO[FL[left, round], round], right];
            odd = True,
            (** else, gerade Runden **)
            out = BitXor[FL[FO[left, round], round], right];
            odd = False
          ];
          out
      ];
```

```
      (****************************************)

      (** Erzeugen der Teilschlüssel mit der KeySchedule **)
      KeySchedule[K];

      (**
       Aufteilen des Eingabeblocks in zwei 32-Bit-Hälften.
       **)
      insplit = SplitBlock[in, 32, 32];
      inleft = insplit[[1]];
      inright = insplit[[2]];

      (**
       Acht mal die Rundenfunktion  mit den jeweiligen Rundenschlüseln
       durchlaufen und die Hälften vertauschen.
       **)
      For[i=1, i<=8, i++,
         round = i;
         newright = inleft;
         (** neue linke Hälfte **)
         inleft = f[inleft, inright];
         (** neue rechte Hälfte **)
         inright = newright;
         If[roundout == 0,
           (** nur die achte Runde ausgeben**)
           If[round == 8,
             out = {{}};
             out[[1]] = Join[inleft, inright]
           ],

           (** Else, Rundenergebnisse eintragen **)
           If[odd,
             (** ungerade Runde, Hälften vertauschen wegen Hamming Distanz **)
             out[[round]] = Join[inright, inleft],
             (** Else, gerade Runde **)
             out[[round]] = Join[inleft, inright]
           ]
         ] (** If! **)
      ]; (** For! **)

      (** Ausgabe der Liste **)
      out
   ] (** If! **)
](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** FL
Author: Axel Bolta, Date: 2003/06/16, Last update: 2003/06/16 **)
(*****************************************************************************)

FL[in_, round_] := Module[
    {
     insplit, inleft, inright,
     outleft, outright, out
    },
```

```
    (**
     Aufteilen des Eingabeblocks zwei 16 Bit Hälften.
     **)
    insplit = SplitBlock[in, 16, 16];
    inleft = insplit[[1]];
    inright = insplit[[2]];

    (**
     Die linke Hälfte inleft mit der linken Hälfte von KL_ UND-verknüpfen, das
     Ergebnis um eins links-rotieren und mit inright XOR-verknüpfen.
     **)
    outright = BitXor[RotateLeft[BitAnd[inleft, KL[[round, 1]]], 1], inright];

    (**
     Die neu berechnete rechte Hälfte outright mit der rechten Hälfte von KL_
     ODER-verknüpfen, das Ergebnis um eins links-rotieren und mit inleft
     XOR-verknüpfen.
     **)
    outleft = BitXor[RotateLeft[BitOr[outright, KL[[round, 2]]], 1], inleft];

    (**
    Ausgabe ist eine Liste der verketteten Hälften outleft und outright.
     **)
    out = Join[outleft, outright]
]


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** FO
Author: Axel Bolta, Date: 2003/06/16, Last update: 2003/06/16 **)
(*****************************************************************************)
FO[in_, round_] := Module[
    {
     insplit, inleft, inright,
     newleft, j, roundFO,
     out
    },

    (********* FO Rundenfunktion ***************)
    FOround[left_, right_, ko_, ki_] := Module[
        {},

        (** Ausgabe der neuen rechten Hälfte **)
        BitXor[FI[BitXor[left, ko], round, roundFO], right]
    ];
    (*****************************************)

    (**
     Aufteilen des Eingabeblocks in zwei 16-Bit-Hälften.
     **)
    insplit = SplitBlock[in, 16, 16];
    inleft = insplit[[1]];
    inright = insplit[[2]];

    (**
     Drei mal die Rundenfunktion durchlaufen und Hälften vertauschen
     mit den jeweiligen Rundenteilschlüsseln.
```

```
     **)
    For[j=1, j<=3, j++,
        roundFO=j;
        newleft = inright;
        (** neue rechte Hälfte **)
        inright = FOround[inleft, inright, KO[[round, j]], KI[[round, j]]];
        (** neue linke Hälfte **)
        inleft = newleft;
    ];

    out = Join[inleft, inright]
]


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** FI
Author: Axel Bolta, Date: 2003/06/16, Last update: 2003/06/16 **)
(*****************************************************************************)
FI[in_, round_, roundFO_] := Module[
    {
     insplit, KIsplit,
     KI7, KI9,
     inleft0, inright0,
     inleft1, inright1,
     inleft2, inright2,
     inleft3, inright3,
     outleft, outright, out
    },

    (**
     Aufteilen des Eingabeblocks und des Schlüssels je in einen 9-Bit-Block
     und einen 7-Bit-Block.
     **)
    insplit = SplitBlock[in, 9, 7];
    inleft0 = insplit[[1]];
    inright0 = insplit[[2]];

    KIsplit = SplitBlock[KI[[round, roundFO]], 7, 9];
    KI7 = KIsplit[[1]];
    KI9 = KIsplit[[2]];

    (** Rechter und linker Block nach erster Runde **)
    inright1 = BitXor[S9[inleft0], ZeroExtend[inright0, 2, "h"]];
    inleft1 = inright0;

    (** Zweite Runde **)
    inright2 = BitXor[BitXor[S7[inleft1], Truncate[inright1, 2, "h"]], KI7];
    inleft2 = BitXor[inright1, KI9];

    (** Dritte Runde **)
    inright3 = BitXor[S9[inleft2], ZeroExtend[inright2, 2, "h"]];
    inleft3 = inright2;

    (** Vierte und letzte Runde **)
    outright = inright3;
    outleft = BitXor[S7[inleft3], Truncate[inright3, 2, "h"]];
```

```
    (** Ausgabe durch Zusammensetzen der Hälften zu 16 Bit **)
    out = Join[outleft, outright]
]


(*************************************************************************)
(*************************************************************************)
(*************************************************************************)
(** KeySchedule
Author: Axel Bolta, Date: 2003/06/13, Last update: 2003/06/13 **)
(*************************************************************************)
KeySchedule[K_] := Module[
    {Cj, Kc, Kj},

    (** Tabelle der Konstanten Cj anlegen **)
    Cj = {
         {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1},
         {0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1},
         {1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1},
         {1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1},
         {1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0},
         {1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0},
         {0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0},
         {0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0}
    };

    (** Aufteilen des 128 Bit Schlüssels in 16 Bit große Blöcke **)
    Kj = Partition[K, 16];

    (** Vorbelegen der Schlüssel-Variablen mit leeren Tabellen **)
    Kc = Table[{}, {8}];
    KL = Table[{}, {8}, {2}];
    KO = Table[{}, {8}, {3}];
    KI = Table[{}, {8}, {3}];

    (** Berechne K\', hier als Kc bezeichnet **)
    For[i=1, i<=8, i++,
       Kc[[i]] = BitXor[Kj[[i]], Cj[[i]]]
        ];

    (** Berechne die einzelnen Teilschlüssel der KeySchedule **)
    For[i=1, i<=8, i++,
       KL[[i, 1]] = RotateLeft[Kj[[i]], 1];
       KL[[i, 2]] = Kc[[(1 + Mod[(1 + i), 8])]];

       KO[[i, 1]] = RotateLeft[Kj[[(1 + Mod[(0 + i), 8])]], 5];
       KO[[i, 2]] = RotateLeft[Kj[[(1 + Mod[(4 + i), 8])]], 8];
       KO[[i, 3]] = RotateLeft[Kj[[(1 + Mod[(5 + i), 8])]], 13];

       KI[[i, 1]] = Kc[[(1 + Mod[(3 + i), 8])]];
       KI[[i, 2]] = Kc[[(1 + Mod[(2 + i), 8])]];
       KI[[i, 3]] = Kc[[(1 + Mod[(6 + i), 8])]]
    ]

]


(*************************************************************************)
(*************************************************************************)
```

```
(*************************************************************************)
(** S7
Author: Axel Bolta, Date: 2003/06/12, Last update: 2003/06/12 **)
(*************************************************************************)
S7[xlist_] := Module[
    {s7list, ylist, z},

    (**
     Prüfe, ob xlist_ genau sieben Elemente hat.
     **)
    If[Length[xlist] == 7,
      (**
       Wenn ja,definiere s7list, mache den Table-Lookup an der Stelle xlist+1,
       da die Tabelle eigentlich bei Null anfängt. Mathematica-Listen haben
       aber Eins als ersten Index.
       **)
      s7list = {
           54, 50, 62, 56, 22, 34, 94, 96, 38,  6, 63, 93,  2, 18,123, 33,
           55,113, 39,114, 21, 67, 65, 12, 47, 73, 46, 27, 25,111,124, 81,
           53,  9,121, 79, 52, 60, 58, 48,101,127, 40,120,104, 70, 71, 43,
           20,122, 72, 61, 23,109, 13,100, 77,  1, 16,  7, 82, 10,105, 98,
          117,116, 76, 11, 89,106,  0,125,118, 99, 86, 69, 30, 57,126, 87,
          112, 51, 17,  5, 95, 14, 90, 84, 91,  8, 35,103, 32, 97, 28, 66,
          102, 31, 26, 45, 75,  4, 85, 92, 37, 74, 80, 49, 68, 29,115, 44,
           64,107,108, 24,110, 83, 36, 78, 42, 19, 15, 41, 88,119, 59,  3
      };
      ylist = s7list[[FromDigits[xlist, 2]+1]],
      (** else **)
      (S7::sizeerror = "Eingabeliste war ungleich sieben Elemente!";
       Message[S7::sizeerror])
    ];

    (** Ausgabe als sieben-stellige Binärzahl**)
    IntegerDigits[ylist, 2, 7]
]


(*************************************************************************)
(*************************************************************************)
(*************************************************************************)
(** S9
Author: Axel Bolta, Date: 2003/06/12, Last update: 2003/06/12 **)
(*************************************************************************)
S9::usage = "S9[xlist_] macht einen Table-Lookup in der KASUMI S-Box S9 und
liefert den Wert an der Stelle xlist_ zurück. Als Eingabe wird eine binäre
Liste mit neun Stellen erwartet; ebenso groß ist die Ausgabe."

S9[xlist_] := Module[
    {s9list, ylist, z},

    (**
     Prüfe, ob xlist_ genau neun Elemente hat.
     **)
    If[Length[xlist] == 9,
      (**
       Wenn ja,definiere s9list, mache den Table-Lookup an der Stelle xlist+1,
       da die Tabelle eigentlich bei Null anfängt. Mathematica-Listen haben
       aber Eins als ersten Index.
       **)
```

```
      s9list = {
          167,239,161,379,391,334,  9,338, 38,226, 48,358,452,385, 90,397,
          183,253,147,331,415,340, 51,362,306,500,262, 82,216,159,356,177,
          175,241,489, 37,206, 17,  0,333, 44,254,378, 58,143,220, 81,400,
           95,  3,315,245, 54,235,218,405,472,264,172,494,371,290,399, 76,
          165,197,395,121,257,480,423,212,240, 28,462,176,406,507,288,223,
          501,407,249,265, 89,186,221,428,164, 74,440,196,458,421,350,163,
          232,158,134,354, 13,250,491,142,191, 69,193,425,152,227,366,135,
          344,300,276,242,437,320,113,278, 11,243, 87,317, 36, 93,496, 27,
          487,446,482, 41, 68,156,457,131,326,403,339, 20, 39,115,442,124,
          475,384,508, 53,112,170,479,151,126,169, 73,268,279,321,168,364,
          363,292, 46,499,393,327,324, 24,456,267,157,460,488,426,309,229,
          439,506,208,271,349,401,434,236, 16,209,359, 52, 56,120,199,277,
          465,416,252,287,246,  6, 83,305,420,345,153,502, 65, 61,244,282,
          173,222,418, 67,386,368,261,101,476,291,195,430, 49, 79,166,330,
          280,383,373,128,382,408,155,495,367,388,274,107,459,417, 62,454,
          132,225,203,316,234, 14,301, 91,503,286,424,211,347,307,140,374,
           35,103,125,427, 19,214,453,146,498,314,444,230,256,329,198,285,
           50,116, 78,410, 10,205,510,171,231, 45,139,467, 29, 86,505, 32,
           72, 26,342,150,313,490,431,238,411,325,149,473, 40,119,174,355,
          185,233,389, 71,448,273,372, 55,110,178,322, 12,469,392,369,190,
            1,109,375,137,181, 88, 75,308,260,484, 98,272,370,275,412,111,
          336,318,  4,504,492,259,304, 77,337,435, 21,357,303,332,483, 18,
           47, 85, 25,497,474,289,100,269,296,478,270,106, 31,104,433, 84,
          414,486,394, 96, 99,154,511,148,413,361,409,255,162,215,302,201,
          266,351,343,144,441,365,108,298,251, 34,182,509,138,210,335,133,
          311,352,328,141,396,346,123,319,450,281,429,228,443,481, 92,404,
          485,422,248,297, 23,213,130,466, 22,217,283, 70,294,360,419,127,
          312,377,  7,468,194,  2,117,295,463,258,224,447,247,187, 80,398,
          284,353,105,390,299,471,470,184, 57,200,348, 63,204,188, 33,451,
           97, 30,310,219, 94,160,129,493, 64,179,263,102,189,207,114,402,
          438,477,387,122,192, 42,381,  5,145,118,180,449,293,323,136,380,
           43, 66, 60,455,341,445,202,432,  8,237, 15,376,436,464, 59,461
      };
      ylist = s9list[[FromDigits[xlist, 2]+1]],
      (** else **)
      (S9::sizeerror = "Eingabeliste war ungleich neun Elemente!";
       Message[S9::sizeerror])
    ];

    (** Ausgabe als neun-stellige Binärzahl**)
    IntegerDigits[ylist, 2, 9]
]
```

D.2 kasumi_use.m

```
(****************************** Kasumi Usage ******************************)

f8::usage = "f8[plain_, CK_] realisiert die UMTS-Verschlüsselungsfunktion f8,
die ansich ein Schlüsselstromgenerator ist. Eingabe ist der Klartextstrom plain_
als binäre Liste (z.B. {0,1,1,0,1,0}) und der geheime 128 Bit Schlüssel CK_.
Ausgegeben wird der Schlüsselstrom als binäre Liste.\n
\n
f8[plain_, CK_, IV_] realisiert die UMTS-Verschlüsselungsfunktion f8, die
ansich ein Schlüsselstromgenerator ist. Eingabe ist der Klartextstrom plain_
als binäre Liste (z.B. {0,1,1,0,1,0}). Der geheime 128 Bit Schlüssel CK_ und
der 64 Bit große Initialisierungsvektor IV_, der sich in der Praxis aus den
```

```
Parametern COUNT-C, BEARER, DIRECTION und 26 Null-Bit zusammensetzt. Ausgegeben
wird der Schlüsselstrom als binäre Liste.\n
\n
f8[plain_, CK_, COUNT_, BEARER_, DIRECTION_] realisiert die
UMTS-Verschlüsselungsfunktion f8, die ansich ein Schlüsselstromgenerator ist.
Eingabe ist der Klartextstrom plain_ als binäre Liste (z.B. {0,1,1,0,1,0}),
der geheime 128 Bit Schlüssel CK_ und die Parameter des Initialisierungsvektors
COUNT_ (32 Bit), BEARER_ (5 Bit), DIRECTION_ (1 Bit)."
       (*************************************************************)

f9::usage = "f9[msg_, IK_] realisiert die UMTS-Integritätsfunktion f9, die von
einer beliebig großen Nachricht msg_ als binäre Liste (z.B. {0,1,1,0,1,0}) einen
Message Authentication Code (MAC) im CBC-Modus berechnet. Zweiter Eingabeparameter
ist der geheime 128 Bit Schlüssel IK_. Ausgegeben wird der MAC als 32 Bit große
binäre Liste.\n
\n
f9[msg_, IK_, COUNT_, DIRECTION_] realisiert die UMTS-Integritätsfunktion f9,
die von einer beliebig großen Nachricht msg_ als binäre Liste (z.B. {0,1,1,0,1,0})
einen Message Authentication Code (MAC) im CBC-Modus berechnet. Weitere
Eingabeparameter sind der geheime 128 Bit Schlüssel IK_, die 32 Bit große
Sequenznummer COUNT_ und der 1 Bit große Parameter DIRECTION_. Die Zufallszahl
FRESH wird bei jedem Aufruf neu als Pseudozufallszahl generiert. Ausgegeben wird
der MAC als 32 Bit große binäre Liste.\n
\n
f9[msg_, IK_, COUNT_, DIRECTION_, FRESH_]realisiert die
UMTS-Integritätsfunktion f9, die von einer beliebig großen Nachricht
msg_ als binäre Liste
(z.B. {0,1,1,0,1,0}) einen Message Authentication Code (MAC) im
CBC-Modus berechnet. Weitere Eingabeparameter sind der geheime 128 Bit
Schlüssel IK_, die 32 Bit große Sequenznummer COUNT_, der 1 Bit große
Parameter DIRECTION_ und die 32 Bit große Zufallszahl
FRESH_. Ausgegeben wird der MAC als 32 Bit große binäre Liste."
       (*************************************************************)

Kasumi::usage = "Kasumi[in_, K_] realisiert die KASUMI Blockchiffre. Eingabeblock
in_ ist eine Liste mit 64 binären Elementen und der 128 Bit Schlüssel K_.
Ausgegeben wird eine Liste im selben Format wie der Eingabeblock.\n
\n
Kasumi[in_, K_, roundout_] realisiert ebenfalls die KASUMI Blockchiffre. Es kann
noch ein zusätzlicher Parameter roundout_ (1 oder 0) angegeben werden, der
bestimmt, ob die Rundenergebnisse mit ausgegeben werden. Die Ausgabe ist eine
Liste von Listen mit je 64 binären Elementen. (Z.B. {{1,0,1,...,0},{0,1,1,...,1}})"
       (*************************************************************)

FL::usage = "FL[in_, round_] realisiert die KASUMI-Unterfunktion FL. Der 32 Bit
 Eingabeblock als Liste in_ wird mit dem globalen 32 Bit Teilschlüssel KL[round_]
  verknüpft. Als Ausgabe erhält man wieder einen 32 Bit Ausgabeblock als Liste."
       (*************************************************************)

FO::usage = "FO[in_] realisiert die KASUMI-Unterfunktion FO. Eingabeblock in_
ist eine Liste mit 32 binären Elementen und die Rundenzahl round_. Intern werden
die zwei globalen Schlüsselblöcke KO_ und KI_ je als Liste mit 48 binären
Elementen verwendet. Ausgegeben wird eine Liste im selben Format wie der
Eingabeblock."
       (*************************************************************)

FI::usage = "FI[in_, round_, roundFO_] realisiert die KASUMI-Unterfunktion FI.
Eingabeblock ist eine Liste mit 16 binären Elementen, die KASUMI-Rundennummer
round_ und die Rundennummer von FO roundFO_. Der Teilschlüsselblock KI ist als
```

```
globale Liste durch die KeySchedule[]-Funktion vorhanden. Ausgegeben wird eine
Liste im selben Format wie die Eingabe."
       (*************************************************************)

KeySchedule::usage = "KeySchedule[K_] realisiert die KASUMI KeySchedule zur
Teilschlüsselgenerierung aus dem 128 Bit Schlüssel. Eingabe ist eine Liste K
mit 128 binären Elementen. Es werden Variablen angelegt, die die entsprechenden
Teilschlüssel enthalten: KL, KO, KI."
       (*************************************************************)

S7::usage = "S7[xlist_] macht einen Table-Lookup in der KASUMI S-Box S7 und
liefert den Wert an der Stelle xlist_ zurück. Als Eingabe wird eine binäre
Liste mit sieben Stellen erwartet; ebenso groß ist die Ausgabe."
       (*************************************************************)

S9::usage = "S9[xlist_] macht einen Table-Lookup in der KASUMI S-Box S9 und
liefert den Wert an der Stelle xlist_ zurück. Als Eingabe wird eine binäre
Liste mit neun Stellen erwartet; ebenso groß ist die Ausgabe."
       (*************************************************************)

(******************************** !Usage ***********************************)

(***************************** Messages ************************************)

f8::sizeerror = "Bitte die Eingabegrößen überprüfen. Siehe ?f8 für Infos."

f9::sizeerror = "Bitte die Eingabegrößen überprüfen. Siehe ?f9 für Infos."

Kasumi::sizeerror = "Bitte die Eingabegrößen überprüfen. Siehe ?Kasumi für Infos."

(***************************** !Messages ***********************************)
```

D.3 crypto.m

```
(***************************** Crypto Package ******************************)
(**
Author: Axel Bolta, FH Ravensburg-Weingarten
Date: 2003/07/06

Last update: 2003/08/11
**)



BeginPackage["Crypto`"]

(******************************** Usage ************************************)

(************************ Externe Packete einlesen *************************)

<<kasumi_use.m
<<des_use.m

(************************** Interne Definitionen ***************************)

BinPlus::usage = "BinPlus[bin_, num_, size_] ist eine Additionsfunktion auf
binäre Listen. Die Binärzahl bin_ wird in dezimal gewandelt und mit der
```

```
Dezimalzahl num_ addiert. Das Ergebnis wird als Binärzahl mit size_ Bit
ausgegeben. Hierbei wird davon ausgegangen, dass der Anwender weiß, wieviel
Bit die Ausgabe mindestens haben muss."
       (*************************************************************)

SplitBlock::usage = "SplitBlock[nlist_, hleft_, hright_] teilt die Liste
nlist_ in zwei Teile mit den hleft_ (z.B. 7) linken Elementen und den
hright_ (z.B. 9) rechten Elementen von nlist_. Die Rückgabe ist ein Array
mit zwei Unterlisten."
       (*************************************************************)

BlockRead::usage = "BlockRead[b_, nfile_] liest eine gegebene Datei nfile_
byteweise (als Charactercode) in eine Liste und wandelt diese in ein Array;
dessen Unterlisten haben je b_ mal 8 Elemente und zwar die jeweiligen
Binärwerte der Bytecodes. Als Ausgabe erhält man also eine Liste mit b_*8
großen Bitblöcken der Datei. Sollte die Summe der Bytes von nfile_ nicht
ohne Rest durch 8 teilbar sein, werden Null-Bytes aufgefüllt. Diese Funktion
ist nützlich, will man eine Datei mit einer Blockchiffre verschlüsseln."
       (*************************************************************)

Truncate::usage = "Truncate[nlist_, nbit_, pos_] verkürzt die Liste nlist_
um nbit_ an der Position pos_; der String pos_ kann entweder \"h\" für high
oder \"l\" für low sein. Z.B. Truncate[mylist, 3, \"h\"] bedeutet, verwerfe
die drei höchstwertigen, also linken Stellen von mylist."
       (*************************************************************)

ZeroExtend::usage = "ZeroExtend[nlist_, nbit_, pos_] hängt nbit_ Null-Bit an
die Position pos_ der Liste nlist_ an; der String pos_ kann entweder \"h\"
für high oder \"l\" für low sein. Z.B. ZeroExtend[mylist, 3, \"h\"] bedeutet,
 hänge drei Null-Bit vor das höchstwertige Bit."
       (*************************************************************)

HammingD::usage = "HammingD[list1_, list2_] gibt die Hamming-Distanz der zwei
 gleichgroßen binären Listen list1_ und list_2 als Dezimalzahl aus."
       (*************************************************************)

ListToString::usage = "ListToString[list_] wandelt die Elemente einer Liste
list_ zu einem fortlaufenden String um."
       (*************************************************************)

BinToHex::usage = "BinToHex[in_] wandelt die binäre Eingabeliste in_ in eine
Hexadezimalzahl um."
        (*************************************************************)

StringToBin::usage = "StringToBin[str_] wandelt den String str_ Byteweise
(Charactercode) in eine Binärliste um."
       (*************************************************************)

BinToString::usage = "BinToString[lst_] wandelt die Binärliste lst_ Byteweise
 in einen Character-String um."
       (*************************************************************)


(****************************** !Usage ***********************************)

(***************************** Messages **********************************)

       (*************************************************************)
```

```
HammingD::sizeerror =
"Bitte die Eingabegrößen überprüfen. Siehe ?HammingD für Infos."
       (*****************************************************************)

BinToHex::sizeerror =
"Bitte die Eingabegrößen überprüfen. Siehe ?BinToHex für Infos."
       (*****************************************************************)

BinToString::sizeerror =
"Bitte die Eingabegrößen überprüfen. Siehe ?BinToString für Infos."
       (*****************************************************************)



(****************************** !Messages ***********************************)
(****************************************************************************)







Begin["`Private`"]

(************************ Externe Packete einlesen **************************)

<<kasumi_pack.m
<<des_pack.m

(*************************** Interne Definitionen ***************************)

(****************************************************************************)
(****************************************************************************)
(****************************************************************************)
(** BinPlus
Author: Axel Bolta, Date: 2003/06/18, Last update: 2003/06/18**)
(****************************************************************************)


BinPlus[bin_, num_, size_] := Module[
    {
        sum
    },
    (** Wandeln in dezimal und num addieren **)
    sum = FromDigits[bin, 2] + num;
    (** Zurückwandeln in Binärdarstellung, size_ Bit groß **)
    sumbin = IntegerDigits[sum, 2, size]

]


(****************************************************************************)
(****************************************************************************)
(****************************************************************************)
(** SplitBlock
Author: Axel Bolta, Date: 2003/06/12, Last update: 2003/06/12**)
(****************************************************************************)
```

```
SplitBlock[nlist_, hleft_, hright_] :=
          {Take[nlist, hleft], Take[nlist, -hright]}


(*************************************************************************)
(*************************************************************************)
(*************************************************************************)
(** BlockRead
Author: Axel Bolta, Date: 2003/06/12, Last update: 2003/06/12**)
(*************************************************************************)
BlockRead[b_, nfile_] := Module[
    {mylist,i,j,len},

    (**
     Lese die Datei "nfile" byte-weise in eine Liste "mylist"
     und wandle die Listenelemente von "mylist" von Dezimal
     in Binärdarstellung mit acht Stellen.
     **)
    mylist = IntegerDigits[ReadList[nfile, Byte], 2, 8];

    (**
     Prüfe jede Element-Liste der Liste "mylist", ob sie 8 Elemente besitzt,
     wenn nicht füge Nullen vorne an, bis 8.

    For[i=1, i <= Length[mylist], i++,
       While[Length[mylist[[i]]] != 8,
            mylist[[i]] = Prepend[mylist[[i]],0]
       ]
    ];

     Erfüllt schon IntegerDigits[] mit drittem Parameter
     **)

    (**
     Fülle die Anzahl der Bytes auf eine durch b_ ganzz. teilbare Summe auf.
     **)
    While[Mod[len=Length[mylist],b] != 0,
         mylist = Append[mylist,Table[0,{8}]]
    ];

    (**
     Schleife, um je b_ Listen aneinanderzuhängen.
     **)
    For[i=1, i<=(len/b), i++,
       For[j=1, j<b, j++,
          mylist[[i]] = Join[mylist[[i]], mylist[[i+1]]];
          mylist = Delete[mylist, i+1]]
    ];

    (**
     Ausgabe
     **)
    mylist

]


(*************************************************************************)
(*************************************************************************)
```

```
(*************************************************************************)
(** Truncate
Author: Axel Bolta, Date: 2003/06/12, Last update: 2003/06/12**)
(*************************************************************************)
Truncate[nlist_, nbit_, pos_] := Module[
    {outlist},

    Switch[pos,
          (** case 1**)
          "h", outlist = Drop[nlist, nbit],
          (** case 2 **)
          "l", outlist = Drop[nlist, -nbit],
          (** default **)
          _,
          (Truncate::inputerror = "Dritter Parameter muss \"h\" oder \"l\"
sein!";Message[Truncate::inputerror])
        ];

    outlist
]


(*************************************************************************)
(*************************************************************************)
(*************************************************************************)
(**  ZeroExtend
Author: Axel Bolta, Date: 2003/06/12, Last update: 2003/06/12**)
(*************************************************************************)
ZeroExtend[nlist_, nbit_, pos_] := Module[
    {outlist},

    Switch[pos,
          (** case 1**)
          "h", outlist = Prepend[nlist, Table[0, {nbit}]],
          (** case 2 **)
          "l", outlist = Append[nlist, Table[0, {nbit}]],
          (** default **)
          _,
          (ZeroExtend::inputerror = "Dritter Parameter muss \"h\" oder \"l\"
sein!";Message[ZeroExtend::inputerror])
        ];

    Flatten[outlist]
]


(*************************************************************************)
(*************************************************************************)
(*************************************************************************)
(** HammingD
Author: Axel Bolta, Date: 2003/06/20, Last update: 2003/06/20**)
(*************************************************************************)
HammingD[list1_, list2_] := Module[
    {},
    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[list1] != Length[list2])
      || (Length[Select[list1, # != 1 && # !=0 &]] != 0)
```

```
      || (Length[Select[list2, # != 1 && # !=0 &]] != 0),

      Message[HammingD::sizeerror],

      (** Else **)
      Apply[Plus, BitXor[list1, list2]]
    ]
]


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** ListToString
Author: Axel Bolta, Date: 2003/06/24, Last update: 2003/06/24**)
(*****************************************************************************)

ListToString[list_] := Module[
        {len, string},

        len = Length[list];
        string = "";

        For[i = 1, i <= len, i++,
                string = StringJoin[string, ToString[list[[i]]]]
        ];

        string
]


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** BinToHex
Author: Axel Bolta, Date: 2003/07/06, Last update: 2003/07/06**)
(*****************************************************************************)

BinToHex[in_] := Module[
    {
        s
    },

    (**
     Prüfen der Eingabegrößen auf Inhalt außer 1 bzw. 0
     **)
    If[Length[Select[in, # != 1 && # !=0 &]] != 0,

      Message[BinToHex::sizeerror],

      (** Else **)
      s = ToString[BaseForm[FromDigits[in, 2], 16]];
      StringDrop[s, {Part[StringPosition[s, "\n"], 1, 1], -1}]

    ]      (** If! **)
](** Module! **)
```

```
(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** StringToBin
Author: Axel Bolta, Date: 2003/06/30, Last update: 2003/06/30**)
(*****************************************************************************)

StringToBin[str_] := Module[
        {},

        Flatten[IntegerDigits[ToCharacterCode[ToString[str]], 2, 8]]
]


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** BinToString
Author: Axel Bolta, Date: 2003/06/30, Last update: 2003/06/30**)
(*****************************************************************************)

BinToString[lst_] := Module[
        {len, blocks, temp, out},

        (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[Select[lst, # != 1 && # !=0 &]] != 0),

      Message[BinToString::sizeerror],

      (** Else **)
      len = Length[lst];
      blocks = Ceiling[len/8];
      (** Geradzahlig durch acht teilbar?, sonst mit 0 padden **)
      If[Mod[len, 8] != 0,
        temp = PadLeft[lst, blocks * 8],
        temp = lst;
      ];

      temp = Partition[temp, 8];
      out = {};
      For[i = 1, i <= blocks, i++,
        (** umwandeln in Character **)
        AppendTo[out, FromCharacterCode[FromDigits[temp[[i]], 2]]]
      ];
          StringJoin[out]
    ]
]




(*****************************************************************************)

End[]

EndPackage[]
```

D.4 des_pack.m

```
(***************************** DES Functions ***************************)
(**
Author: Axel Bolta
Date: 2003/07/06

Last update: 2003/07/07
**)
(***********************************************************************)
(***********************************************************************)
(***********************************************************************)
(** DES V1 ohne Rundenausgabe **)
(***********************************************************************)

DES[in_, key_, mode_] := Module[
    {
     K, insplit, inleft, inright,
     newleft, i, round,
     out
    },

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[in]!= 64) || (Length[key]!= 64)
      || (Length[Select[in, # != 1 && # !=0 &]] != 0)
      || (Length[Select[key, # != 1 && # !=0 &]] != 0),

      Message[DES::sizeerror],

      (** Else **)

      (** Erzeugen der Teilschlüssel mit der DES KeySchedule **)
      K = DESKeys[key];

      (**
       Aufteilen des permutierten Blocks in zwei 32-Bit-Hälften.
       **)
      insplit = SplitBlock[DESIp[in], 32, 32];
      inleft = insplit[[1]];
      inright = insplit[[2]];

      (**
       16 mal die Rundenfunktion f mit den jeweiligen Rundenschlüseln
       durchlaufen je nach mode rueckwaerts oder vorwaerts
       **)
      Switch[mode,
            (** Verschluesseln, also vorwaerts **)
            0,
            For[i=1, i<=16, i++,
               round = i;
               newleft = inright;
               (** neue rechte Hälfte **)
               inright = BitXor[inleft, DESf[inright, K[[round]]]];
               (** neue linke Hälfte **)
               inleft = newleft
            ],
            (** Entschluesseln, also rueckwaerts **)
```

```
            1,
            For[i=16, i>=1, i--,
               round = i;
               newleft = inright;
               (** neue rechte Hälfte **)
               inright = BitXor[inleft, DESf[inright, K[[round]]]];
               (** neue linke Hälfte **)
               inleft = newleft
            ],
            (** default ist vorwaerts **)
            _,
            For[i=1, i<=16, i++,
               round = i;
               newleft = inright;
               (** neue rechte Hälfte **)
               inright = BitXor[inleft, DESf[inright, K[[round]]]];
               (** neue linke Hälfte **)
               inleft = newleft
            ] (** For! **)
      ]; (** Switch! **)


      (** Hälften nochmals vertauschen und inverse Eingangspermutation **)
      DESIpInv[Join[inright, inleft]]

    ]      (** If! **)
](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
(** DES V2 mit Rundenausgabe ohne Eingangs- und Schlusspermutation **)
(*****************************************************************************)

DES[in_, key_, mode_, roundout_] := Module[
    {
     K, insplit, inleft, inright,
     newleft, i, round,
     out
    },

     (** out formatieren **)
     out = Table[{}, {16}];

    (**
     Prüfen der Eingabegrößen auf Länge und Inhalt außer 1 bzw. 0
     **)
    If[(Length[in]!= 64) || (Length[key]!= 64)
      || (Length[Select[in, # != 1 && # !=0 &]] != 0)
      || (Length[Select[key, # != 1 && # !=0 &]] != 0),

      Message[DES::sizeerror],

      (** Else **)

      (** Erzeugen der Teilschlüssel mit der DES KeySchedule **)
      K = DESKeys[key];
```

```
(**
 Aufteilen des Blocks in zwei 32-Bit-Hälften.
 **)
insplit = SplitBlock[in, 32, 32];
inleft = insplit[[1]];
inright = insplit[[2]];

(**
 16 mal die Rundenfunktion f mit den jeweiligen Rundenschlüseln
 durchlaufen je nach mode rueckwaerts oder vorwaerts
 **)
Switch[mode,
      (** Verschluesseln, also vorwaerts **)
      0,
      For[i=1, i<=16, i++,
         round = i;
         newleft = inright;
         (** neue rechte Hälfte **)
         inright = BitXor[inleft, DESf[inright, K[[round]]]];
         (** neue linke Hälfte **)
         inleft = newleft;
         If[roundout == 0,
           (** nur die letzte Runde ausgeben**)
           If[round == 16,
             out = {{}};
             out[[1]] = Join[inright, inleft]
           ], (** If! **)

           (** Else, Rundenergebnisse eintragen **)
           If[round == 16,
             out[[round]] = Join[inright, inleft]
           ,
           (** Else **)
           out[[round]] = Join[inleft, inright]
           ] (** If! **)
         ] (** If! **)
      ],
      (** Entschluesseln, also rueckwaerts **)
      1,
      For[i=16, i>=1, i--,
         round = i;
         newleft = inright;
         (** neue rechte Hälfte **)
         inright = BitXor[inleft, DESf[inright, K[[round]]]];
         (** neue linke Hälfte **)
         inleft = newleft;
         If[roundout == 0,
           (** nur die letzte Runde ausgeben**)
           If[round == 1,
             out = {{}};
             out[[1]] = Join[inright, inleft]
           ], (** If! **)

           (** Else, Rundenergebnisse eintragen **)
           If[round == 1,
             out[[round]] = Join[inright, inleft]
           ,
           (** Else **)
           out[[round]] = Join[inleft, inright]
```

```
                ] (** If! **)
              ] (** If! **)
           ],
           (** default ist vorwaerts **)
           _,
           For[i=1, i<=16, i++,
              round = i;
              newleft = inright;
              (** neue rechte Hälfte **)
              inright = BitXor[inleft, DESf[inright, K[[round]]]];
              (** neue linke Hälfte **)
              inleft = newleft;
              If[roundout == 0,
                (** nur die letzte Runde ausgeben**)
                If[round == 16,
                  out = {{}};
                  out[[1]] = Join[inright, inleft]
                ], (** If! **)

                (** Else, Rundenergebnisse eintragen **)
                If[round == 16,
                  out[[round]] = Join[inright, inleft]
                ,
                (** Else **)
                out[[round]] = Join[inleft, inright]
                ] (** If! **)
              ] (** If! **)
           ] (** For! **)
      ]; (** Switch! **)

      out

    ]      (** If! **)
](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(** DESf **)
(*****************************************************************************)

DESf[in_, key_] := Module[
    {
     temp
    },

     (** Expansionspermutation und XOR-Verknüpfung mit Schlüssel **)
     temp = BitXor[DESExp[in], key];

     (** S-Box-Substitution und anschließende P-Box-Permutation **)
     DESPbox[DESSbox[temp]]

](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(** DESExp **)
(*****************************************************************************)
```

```
DESExp[in_] := Module[
    {
     i, out
    },

     out = Table[ 0, {48}];

     (** Selektions-Tabelle der Expansionspermutation **)
     etab = {
         32,  1,  2,  3,  4,  5,
          4,  5 , 6,  7,  8,  9,
          8,  9, 10, 11, 12, 13,
         12, 13, 14, 15, 16, 17,
         16, 17, 18, 19, 20, 21,
         20, 21, 22, 23, 24, 25,
         24, 25, 26, 27, 28, 29,
         28, 29, 30, 31, 32,  1
     };

     (** 48 Bit binäre Ausgabeliste erzeugen **)
     For[i = 1, i <= 48, i++,
        out[[i]] = in[[etab[[i]]]]
     ];

     out

](** Module! **)


(*************************************************************************)
(*************************************************************************)
(** DESIp **)
(*************************************************************************)

DESIp[in_] := Module[
    {
     i, out
    },

     out = Table[ 0, {64}];

     (** Selektions-Tabelle der Eingangspermutation **)
     iptab = {
         58, 50, 42, 34, 26, 18, 10, 2,
         60, 52, 44, 36, 28, 20, 12, 4,
         62, 54, 46, 38, 30, 22, 14, 6,
         64, 56, 48, 40, 32, 24, 16, 8,
         57, 49, 41, 33, 25, 17,  9, 1,
         59, 51, 43, 35, 27, 19, 11, 3,
         61, 53, 45, 37, 29, 21, 13, 5,
         63, 55, 47, 39, 31, 23, 15, 7
     };

     (** 64 Bit binäre Ausgabeliste erzeugen **)
     For[i = 1, i <= 64, i++,
        out[[i]] = in[[iptab[[i]]]]
     ];
```

```
     out

](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(** DESIpInv **)
(*****************************************************************************)

DESIpInv[in_] := Module[
    {
     i, out
    },

     out = Table[ 0, {64}];

     (** Selektions-Tabelle der inversen Eingangspermutation **)
     ipinvtab = {
         40, 8, 48, 16, 56, 24, 64, 32, 39, 7, 47, 15, 55, 23, 63, 31,
         38, 6, 46, 14, 54, 22, 62, 30, 37, 5, 45, 13, 53, 21, 61, 29,
         36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43, 11, 51, 19, 59, 27,
         34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49, 17, 57, 25
     };

     (** 64 Bit binäre Ausgabeliste erzeugen **)
     For[i = 1, i <= 64, i++,
        out[[i]] = in[[ipinvtab[[i]]]]
     ];

     out

](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(** DESKeyPerm **)
(*****************************************************************************)

DESKeyPerm[in_] := Module[
    {
     i, out
    },

     out = Table[ 0, {56}];

     (** Selektions-Tabelle der Schlüsselpermutation **)
     keyperm = {
         57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18,
         10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36,
         63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22,
         14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 28, 20, 12, 4
     };

     (** 56 Bit durch Schlüsselpermutation **)
     For[i = 1, i <= 56, i++,
        out[[i]] = in[[keyperm[[i]]]]
     ];
```

```
     out

](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(** DESKeys **)
(*****************************************************************************)

DESKeys[in_] := Module[
    {
        i, split, temp, K
    },

     K = Table[{}, {16}];

     (** 56 Bit durch Schlüsselpermutation **)
     temp = DESKeyPerm[in];

     split = SplitBlock[temp, 28, 28];

     For[i = 1, i <= 16, i++,
        (** LeftShift Operationen je nach Rundenzahl um eins oder zwei **)
        If[(i == 1) || (i == 2) || (i == 9) || (i == 16),
          split[[1]] = RotateLeft[split[[1]]];
          split[[2]] = RotateLeft[split[[2]]],
          (** Else **)
          split[[1]] =  RotateLeft[split[[1]], 2];
          split[[2]] = RotateLeft[split[[2]], 2]
        ];
        (** 48 Bit durch Kompressionspermutation als Teilschlüssel **)
        K[[i]] = DESComprPerm[Flatten[split]]
     ];

     K

](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(** DESComprPerm **)
(*****************************************************************************)

DESComprPerm[in_] := Module[
    {
     i, temp, out
    },

     out = Table[ 0, {48}];

     (** Selektions-Tabelle der Kompressionspermutation, -> 48 Bit **)
     comprperm = {
         14, 17, 11, 24,  1,  5,  3, 28, 15,  6, 21, 10,
         23, 19, 12,  4, 26,  8, 16,  7, 27, 20, 13,  2,
         41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33, 48,
         44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29, 32
```

```
     };

     (** 48 Bit durch Kompressionspermutation **)
     For[i = 1, i <= 48, i++,
        out[[i]] = in[[comprperm[[i]]]]
     ];

     out

](** Module! **)


(*************************************************************************)
(*************************************************************************)
(** DESPBox **)
(*************************************************************************)

DESPbox[in_] := Module[
    {
     i, out
    },

     out = Table[ 0, {32}];

     (** Die P-Box **)
     pbox = {
         16,  7, 20, 21,
         29, 12, 28, 17,
          1, 15, 23, 26,
          5, 18, 31, 10,
          2,  8, 24, 14,
         32, 27,  3,  9,
         19, 13, 30,  6,
         22, 11,  4, 25
     };

     (** 32 Bit binäre Ausgabeliste erzeugen **)
     For[i = 1, i <= 32, i++,
        out[[i]] = in[[pbox[[i]]]]
     ];

     out

](** Module! **)


(*************************************************************************)
(*************************************************************************)
(** DESSbox **)
(*************************************************************************)

DESSbox[in_] := Module[
    {
        split, row, col, sbox, i, out
    },

     out = {};

     (** Liste der acht S-Boxen **)
```

```
sbox = {
    (** S-Box 1 **)
    {
        {14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},
        {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},
        {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},
        {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}
    },
    (** S-Box 2 **)
    {
        {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10},
        {3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5},
        {0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15},
        {13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9}
    },
    (** S-Box 3 **)
    {
        {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8},
        {13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1},
        {13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7},
        {1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12}
    },
    (** S-Box 4 **)
    {
        {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15},
        {13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9},
        {10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4},
        {3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14}
    },
    (** S-Box 5 **)
    {
        {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9},
        {14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6},
        {4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14},
        {11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3}
    },
    (** S-Box 6 **)
    {
        {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11},
        {10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8},
        {9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6},
        {4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13}
    },
    (** S-Box 7 **)
    {
        {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1},
        {13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6},
        {1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2},
        {6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12}
    },
    (** S-Box 8 **)
    {
        {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7},
        {1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2},
        {7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8},
        {2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}
    }
};
```

```
    (** Aufteilen des Eingabeblocks in acht Sechserblöcke **)
    split = Partition[in, 6];

    (**
     das erste und sechste Bit als Dezimalzahl ergibt die Zeile einer S-Box;
     Bits zwei bis fünf als Dezimalzahl ergibt die Spalte einer S-Box;
     die Zahl an der entsprechenden Stelle wird als 4 Bit binär in die
     Ausgabeliste geschrieben.
     **)

    For[i = 1, i <= 8, i++,
       (** je +1, da bei Mathematica Listen bei 1 anfangen, nicht bei 0!! **)
       row = FromDigits[{split[[i, 1]], split[[i, 6]]}, 2] + 1;
       col = FromDigits[Take[split[[i]], {2, 5}], 2] + 1;
       AppendTo[out, IntegerDigits[sbox[[i, row, col]], 2, 4]]
    ];

    Flatten[out]

](** Module! **)


(*****************************************************************************)
(*****************************************************************************)
(*****************************************************************************)
```

D.5 des_use.m

```
(******************************** DES Usage *********************************)
(**
Author: Axel Bolta
Date: 2003/07/06

Last update: 2003/08/12
**)
(*****************************************************************************)
       (************************************************************)

DES::usage = "DES[in_, key_, mode_] realisiert die DES Blockchiffre.
Eingabe ist in_, eine Liste mit 64 binären Elementen und der 64 Bit
Schlüssel key_. Ausgegeben wird eine Liste im selben Format wie der
Eingabeblock. Der Parameter mode_ bestimmt, ob DES zur Ver- (mode = 0 und
default) oder Entschluesselung (mode = 1) verwendet werden soll.\n
\n
DES[in_, key_, mode_, roundout_] gibt zusaetzlich bei roundout = 1 die
Rundenergebnisse aus. roundout = 0 gibt nur die letzte Runde aus. Generell
wird bei dieser Version auf die Eingangs- und Schlusspermutation (Ip und
IpInv) verzichtet."

       (************************************************************)

DESf::usage = "DESf[in_, key_] realsiert die Funktion f von DES. Die 32 Bit
große binäre Eingabeliste in_ wird mit dem 48 Bit Schlüssel key_
verarbeitet. Eine 32 Bit große binäre Liste wird ausgegeben."
```

```
       (*************************************************************)

DESExp::usage = "DESExp[in_] ist die Expansionspermutation von DES. Die 32
Bit große binäre Eingabeliste in_ wird entsprechend einer Tabelle permutiert
und auf 48 Bit erweitert."

       (*************************************************************)

DESIp::usage = "DESIp[in_] ist die Eingangspermutation von DES. Die 64 Bit
große binäre Eingabeliste in_ wird entsprechend einer Tabelle permutiert.
Ausgegeben wird eine Liste im selben Format."

       (*************************************************************)

DESIpInv::usage = "DESIpInv[in_] ist die inverse Eingangspermutation von DES.
 Die 64 Bit große binäre Eingabeliste in_ wird entsprechend einer Tabelle
 permutiert. Ausgegeben wird eine Liste im selben Format."

       (*************************************************************)

DESKeyPerm::usage = "DESKeyPerm[in_] realisiert die Schlüsselpermutation von
DES. Aus dem Schlüssel als 64 Bit große binäre Eingabeliste in_ wird
entsprechend einer Tabelle ein 56 Bit Schlüssel erzeugt."

       (*************************************************************)

DESComprPerm::usage = "DESComprPerm[in_] ist die Komressionspermutation von
DES. Aus der 56 Bit großen binären Eingabeliste in_ wird entsprechend der
Tabelle eine 48 Bit Liste erzeugt."

       (*************************************************************)

DESKeys::usage = "DESKeys[in_] realisiert die Keyschedule von DES. Aus dem
Schlüssel als 64 Bit große binäre Eingabeliste in_ werden 16 48 Bit
Teilschlüssel erzeugt, die als Liste zurückgegeben werden."

       (*************************************************************)

DESPbox::usage = "DESPbox[in_] ist die P-Box-Permutation von DES. Die 32 Bit
große binäre Eingabeliste in_ wird entsprechend einer Tabelle permutiert. Eine
32 Bit binäre Liste wird ausgegeben."

       (*************************************************************)

DESSbox::usage = "DESSbox[in_] ist die S-Box-Substitution von DES. Die 48 Bit
große binäre Eingabeliste in_ wird durch die S-Boxen substituiert. Als Ausgabe
erhält man eine 32 Bit große binäre Liste."

       (*************************************************************)

(****************************** !Usage ***********************************)

(***************************** Messages **********************************)

DES::sizeerror = "Bitte die Eingabegrößen überprüfen. Siehe ?DES für Infos."


(***************************** !Messages *********************************)
```

Literatur

[3GP] 3GPP: GPRS Tunnelling Protocol (GTP) across the Gn and Gp interface. In: *3GPP Technical Secifications* (TS 29.060)

[3GP00] 3GPP: 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Report on the Evaluation of 3GPP Standard Confidentiality and Integrity Algorithms. In: *3GPP Technical Secifications*. V1.0.0. 2000 (TS 33.909)

[3GP01] 3GPP: 3G Security; Cryptographic Algorithm Requirements. In: *3GPP Technical Secifications*. V4.1.0. 2001 (TS 33.105)

[3GP02a] 3GPP: 3G Security; Network Domain Security; MAP application layer security. In: *3GPP Technical Secifications*. V5.1.0. 2002 (TS 33.200)

[3GP02b] 3GPP: 3G Security; Security architecture. In: *3GPP Technical Secifications*. V5.1.0. 2002 (TS 33.102)

[3GP02c] 3GPP: 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification. In: *3GPP Technical Secifications*. V5.0.0. 2002 (TS 35.202)

[3GP02d] 3GPP: 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 1: f8 and f9 Specification. In: *3GPP Technical Secifications*. V5.0.0. 2002 (TS 35.201)

[3GP02e] 3GPP: Technical Specification Group Services and System Aspects; Personalisation of Mobile Equipment (Me); Mobile functionality specification. In: *3GPP Technical Secifications*. V5.0.0. 2002 (TS 22.022)

[3GP02f] 3GPP: Technical Specification Group Terminals; UICC-Terminal Interface; Physical and Logical Characteristics. In: *3GPP Technical Secifications*. V6.1.0. 2002 (TS 31.101)

[3GP02g] 3GPP: Technical Specification Group Terminals; USIM and IC card requirements. In: *3GPP Technical Secifications*. V5.1.0. 2002 (TS 21.111)

[3GP03a] 3GPP: 3G Security; Network Domain Security; IP network layer security. In: *3GPP Technical Secifications*. V6.1.0. 2003 (TS 33.210)

[3GP03b] 3GPP: Radio Access Network; RRC Protcol Specification. In: *3GPP Technical Secifications*. V5.4.0. 2003 (TS 25.331)

[3GP03c] 3GPP: Technical Specification Group Terminals; Security mechanisms for the (U)SIM application toolkit; Stage 2. In: *3GPP Technical Specifications*. V5.6.0. 2003 (TS 23.048)

[AFK+00] Aamodt, Tom E. ; Friisø, Trond ; Køien, Geir ; Langnes, Runar ; Eilertsen, Øivind: *Security in UMTS – Confidentiality*. Norway : Telenor R&D, 2000

[AFK+01] Aamodt, Tom E. ; Friisø, Trond ; Køien, Geir ; Langnes, Runar ; Eilertsen, Øivind: *Security in UMTS – Integrity*. Norway : Telenor R&D, 2001

[AFKL00] Aamodt, Tom E. ; Friisø, Trond ; Køien, Geir ; Langnes, Runar: *UMTS security – Authentication and Key Agreement*. Norway : Telenor R&D, 2000

[Bem02] Jeroen van Bemmel, Gerard H.: Security Aspects of 4G Services / Wireless World Research Forum (WWRF). 2002. – Forschungsbericht

[BP03] B. Preneel, u.a.: Performance of Optimized Implementations of the NESSIE Primitives / NESSIE. 2003. – Forschungsbericht

[BS02] Benker, Thorsten ; Stepping, Christoph: *UMTS*. Weil der Stadt : J. Schlemmbach Fachverlag, 2002. – ISBN 3–935340–07–9

[CTI03] CTIA. *Wireless Payment Processing in Stadiums*. http://www.wow-com.com/solutions/case_studies. 2003

[Dob99] Dobbertin, Hans: Almost Perfect Nonlinear Power Functions on $GF(2^n) : TheWelchCase$. In: *IEEE Transactions on Information Theory* 45 (1999), Nr. 4

[Eil00] Eilertsen, Øivind: *Security in UMTS – The KASUMI algorithm*. Norway : Telenor R&D, 2000

[Ert01] Ertel, Wolfgang: *Angewandte Kryptographie*. München ; Wien : Fachbuchverlag Leipzig im Carl Hanser Verlag, 2001. – ISBN 3–446–21549–2

[ETS99] ETSI/SAGE: Statistical Evaluation of the 3GPP Confidentiality and Integrity Algorithms / 3GPP. 1999. – Forschungsbericht

[Fox97] Fox, Dirk: Der IMSI-Catcher. In: *Datenschutz- und Datensicherheit (DuD)* (1997), Nr. 9

[Glo03] Glossar.de. *Glossar.de homepage*. http://www.glossar.de/glossar/z_java.htm. 2003

[Inc03] Inc., TeleGeography. *TeleGeography homepage - International Traffic and Main Line Growth: Statistics: Resources*. http://www.telegeography.com/resources/statistics/telephony/. 2003

[ISO97] ISO/IEC, International S.: Part1: General. In: *Information technology – Security techniques – Entity authentication*. 2nd. 1997 (ISO/IEC 9798-1:1997(E))

[ISO99] ISO/IEC, International S.: Part4: Mechanisms using a cryptographic check function. In: *Information technology – Security techniques – Entity authentication*. 2nd. 1999 (ISO/IEC 9798-4:1999(E))

[ITU03] ITU. *ICT - Free Statistics Home Page*. http://www.itu.int/ITU-D/ict/statistics. 2003

[Kla02] Klaus, Christopher W.: Wireless LAN Security FAQ / Internet Security Systems Inc. 2002. – Forschungsbericht

[Mat97] Matsui, Mitsuru: New Block Encryption Algorithm MISTY / Mitsubishi Electric Corporation, Information Technology R&D Center. 1997. – Forschungsbericht

[MP92] Mouly, Michel ; Pautet, Marie-Bernadette: *The GSM System for Mobile Communications*. Palaiseau, France : CELL & SYS, 1992. – ISBN 2–9507190–0–7

[Mue03] Muenz, Stefan. *SELFHTML 8.0 homepage*. http://selfhtml.teamone.de/. 2003

[Mur99] Murphy, Sean: Comments by the NESSIE Project on the AES Finalists / NESSIE. 1999. – Forschungsbericht

[NES03] NESSIE. *NESSIE homepage*. http://www.cryptonessie.org. 2003

[NK95] Nyberg, Kaisa ; Knudsen, Lars R.: Provable Security Against a Differential Attack. In: *Journal of Cryptology* 8 (1995), Nr. 1

[Pea03] Pearce, James. *Excited by 3G? Wait for 4G*. http://www.zdnet.com.au/newstech/communications/. 2003

[SAG01] SAGE: 3GPP KASUMI Evaluation Report, Version 2.0 / 3GPP. 2001. – Forschungsbericht

[Sch96] Schneier, Bruce: *Angewandte Kryptographie*. Bonn : Addison-Wesley, 1996. – ISBN 3–89319–854–7

[Spe03] Spengler, Maria L. *Unterrichtseinheit zum Thema Kryptologie - homepage*. http://520015785443-0001.bei.t-online.de/KRYPT/. 2003

[Sti95] Stinson, Douglas R.: *Cryptography: Theory and Practice*. USA : CRC Press, 1995. – ISBN 0–8493–8521–0

[WAS01] Walke, Bernhard ; Althoff, Marc P. ; Seidenberg, Peter: *UMTS – Ein Kurs*. Weil der Stadt : J. Schlemmbach Fachverlag, 2001. – ISBN 3–935340–15–X

[Was03] Wassenaar. *The Wassenaar Arrangement On Export Controls for Conventional Arms and Dual-Use Goods and Technologies*. http://www.wassenaar.org. 2003

[WJ01] Wickham-Jones, Tom: *webMathematica : A User Guide*. USA : Wolfram Research, Inc., 2001

[Wol96] Wolfram, Stephen: *The Mathematica Book*. USA : Wolfram Media, Inc., 1996

[WR03] Wolfram Research, Inc. *Wolfram Research homepage*. http://www.wolfram.com/. 2003