



Studienarbeit

Programmierung einer SIM-Authentifizierung

Irgend ein Untertitel



Name: **Marco Heumann**
Matrikelnummer: 4188528
Kurs: TINF13AI-BI
Studiengang: Angewandte Informatik
Studiengangsleiter: Prof. Dr. C. Bürgy
Betreuer:
Semester: 5. - 6. Semester
Datum:



Ehrenwörtliche Erklärung


Gemäß § 5 Abs. 3 der Studien- und Prüfungsordnung DHBW Technik vom 22.09.2011 versichere ich hiermit, die vorliegende Arbeit selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln verfasst zu haben.


Datum

Marco Heumann

Inhaltsverzeichnis

Ehrenwörtliche Erklärung	III
Inhaltsverzeichnis	IV
Abkürzungsverzeichnis	VII
Abbildungsverzeichnis	IX
Tabellenverzeichnis	X
Quellcodeverzeichnis	XI
1 Einleitung 	1
1.1 Idee zur Arbeit	2
2 Theorie 	3
2.1 Chipkarten	3
2.1.1 Entwicklung	3
2.1.2 Typen	3
2.1.3 SIM-Karten	4
2.2 Mobilfunkstandards	8
2.3 Authentifizierungsvorgang	10
2.3.1 Ablauf	10
2.3.2 Stärken der UMTS Authentifizierung	13
2.4 Milenage Algorithmus	14
2.4.1 Warum Milenage sicher ist	14
2.4.2 Funktionsweise	15
2.5 AES / Rijndael	19
2.5.1 Geschichte	19
2.5.2 Mitbewerber	20
2.5.3 Algebraische endliche Körper	21
2.5.4 Polynome Arithmetik in $GF(2^8)$	24
2.5.5 Funktionsweise	26
2.5.6 Bewertung von AES	31
2.5.7 AES Vizualization Tool	31
2.6 PPP	32
2.6.1 Architektur PPP	33
2.6.2 Architektur PPPoE	36
2.6.3 Roaring Penguin PPPoE	38
2.7 PC/SC	39
2.7.1 Spezifikation und Aufbau der Schnittstelle	39
2.7.2 PCSClite	41
2.8 Die Sprache C	41

2.8.1	Geschichte	41
2.8.2	Prozedurales Programmierparadigma	42
2.8.3	Besonderheiten	42
2.9	Die Sprache Python	44
2.9.1	Grundlagen	44
2.9.2	Motivation zur Nutzung	45
2.10	Bibliotheken	45
2.10.1	pysim	45
2.10.2	osmo-sim-auth	46
2.10.3	pyscard	46
2.11	Ubuntu	47
2.12	Raspberry Pi	47
2.12.1	Raspberry Pi Foundation	47
2.12.2	Hardware Modell B	48
2.12.3	Betriebssystem	48
2.13	VirtualBox	49
2.13.1	Grundlage	49
2.13.2	Hypervisor	50
2.13.3	Motivation	53
2.14	Projektumsetzung	53
2.14.1	Anforderungen	53
2.14.2	Unterstützende Tools	55
3	Tätigkeit 	57
3.1	Erstellen des Netzproviders	57
3.1.1	Ubuntu	57
3.1.2	PPPoE	57
3.1.3	Implementierung AES	59
3.1.4	Implementierung Milenage	62
3.2	Erstellen des UE	65
3.2.1	Aufsetzen des Raspberry Pis	65
3.3	Integration mit PPPoE	66
3.3.1	Server	66
3.3.2	Client	66
3.4	Implementierung der Server-Client Architektur	67
3.4.1	Implementierung des Servers	67
3.4.2	Implementierung des Clients	68
3.4.3	Kommunikationsablauf zwischen Client und Server	70
4	Ergebnis	72
4.1	Funktionsprüfung	72
4.1.1	AES	73
4.1.2	Milenage	73
4.1.3	SIM-Karte und SIM-Kartenleser	74

4.1.4	Client- und Server-Kommunikation	74
4.1.5	PPPoE	74
4.2	Fehler	75
4.2.1	Einrichtung verwendeter Werkzeuge	75
4.2.2	Fehleinschätzungen	75
5	Diskussion	78
5.1	Alternative Vorgehensweisen	78
5.2	Ausblick	78
5.2.1	Verbesserungen	78
5.2.2	Mögliche zukünftige Projekte	79
6	Appendix sections 	81
6.1	Abbildungen	81
6.2	Listings	87
	Literatur	i

Abkürzungsverzeichnis

3GPP	3rd Generation Partnership Project
AES	Advanced Encryption Standard
AMF	Authentication Management Field
APDU	Application Protocol Data Unit
AuC	Authentication Center
AV	Authentifizierungsvektor
BSC	Base Station Controller
BSS	Base Station System
CHAP	Challenge Handshake Authentication Protocol
DES	Data Encryption Standard
DF	Dedicated File
EEPROM	Electrically Erasable Programmable Read-Only Memory
EF	Elementary File
EIR	Equipment Identity Register
GSM	Global System for Mobile communications
HDLC	High Level Data Link Control
HLR	Home Location Register
IANA	Internet Assigned Numbers Authority
ICC	Integrated Chip Card
ICCID	Integrated Circuit Card Identifier
IEEE	Institute of Electrical and Electronics Engineers
IFD	Interface Device
IMEI	International Mobile Station Equipment Identity
IMSI	International Mobile Subscriber Identity
IPCP	Internet Protocol Control Protocol
ISO	Internationale Organisation für Normung
ISP	Internet Service Provider
LAI	Local Area Identity
LAN	Local Area Network
LCP	Link Control Protocol

LTE	Long Term Evolution
MCC	Mobile Country Code
MF	Master File
MNC	Mobile Network Code
MSC	Mobile Switching Center
MSIN	Mobile Subscriber Identification Number
NAT	Network Address Translation
NCP	Network Control Protocol
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OP	Operator Configuration Field
PAP	Password Authentication Protocol
PC/SC	Personal Computer / Smart Card
PIN	Personal Identification Number
PPP	Point To Point Protocol
PPPoE	Point to Point Protocol over Ethernet
PUK	Personal Unlocking Key
RNC	Radio Network Controller
SIM	Subscriber Identity Module
SLIP	Serial Line Internet Protocol
SN	Serving Network
SQN	Sequence Number
ssh	Secure Shell
TMSI	Temporary Mobile Subscriber Identity
tty	tele-typewriter
UE	User Equipment
UICC	Universal Integrated Circuit Card
UMTS	Universal Mobile Telecommunications System -
USIM	Universal Subscriber Identity Module
VMM	Virtual Machine Monitor
VLR	Visitor Location Register

Abbildungsverzeichnis

1	Ablauf der GSM-Authentifizierung	9
2	Sequenzdiagramm über die Kommunikation zwischen SIM-Karte und Au- thentication Center	11
3	Übersicht über die Generierung der Authentifizierungsvektoren	16
4	Schematische Darstellung der Berechnung der Authentifizierungsvektoren	17
5	Aufbau eines PPP-Frames	34
6	Roaring Penguin PPPoE Devicehierarchie	38
7	Architektur des PC/SC-Standards	40
8	Konzept von Pointers in C	43
9	Schematische Darstellung des Rotations-Algorithmus	64
10	Pinbelegung einer Chipkarte	81
11	Filesystemarchitektur einer Chipkarte	81
12	Befehlsklassen einer Chipkarte	82
13	Teilnehmer in Mobilfunknetzen	82
14	Architektur eines Typ-1-Hypervisors	83
15	Architektur eines Typ-2-Hypervisors	83
16	Grafische Darstellung der AES-Verschlüsselung	84
17	Substitutionstabelle für AES-Verschlüsselung	85
18	Aufbauphasen einer PPP-Verbindung	85
19	Architektur der pycard-Bibliothek	86

Tabellenverzeichnis

1	Auf SIM-Karten gespeicherte Informationen	5
2	Ländercodes und Mobilfunkbetreiber	6
3	Prozessarchitektur von VirtualBox	50
4	Eigenschaften der virtuellen Maschine	57
5	Ergebnisse der Funktionsprüfung	72

Listings


1	main.c	87
2	milénage.c	90
3	rijndael.c	95
4	client.py	101

1 Einleitung

”[...] Seit der Einführung der Telefonkarte im Jahr 1985 in Deutschland sind Chipkarten ein Massenartikel. [...] ”[1]

Laut Spitz begann die weite Verbreitung von Chipkarten etwa ein Jahrzehnt nachdem erste Patente zu 'Identitätskarten mit Mikroprozessor' eingereicht wurden. Seit dieser Entwicklung werden Chipkarten in unterschiedlichen Formfaktoren, Funktionsumfang und Einsatzgebieten verwendet.

Chipkarten, die oft auch als SmartCard oder Integrated Circuit Card bezeichnet werden, existieren allgemein zu dem Zweck eine Identität beziehungsweise Berechtigung (z.B. auch in Form von Guthaben) nachzuweisen. Die beiden populärsten Vertreter sind die elektronische Krankenversichertenkarte im Gesundheitsbereich sowie die SIM¹-Karte im Bereich des Mobilfunk.

Eine bestimmte Ausprägung einer Chipkarte wird **dann** einem Mobilfunkvertrags- beziehungsweise Versicherungsteilnehmer ausgeliefert, damit dieser sich über selbige 'ausweisen' kann. Der reine Identitätsnachweis ist allerdings nicht die einzige Funktion, die eine Chipkarte liefern kann. Neben reinen Speicherkarten existieren auch Prozessorkarten, die ber das reine Liefern (meist sensibler) Informationen hinaus, Algorithmen integrieren. Diese Algorithmen dienen unter anderem zur Sicherstellung der Geheimhaltung sensibler Informationen.

Der Ursprung der Chipkarten im Bereich des Mobilfunk liegt bei der 1985 in Deutschland eingeführten Telefonkarte. Sie setzten sich aufgrund ihrer Robustheit und Fälschungssicherheit gegen ebenfalls geprüfte Magnetkarten sowie Lochkarten durch. Als sich Ende der 90er-Jahre der Mobilfunk stark ausbreitete, sank die Nachfrage nach Telefonkarten und wurde durch SIM-Karten abgelöst. Sowohl bei Prepaid- als auch bei gewöhnlichen Telefonverträgen ist es notwendig, dass der Teilnehmer eine Chipkarte vom Provider erhält und diese in sein Mobiltelefon einsetzt.

Diese SIM-Karte ist optional mit einer Zahlenkombination (PIN²) vor Zugriff von Fremden geschützt. Ist die Karte erst freigeschaltet, kann das Mobiltelefon den Authentifizierungsprozess (unter Verwendung der auf der Karte integrierten Kryptoalgorithmen) mit dem Provider initiieren. Nach erfolgreicher Durchführung dieses Prozesses, also Identifi-

¹Subscriber Identity Module

²Personal Identification Number

zierung durch den Provider, ist der Mobilfunknehmer dazu berechtigt, im Rahmen seines Vertrages das Mobilfunknetz zu benutzen.

Welche Applikationen **wie** auf der Prozessorchipkarte laufen ist von Hersteller zu Hersteller unterschiedlich. Eine Standardisierung findet durch die Norm *ISO 7816* statt. Sie definiert vor allem physikalische und elektrische Eigenschaften von Chipkarten.

Mit dieser Grundlage entstand die Idee zu dieser Studienarbeit.

1.1 Idee zur Arbeit

Ziel dieser Studienarbeit ist es, die Freischaltung eines Netzzugangspunktes unter Verwendung einer gebräuchlichen SIM-Karte zu realisieren. Der Vorteil hierbei ist, dass nahezu jeder Vertragsnehmer bei einem Provider für Breitbandanschlüsse über einen Mobilfunkvertrag verfügt. Sogar Senioren besitzen in der Regel ein Handy oder gar Smartphone und somit zwingend auch eine SIM-Karte. Nach einer Umfrage durch Pro Senectute³ im Jahr 2014 sind dies über 70% der ab 65-Jährigen.

Nicht nur als Ergänzung des heimischen Anschlusses, sondern auch als zuverlässige und sichere Lösung für Hotels oder ähnliche Institutionen ist dieses Setup denkbar.

Es soll gezeigt werden, dass dies durch Kombination bereits bestehender technischer Mittel und eigener Implementierung umgesetzt werden kann. Grundlage dieser Umsetzung ist eine von Herrn Prof. Müller bereitgestellte SIM-Karte, deren Geheimnisse bekannt sind. Weiterführend wird ein Kleincomputer (Raspberry Pi⁴) als Endgerät (UE⁵) sowie eine virtuelle Maschine zur Simulation des Netzproviders (AuC⁶) eingesetzt.

Der Raspberry Pi bietet sich besonders für dieses Vorhaben an, da dieser preisgünstig in der Anschaffung und sehr gut in Bezug auf Erweiterbarkeit konstruiert ist. Das Lesen der SIM-Karte wird durch einen entsprechenden Kartenleser via USB-Technologie ermöglicht. Um möglichst nah an der realen Umgebung zu bleiben, soll die Verbindung zwischen UE und AuC über eine PPPoE⁷-Verbindung (PPP⁸ über Ethernet) realisiert werden.

³<https://www.prosenectute.ch/de.html>

⁴<https://www.raspberrypi.org/>

⁵User Equipment

⁶Authentication Center


⁷Point to Point Protocol over Ethernet

⁸Point To Point Protocol

2 Theorie

2.1 Chipkarten

2.1.1 Entwicklung

Wie bereits in der Einleitung erwähnt, startete die Verbreitung  **SIM-Karten** mit Einführung der Telefon- und Krankenversicherungskarten im Jahr 1985. Bereits 6 Jahre später kamen die ersten Karten mit Mikroprozessoren - die SIM-Karten - zum Einsatz. Seit diesem Zeitpunkt haben sich die Chipkarten unter anderem mit der enorm wachsenden Mobilfunkbranche stetig weiterentwickelt. Für diese Arbeit ist vor allem die Chipkartenvariante SIM wichtig. Neben den klassischen Mikroprozessorkarten für GSM⁹-Netze wurden im Jahr 1998 damit begonnen, JavaCards zu entwickeln. Diese verfügen über eine Java-Laufzeitumgebung.

2.1.2 Typen

Bei Chipkarten wird zwischen zwei verschiedenen Typen unterschieden: Speicher- und Prozessor-Karten [2]. Die Karten mit Mikroprozessoren sind mittlerweile stärker vertreten als die reinen Speicherkarten.

Anwendungsgebiete für diese verschiedene Varianten von Chipkarten sind:

- Telekommunikation - SIM-Karte
- Bankwesen - elektronische Geldbörse
- Gesundheitswesen - Krankenkassenkarte
- Sicherheitskritische Anwendungen - Identifizierung und Authentifizierung
- Service-Anwendungen - Pay-TV

Speicherkarten werden dort verwendet, wo kleine Datenmengen abrufbar sein müssen. Meist wird dies in Form von EEPROM¹⁰-Speicherbausteinen realisiert, die über ein serielles Protokoll beschrieben und gelesen werden können [1]. Manche von ihnen sind

⁹Global System for Mobile communications

¹⁰Electrically Erasable Programmable Read-Only Memory

dazu in der Lage über einfache Schaltungen simple Funktionen zu integrieren. Nach Ausgabe ist es jedoch nicht mehr möglich die Funktionalitäten zu ändern oder erweitern.

Prozessorkarten erweitern die Speicherkarten um das Vorhandensein eines Prozessors, der dazu in der Lage ist verschiedene (z.B. arithmetische) Operationen auszuführen. Diese sind für rechenintensive Implementierungen von Kryptoalgorithmen wie z.B. auch AES wichtig. Ebenfalls verfügt diese Variante von Chipkarten über ein spezifisches Betriebssystem mithilfe derer die verschiedenen Anwendungen ausgeführt werden können. Neuere Architekturen erlauben es nach der Ausgabe der Karten Codestrecken zu ändern oder erweitern.

2.1.3 SIM-Karten

Standardisierung *SIM-Karten* folgen dem Standard der Chipkartentechnologie und sind in ISO/IEC 786 definiert. Dort werden mechanische, physikalische und elektrische Eigenschaften festgelegt. So werden z.B. verschiedene Formfaktoren eingeführt, die Kompatibilität auf mechanischer Ebene sicherstellt. Mobilfunkkarten müssen dem ID-000-Format entsprechen, wohingegen Kreditkarten dem ID-1-Format entsprechen. Die physikalischen und elektrischen Definitionen behandeln vor allem die Konformität der Kontaktflächen, damit der Informationsaustausch zwischen Chipkarte und Lesegerät gesichert ist. Eine Illustration befindet sich unter Anhang 10: Pinbelegung einer Chipkarte auf S. 81.

Neben den bereits genannten Aspekten wird auch das logische Verhalten des Betriebssystems genau definiert. Interaktionen von Anwendungen mit dem Betriebssystem auf der Chipkarte werden auf diesem Weg festgelegt. Desweiteren wird auch die Struktur des Dateisystems genau beschrieben. Hier wird hohen Wert darauf gelegt auch auf kleine Datenmengen granulare Rechtevergabe zu gewährleisten. Eine Illustration befindet sich unter Anhang 11: Filesystemarchitektur einer Chipkarte auf S. 81.


Benutzte Typen sind MF¹¹ für das Wurzelverzeichnis, DF¹² für Dateien von Anwendungen sowie EF¹³ für die eigentlichen Dateien. Für jeden genannten Typ existieren unterschiedliche Formate.

¹¹Master File

¹²Dedicated File


¹³Elementary File

Grundlegend besteht eine Datei immer aus einem Header und einem Body. Während der Body die Nutzdaten trägt, wird im Header der Dateityp und Zugriffsregeln angegeben. Die Dateitypen beschreiben die genaue Struktur der Daten (zyklisch, linear fest, linear flexibel,...).

Informationen in Dateien auf SIM-Karten können entweder read-only- oder read-write- Berechtigungen aufweisen. Je nach Art der Information. Sicherheits- und identitätsbezogene Dateien sind in der Regel read-only abgelegt,  bei SMS-Nachrichten sowie Kontaktdaten read-write-fähig sein müssen. Unter anderem werden folgende Informationen auf der SIM-Karte gespeichert:

Name	Beschreibung
PIN	Sicherheitscode zur Freischaltung der SIM-Karte
PUK ¹⁴	Sicherheitscode bei Wiederfreischaltung der SIM-Karte
K _i	Eindeutiger Authentifizierungsschlüssel. Teilnehmer und Provider verfügen über ihn.
IMSI ¹⁵	Identifiziert einen Netzteilnehmer weltweit anhand dieser Zeichenfolge
ICCID ¹⁶	Identifiziert eine SIM-Karte weltweit anhand dieser Zahlenfolge (19-20 Stellen)
LAI ¹⁷	nimmt den aktuellen Locationcode beim eingewählten Punkt auf
SMS	Vom Teilnehmer empfangene oder versandte Nachrichten
Kontakte	Vom Teilnehmer verwaltete Kontakte

Tabelle 1: Auf SIM-Karten gespeicherte Informationen

IMSI Wie bereits aus der Tabelle hervorgeht, identifiziert die IMSI weltweit einen Netzteilnehmer. Sie wird von Netzbetreibern ausgelesen und besteht aus drei Abschnitten: MCC¹⁸ (3 Stellen), MNC¹⁹ (2-3 Stellen) und MSIN²⁰  10 Stellen). Üblicherweise wird die IMSI als 15-stellige Zahl angegeben, sofern das Land keine andere Vorgabe macht. Nach dem MCC werden in Europa beispielsweise 2-stellige MNCs verwendet. Nordamerika hingegen verwendet 3 Stellen.

In Deutschland wird der Code für z.B. die Telekom oder Vodafone wie folgt zusammengesetzt:

¹⁸Mobile Country Code

¹⁹Mobile Network Code

²⁰Mobile Subscriber Identification Number

Betreiber	MCC	MNC	MSIN
Telekom	262	01	...
Vodafone	262	02	...

Tabelle 2: Ländercodes und Mobilfunkbetreiber

Nach dem MCC und MNC identifiziert die MSIN letztendlich den Teilnehmer im jeweiligen Providerteil des Landes.

Eine IMSI wird in jedem Mobilfunknetz benötigt: GSM, UMTS²¹ und LTE²². Sie ist auf der SIM-Karte gespeichert und wird beim initiieren des Authentifizierungsvorgangs (mit weiteren Werten) an den Provider geschickt. Da diese Nummer benötigt wird, um einen Netzteilnehmer zu identifizieren, besteht hier Angriffsfläche für einen Man-In-The-Middle-Angriff. Diese finden in Form von *IMSI-Catchern* statt. Aus diesem Grund wird gemieden, die IMSI öfter als tatsächlich notwendig zu versenden. Damit diese nicht immer benötigt wird, generiert das VLR²³ immer eine vorübergehend gültige TMSI²⁴ [3].

TMSI Diese Zahl wird generiert, damit Teilnehmer nicht durch die Übermittlung ihrer IMSI identifiziert und abgehört werden. Ist ein mobiles Endgerät im Mobilfunknetz angemeldet, bekommt es vom VLR, bei dem es örtlich angemeldet ist, eine TMSI zugewiesen. Diese ist in einem begrenzten Zeitraum gültig und kann statt der tatsächlichen IMSI übermittelt werden, bis diese erneuert wird. Die Erneuerung kann von beiden Seiten initiiert werden [3].

IMSI-Catcher Machen sich zu Nutze, dass der Authentifizierungsvorgang nur den Mobilfunkteilnehmer, nicht aber den Provider, verifiziert. Dies wurde erst mit der Einführung von UMTS nachgerüstet.

Ein IMSI-Catcher gibt sich als VLR aus und triggert beim neu erkannten mobilen Endgerät das erneuern bzw. versenden der TMSI beziehungsweise IMSI sowie unter Umständen auch der IMEI²⁵. Der Catcher kann nun mit diesen Daten (als Man-In-The-Middle) Gespräche abhören. Er leitet das Gespräch weiter, indem er die Übertragung der Daten vom Endgerät unverschlüsselt fordert. In Richtung des realen VLR muss das

²¹Universal Mobile Telecommunications System

²²Long Term Evolution

²³Visitor Location Register

²⁴Temporary Mobile Subscriber Identity


²⁵International Mobile Station Equipment Identity



Gespräch wieder verschlüsselt sein. Es muss also eine SIM-Karte verwendet werden. Da diese eine vom Originalabsender abweichende Telefonnummer aufweist, wird diese meist beim Senden unterdrückt.

IMEI Sie wird dazu verwendet ein mobiles Endgerät als valide Hardwarekomponente zu identifizieren. Sie folgt ebenfalls den *3GPP*-Standards und ist sowohl physikalisch (als Aufschrift oder Gravur etc.) als auch mittels Software auszulesen. Sie wird nicht auf der SIM-Karte gespeichert. Gestohlene Geräte können auf diesem Weg von der Nutzung in Mobilfunknetzen ausgeschlossen werden.

Befehlsklassen werden nach Dateiverwaltung, Authentifizierung, Kryptographie und Zählvariablen unterschieden[1]. Einzelne Kommandos, die in diese Klassen eingeteilt sind, werden immer als APDU²⁶ versandt sowie empfangen.

Unter Anhang 12: Befehlsklassen einer Chipkarte auf S. 82 befindet sich eine genauere Auflistung der verschiedenen Befehle mit zugehörigen APDU-Beispielen.

Sicherheit Über Chipkarten wird in GSM- und UMTS-Netzen Sicherheit abgebildet. Sie  Kryptoalgorithmen auf Prozessoren sowie Kryptoschlüssel im Filesystem. So werden beispielsweise die Kryptoalgorithmen A3, A5 und A8 auf SIM-Karten ausgeliefert, die zum Nachweis der Authentizität des Benutzers dienen. Diese werden in Abschnitt 2.3: Authentifizierungsvorgang auf S. 10 näher erläutert.

Angriffe auf die Architektur waren vor allem vor den aktuell durch UMTS integrierten Funktionen möglich. Hierzu  die Implementierung des *COMP128*-Algorithmus. Er ist dazu entwickelt,  A3- und A8-Algorithmus bei der GSM-Architektur zu realisieren. Bei der Authentifikation des UC beim AuC erweist sich das eingesetzte Hashverfahren als problematisch, da kleine Veränderungen in der Eingabe nicht ausreichend streuen. Somit ist es möglich, dass auf der SIM-Karte gespeicherte geheime Schlüssel durch Kollisionsangriffe ausgelesen werden können.


Ebenso ist es bedingt möglich, SIM-Karten zu klonen. Mit zunehmender Anzahl von implementierten Sicherheitsmechanismen steigt die Herausforderung an den Klonprozess. So reicht es mittlerweile nicht mehr aus, nur Informationen auszulesen und zu klonen.

²⁶~~Application Protocol Data Unit~~

~~Aufgrund der Tatsache, dass~~ SIM-Karten mittlerweile nicht nur gespeicherte Informationen ausliefern, sondern auch eigene Funktionen integrieren. Dazu gehört beispielsweise das ableiten des Sitzungsschlüssels mittels abgelegter und mit dem AuC ausgetauschter Informationen. Ebenso wird das **a**uslesen von Informationen durch das bereits erwähnte Berechtigungssystem erschwert.


Bei der Nutzung älterer Architekturen ist ein Man-In-The-Middle-Angriff via IMSI-Catcher **weiter** zu beachten. Dieser wurde unter 2.1.3: TMSI auf S. 6 bereits genauer beschrieben.

USIM²⁷ ist die Weiterentwicklung der vorhergehenden normalen SIM-Karte. Sie wird im 3G- beziehungsweise UMTS-Netz verwendet und erweitert das ursprüngliche Modell um einige Funktionen. Diese Funktionen werden verwendet, sofern das Gerät, in die sie eingesetzt wird, den Standard von USIM-Karten unterstützt. Manche Ausprägungen dieser Kartenvariante erlauben auch das Ablegen von Mailadressen zu Kontakten. Der Hardwarekern dieser Architektur ist die UICC²⁸.

Formfaktoren der SIM-Karte können *Full*, *Mini*, *Micro*, *Nano* und *Embedded* sein. Sie reichen von den Abmessungen einer Kreditkarte über nur wenige Millimeter bis hin zur  letten Integration in ein Gerät. Sie werden von unterschiedlichen Standards der ISO/IEC, ETSI und JEDEC definiert.

Die meisten größeren SIM-Karten werden gezielt größer gefertigt. Bei *Fullsize*-SIM-Karten wird das eigentliche SIM-Modul in eine Kunststoffkarte der Größe einer Kreditkarte eingefasst. Die Funktion entspricht trotzdem der einer gewöhnlichen *Mini*- oder *Micro*-SIM. Zur Verwendung der unterschiedlichen Formfaktoren muss lediglich der eingesetzte Kartenleser geeignet gewählt werden.

2.2 Mobilfunkstandards

 Technologien für den Mobilfunk basieren unter anderem auf den bereits beschriebenen Chipkartenstandards. Weit verbreitete Standards für die Kommunikation sind *UMTS* und *GSM*. Sie schließen neben den Chipkarten einen weiteren Kontext mit ein. So

²⁷Universal Subscriber Identity Module

²⁸Universal Integrated Circuit Card

gehören zu diesen Standards auch die Rollen des Mobiltelefons (UE), Authentifizierungsstellen (AuC) sowie weitere Teilnehmern.

Die GSM- und UMTS-Kryptoalgorithmen basieren auf symmetrischen Verfahren, die nicht vollständig veröffentlicht wurden. Kryptoschlüssel müssen deshalb auf Seiten der SIM-Karte und Authentifizierungsstelle parallel aufbewahrt werden. Durch die Stelle der HLR²⁹ wird den AuC die Verwaltung der Schlüssel erleichtert. Der Anwender selbst meldet sich im BSS³⁰ bei GSM oder dem RNC³¹ bei UMTS an [1], die wiederum über das MSC³² verbunden sind. Eine Illustration dieser Hierarchie befindet sich unter Anhang 13: Teilnehmer in Mobilfunknetzen auf S. 82

GSM Möchte sich ein Teilnehmer im GSM-Mobilfunknetz anmelden, wird zuerst dessen Identität gesichert. Form der TMSI ausgehend von der SIM-Karte über das VLR an das HLR übertragen. Aus der TMSI und dem Standort, der sich aus dem VLR ergibt kann dann die IMSI ermittelt werden [1]. In der nachfolgenden Abbildung wird der Zusammenhang dargestellt.

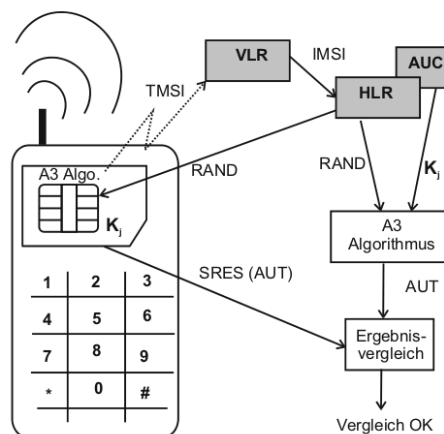


Abbildung 1: Ablauf der GSM-Authentifizierung [1]

Das HLR überprüft dann mittels IMSI die Benutzerrechte des Teilnehmers und stößt den Authentifizierungsvorgang an. Selbiger wird dann auf Seite des Mobilfunkteilnehmers sowie Providers ausgeführt und das Ergebnis verglichen. Stimmen beide Ergebnisse

²⁹Home Location Register

³⁰Base Station System

³¹Radio Network Controller

³²Mobile Switching Center

überein werden weiterführend Sitzungsschlüssel erzeugt, mit welchen die Nutzdaten (z.B. Sprache) sicher übertragen werden können.


UMTS weist einige Unterschiede zum GSM-Mobilfunknetz auf, wobei die vorhergehende Architektur aus Gründen der Abwärtskompatibilität weiterhin bestehen bleibt (VLR,HLR,AuC). Es wird die Variante der USIM als zentrale Chipkarte eingesetzt. Der Ablauf der Authentifizierung hingegen weicht vom Vorgänger etwas ab. Dieser wird dadurch erweitert, dass das AuC sich nun auch gegenüber der USIM authentisiert. Auf diesem wird die Sicherheit verbessert, da in beide Richtungen geprüft wird, ob eine valide Gegenstelle vorliegt. IMSI-Catchern wird es damit schwerer gemacht eine Form des *Man-In-The-Middle*-Angriffs zu ermöglichen. Dementsprechend wird auch das Mitschneiden von Gesprächen erschwert. Weiterführend wird im Gegensatz zu GSM nun auch der Verkehr zwischen HLR und VLR verschlüsselt realisiert [1]. Die Authentifizierung selbst über den Milenage-Algorithmus ist ebenfalls ausgebaut und verfügt nun über weitere Vektoren, die im nachfolgenden Kapitel näher erläutert werden.

2.3 Authentifizierungsvorgang

Eine Darstellung des Authentifizierungsvorgangs ist in Abbildung 2 zu sehen. Diese beschreibt nur die Authentifizierung einer USIM-Karte, aber nicht einer GSM SIM-Karte.

Die Authentifizierung ist recht umfangreich und soll hier anhand des Diagramms näher erläutert werden. Anschließend wird auf die Stärken des Authentifizierungsvorgangs eingegangen unter anderem in Bezug auf die Vorgehensweise in GSM.

2.3.1 Ablauf

Die Authentifizierung mit USIM-Karten geschieht in zwei Richtungen. Es muss nicht nur die USIM verifizieren können, dass sie eine valide Karte ist, sondern der Netzprovider muss sich auch gegenüber der USIM verifizieren. Möchte eine USIM nun Zugang zum Netz erhalten, schickt sie Registrierungsanfrage an den nächsten MSC. Diese fragt daraufhin beim AuC des Netzproviders nach einem AV³³. Dieser wird berechnet und nutzt dabei den in Kapitel 2.4 auf S. 14 angesprochenen Milenage-Algorithmus. Der AV ist ein Quintett und besteht aus 

³³Authentifizierungsvektor

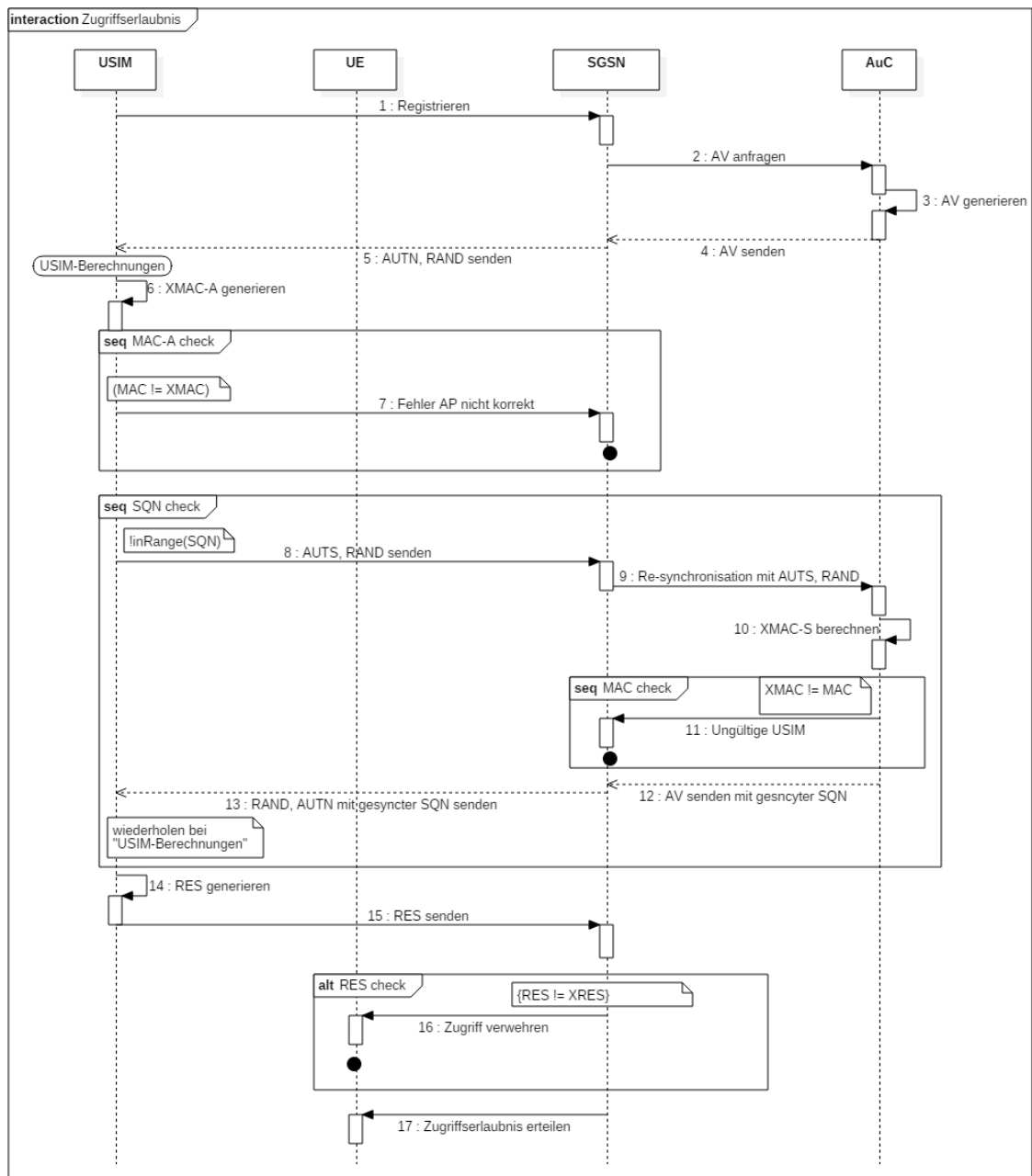


Abbildung 2: Sequenzdiagramm über die Kommunikation

RAND Ein Zufallswert, der vom AuC jedes Mal neu generiert wird

XRES Der Wert mit dem sich die USIM verifiziert

CK, IK Für Verschlüsselungs- und Integritätsschutz bei der späteren Kommunikation

AUTN Authentifizierungstoken, den die USIM benötigt. Ist selber ein Triplet aus:

SQLN \oplus **AK** Eine Sequenznummer verschleiert durch einen Anonymitätsschlüssel

AMF Providerspezifisches Managementfeld

MAC-A Token zur Verifizierung des Netzwerks

Die Werte XRES, CK und IK speichert der MSC zwischen und RAND, sowie AUTN werden an die USIM weitergegeben als Antwort auf die Registrierungsanfrage.

Mit Hilfe des RAND kann die USIM nun den XMAC-A berechnen, also den erwarteten Netzwerktoken, um zu überprüfen, dass die USIM sich bei dem richtigen Netzprovider registriert. Wenn XMAC-A und MAC-A nicht übereinstimmen, wird dem MSC mitgeteilt, dass die Authentifizierung nicht erfolgreich war und die Kommunikation beendet.

Sollten die Werte jedoch wie erwartet übereinstimmen, überprüft die USIM, ob die SQLN im erlaubten Bereich ist. Wenn dem nicht so ist, bedeutet das nicht automatisch, dass die Kommunikation wieder beendet wird. Die USIM speichert die gültige SQLN, was in Kapitel 2.4.2: Funktionsweise auf S. 15 nochmal näher erläutert wird. Diese wird ~~aktualisiert~~ bei jeder Authentifizierungsanfrage und so kann es passieren, dass das AuC keine aktuelle SQLN generiert hat und eine Resynchronisation vorgenommen werden muss, welche am Ende dieses Kapitels erklärt wird.

Wenn die SQLN im erlaubten Bereich liegt, generiert die USIM ihrerseits RES und schickt diesen an den MSC. Der MSC überprüft nun, ob XRES des Authentification Centers mit dem RES der USIM übereinstimmt. Je nach Ergebnis erteilt der MSC dem UE den Zutritt zum Mobilfunknetz oder nicht.

Resynchronisation Wie vorher in diesem Kapitel erwähnt, kann es passieren, dass die SQLN zwischen USIM und AuC nicht mehr synchron sind und deswegen resynchronisiert werden müssen. Dafür generiert die USIM ebenfalls einen RAND-Wert und einen AUTS-Token, welcher dem AUTN sehr ähnlich ist. Der AUTS hat im Gegensatz zum AUTN aber keinen AMF und statt dem MAC-A, wird ein MAC-S verschickt. Die SQLN im AUTS, die geschickt wird, ist die letzte gültige SQLN, die die USIM erhalten hat. Aus dieser SQLN kann das AuC nun eine gültige SQLN berechnen. Wichtig ist an dieser Stelle

zu erwähnen, dass SQN auch bei der Resynchronisation mit einem AK verknüpft wird, aber dieser AK wird etwas anders berechnet, als der AK der für die erste SQN seitens des Netzproviders verwendet wurde. Genauer wird dies in Kapitel 2.4.2: Funktionsweise auf S. 15 erklärt.

Ähnlich wie bei der Authentifizierung berechnet diesmal das AuC eine XMAC-S und vergleicht diese mit dem MAC-S aus dem AUTS. Sind diese nicht gleich, geht wieder eine Meldung an den MSC, ~~die~~ einen Fehler meldet und den Authentifizierungsvorgang beendet. Wenn diese Werte jedoch gleich sind, wird mit Hilfe der neuen SQN nochmal der AV berechnet und an den MSC geschickt. Dieser gibt wieder AUTN und RAND an die USIM weiter, welche die selben Schritte wie zuvor ausführt und diesmal zu dem Ergebnis kommen sollte, dass SQN im erlaubten Bereich liegt.

Als Quelle für dieses Kapitel und für weitere Informationen dient [4].

2.3.2 Stärken der UMTS Authentifizierung

Der Aufbau der UMTS-Struktur beruht auf der selben Struktur, wie bereits GSM. Denn auch wenn GSM einige Schwächen aufweist, ~~zeigen~~ die zehnjährige Existenz von GSM, dass die Sicherheitsarchitektur gut ist [5]. Deshalb hat sich die 3GPP³⁴ auch dazu entschieden, dass die Basissicherheitsfeatures erhalten bleiben in UMTS. Konkret nennen Pütz, Schmitz und Martin (s. [5]) unter anderem die folgenden Punkte:

- Vertraulichkeit der Teilnehmeridentität
- Teilnehmerauthentifizierung gegenüber dem Netzwerk
- Authentifizierung des Teilnehmers gegenüber der SIM
- Sicherheitsfunktionen ohne Aktion des Nutzers
- Eine Authentifizierungsmethode, die vom SN³⁵ durchgeführt wird, aber der gleichzeitig nur minimal vertraut werden muss
- Die Möglichkeit für jeden Provider eigene Authentifizierungsalgorithmen zu nutzen

Zusätzlich zu den Sicherheitsfeatures, die aus GSM übernommen wurden, hat UMTS einige Schwächen aus GSM beseitigt oder zumindest das Ausnutzen eben jener erschwert. So können abgefangene AVs von Angreifern nicht wiederverwendet werden, da die SQN eingeführt wurde. [5]

³⁴3rd Generation Partnership Project


³⁵Serving Network

Außerdem muss sich nun auch der Netzprovider der USIM gegenüber verifizieren, was Man-in-the-Middle Angriffe erschwert. [5]

Als letzter Punkt ist zu erwähnen, dass Daten noch länger verschlüsselt bleiben, als noch bei GSM, was die Angriffspunkte an denen man an die unverschlüsselten Daten direkt kommt weiter minimiert. [1]

2.4 Milenage Algorithmus

Zwischen SIM-Karte und Netzprovider muss eine sichere Authentifizierung und Kommunikation gewährleistet werden können. Dies für den USIM-Vorgänger entwickelten Algorithmus des 3GPP nicht mehr gewährleistet. Mit der Entwicklung des neuen Netzstandards wurde deshalb entschieden auch einen neuen Algorithmus zu entwickeln, nämlich der Milenage Algorithmus.

Dieser verfügt über die sieben Funktionen $f1, f1^*, f2, f3, f4, f5, f5^*$ mit Hilfe derer eine sichere Authentifizierung und Schlüsselgenerierung ermöglicht wird. Die Funktionen mit * sind dabei lediglich für die Re-synchronisation nötig. 

3GPP hat, wie auch beim Vorgänger, diese Funktionen nicht näher spezifiziert und ermöglicht den Netz Providern eigene Lösungen zu implementieren. Deshalb wird nur beschrieben in welchem Kontext diese Funktionen Anwendung finden und generelle Anforderungen an diese Algorithmen definiert [6].

Der Milenage Algorithmus hat wie erwähnt zwei Hauptaufgaben, nämlich einerseits die Authentifizierung und andererseits die Generierung eines Schlüssels, um die versendeten Nachrichten zu ver- und entschlüsseln.

In den nachfolgenden Unterkapiteln werden die Vorteile und die Funktionsweise des Algorithmus, erläutert.

2.4.1 Warum Milenage sicher ist

In Kapitel 2.3: Authentifizierungsvorgang auf S. 10 wurde schon erklärt, dass die Mechanismen zur Authentifizierung und zum Schlüsselaustausch angepasst wurden und durch einige Änderungen Vorteile mit sich bringen, aber auch der weiterentwickelte Algorithmus zur Berechnung der Daten weist einige Vorteile auf.

Ein wichtiger Punkt dabei ist, dass der Vorgänger geheim war, aber Milenage wurde

öffentlich diskutiert. Der Vorteil der Offenlegung ist, dass auf diesem Weg Schwachstellen bemerkt werden bevor der Algorithmus implementiert ist.

Stephan Spitz zeigt in seinem Buch “Kryptographie und IT-Sicherheit” [1] drei wesentliche Gründe auf, die den Algorithmus sicher machen:

Ergebnisse mit hoher Entropie Wenn der Schlüssel K unbekannt ist und die Eingabeparameter SQN^{36} , $RAND$ und AMF^{37} variieren, dann werden “sehr gute Pseudo-Zufalls-Ergebnisse mit einer hohen Entropie” [1] erreicht. Dafür muss aber auch eine gute Blockchiffre wie AES eingesetzt werden.

Keine Rückschlüsse auf K möglich Wenn die Eingabe- und Ausgabeparameter der einzelnen Funktionen $f1$ bis $f5$ analysiert werden, lassen sich keine Rückschlüsse auf den Schlüssel K oder das OP^{38} ziehen, auch nicht auf Teile dessen. Dies hängt unter anderem damit zusammen, dass K nicht direkt in die Funktionen eingeht.

Schlüssellänge Brut-Force-Angriffe, also stumpfes ausprobieren der Schlüssel, dauert aufgrund der Schlüssellänge von 128 Bits selbst mit aktuellen Computern noch zu lange.

2.4.2 Funktionsweise

In Kapitel 2.3: Authentifizierungsvorgang auf S. 10 wurde beschrieben, welche Daten zwischen AuC und UE verschickt werden, jedoch nicht, wie diese Daten generiert werden. Es gibt einige Werte, die auf der USIM und der Datenbank des AuC fest eingespeichert sind. Diese sind der OP und K , sowie jeweils fünf Rotations- und XOR-Konstanten ($r1, \dots, r5$ und $c1, \dots, c5$). Welche Funktion welche Werte benötigt und generiert zeigt dabei Abbildung 3.

In Abbildung 3 ist zu sehen, dass zu Beginn die SQN generiert wird. Diese ist insgesamt 48 Bits lang und besteht aus den beiden Teilen SEQ und IND , mit SEQ als die eigentliche Sequenznummer und IND als Arrayindex. Dieser Index wird benötigt, da auf der SIM-Karte die letzten $SQNs$ in einem Array gespeichert sind. Die empfohlene Arraygröße ist 32, was für IND eine Länge von fünf Bits bedeutet. Mit diesem Index kann nachher die

³⁶Sequence Number

³⁷Authentication Management Field

³⁸Operator Configuration Field

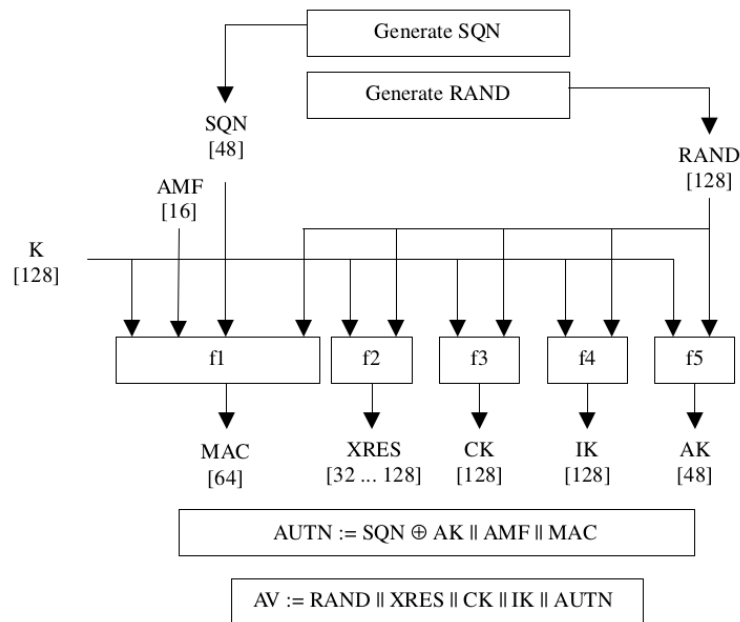


Abbildung 3: Übersicht über die Generierung der Authentifizierungsvektoren [4]

Aktualität der SEQ überprüft werden.[4] 

Für die Bildung der SEQ selbst gibt es drei verschiedene Möglichkeiten:

- teilweise zeitbasiert
- nicht zeitbasiert
- komplett zeitbasiert

Die einfachste Variante ist die nicht zeitbasierte Lösung, bei der lediglich ein Zähler hochgezählt wird mit jeder Authentifizierungsanfrage. Die SEQ ist initial also 0 und wird hochgezählt. Das AuC speichert in einer Datenbank [4] zu jeder USIM die aktuelle SQN. Auf die anderen Möglichkeiten wird hier nicht näher eingegangen, da sie in dieser Arbeit keine Anwendung finden und deutlich komplexer sind.

Als nächstes wird die RAND gebildet. Das Verfahren, ~~wie~~ der Netzprovider diese RAND generiert, darf nicht offen gelegt werden, da dies die Sicherheit stark beeinflussen würde. Die Spezifikation gibt deshalb für die Generierung der RAND keine Empfehlung, wie für die anderen Werte. Generell handelt es sich bei der RAND um eine 128-bit lange Zufallszahl, die für alle anderen Funktionen benötigt wird.

Abbildung 3 zeigt zwar, welche Variablen in die Funktionen einfließen und welche Werte sie zurückgeben, aber sie zeigt nicht näher, wie diese Werte nun in den einzelnen Funk-

tionen angewendet werden. Dies zeigt Abbildung 4 besser. Dort ist zu erkennen, dass $f2$ bis $f5^*$ nach dem selben Schema berechnet werden können und $f1$, sowie $f1^*$ noch einige zusätzliche Parameter benötigen.

Zunächst die Erklärung der Symbole sowie einiger weiterer Abkürzungen. OP_C wird durch folgende Formel generiert:

$$OP_C = OP \oplus E(OP)_K$$

$E()$ ist die Blockchiffre. In diesem Falle wird also OP mit dem Schlüssel K verschlüsselt. Welche Verschlüsselung gewählt wird, ~~wird von 3GPP nicht vorgegeben~~. In dieser Arbeit wurde AES³⁹ verwendet, welche im Kapitel 2.5: AES / Rijndael auf S. 19 näher beschrieben wird.

Der verschlüsselte OP wird dann im zweiten Schritt über XOR (\oplus) mit dem ursprünglichen OP verknüpft.

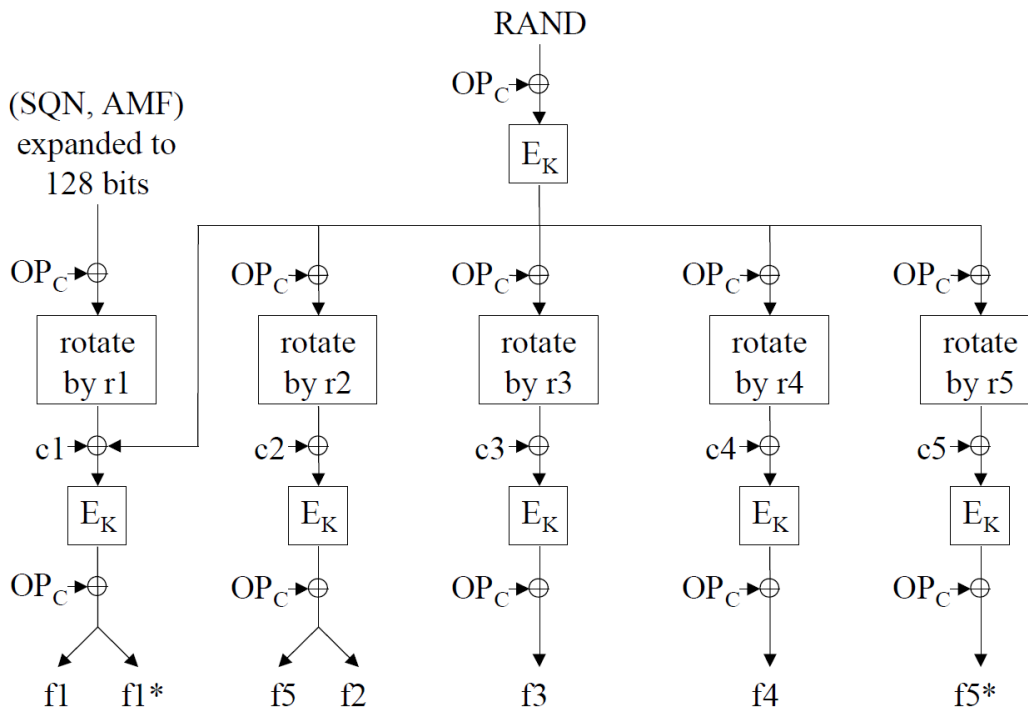


Abbildung 4: Schematische Darstellung der Berechnung [4]

In Abbildung 4 ist weiterhin der Funktionsblock “rotate by r” zu lesen. Beim rotieren wird der Eingabewert um die Anzahl an Bits des Wertes von r rechts rotiert und die

³⁹Advanced Encryption Standard

Bits die herausfallen links wieder eingefügt. Beispielsweise wird aus 110101 bei einem Rotationswert r von 2: 011101.

Mit den genannten Infos ist verständlich, dass die Funktionen f_2 bis f_5^* ~~ausgehen von~~ einem verschlüsselten $RAND \oplus OP_C$, welcher in der Dokumentation auch als TEMP bezeichnet wird. Dieser Wert wird wieder über XOR-mit OP_C verknüpft und um die entsprechende Rotationskonstante r rotiert. Im Anschluss wieder eine XOR Verknüpfung mit der speziellen XOR-Konstante c und die Chiffre des Ausgabewertes. Im letzten Schritt wird dieser dann nochmal über XOR mit OP_C verknüpft.

Wie die Abbildung 4 zeigt funktionieren f_1 und f_1^* sehr ähnlich. Bevor jedoch TEMP in die Berechnung einfließt wird SQN und AMF auf 128 Bits erweitert und bekommt in der Dokumentation die Bezeichnung IN1. IN1 besteht aus SQN und AMF abwechselnd konkateniert, also $SQN \parallel AMF \parallel SQN \parallel AMF$. [4]

Wie im Kapitel 2.5 erklärt wird, sind die Blöcke, die verschlüsselt werden immer 128 Bits lang. ~~Deshalb sind auch RAND, TEMP oder IN1 128 Bits lang.~~ Ein MAC-A oder RES jedoch sind kürzer, weshalb diese nur aus Teilen eines Blocks bestehen. Deswegen stehen an manchen Enden des Diagramms in Abbildung 4 auch mehr als ein Funktionsname. Es gibt fünf Blöcke aus denen sich die sieben Funktionswerte bilden. Sie werden in der Spezifikation von Milenage mit OUT_n bezeichnet.

OUT1 Beinhaltet die beiden jeweils 64 Bits langen Werte MAC-A und MAC-S

OUT2 Die ersten 64 Bits sind AK und die hinteren 64 Bits RES

OUT3 Da CK 128 Bits lang ist, ist OUT3 komplett CK

OUT4 Ähnlich CK ist auch IK 128 Bits lang

OUT5 Die ersten 64 Bits sind AK und der Rest wird nicht benötigt

Es ist kein Fehler, dass AK sowohl bei OUT2 als auch bei OUT5 erwähnt wird, sondern wie schon in Kapitel 2.3: Authentifizierungsvorgang auf S. 10 ~~geschrieben~~, wird der Anonymitätsschlüssel AK der mit SQN verknüpft wird im Falle der Resynchronisation anders berechnet als bei der Generierung der Authentifizierungsvektoren seitens des Netzproviders.

2.5 AES / Rijndael

Rijndael oder AES sind beide Blockverschlüsselungen. Das bedeutet, dass der zu verschlüsselnde Text in gleich große Blöcke aufgeteilt wird und jeder Block mit einem Schlüssel chiffriert wird. Die Blocklänge des verschlüsselten Text bleibt dabei gleich.

Der Unterschied zwischen Rijndael und AES besteht lediglich in der Länge der Text- sowie Schlüsselblöcke und ist sonst identisch. Für Rijndael gilt, dass Block- und Schlüssellänge unabhängig von einander auf Vielfache von 32 Bits definiert werden können. Minimal müssen beide aber 128 Bits lang sein und maximal 256 Bits [7].

AES auf der anderen Seite hat die Blocklänge auf 128 Bits festgelegt und die Schlüssellänge darf nur 128, 192 oder 256 Bits betragen [8]. Welche Schlüssellänge genutzt wird, kann die unterschiedlichen Bezeichnung AES-128, AES-192 und AES-256 deutlich gemacht werden.

Im folgenden soll kurz die Geschichte von AES beziehungsweise Rijndael erläutert werden und wie der Verschlüsselungsalgorithmus der beiden funktioniert. Nicht angesprochen wird der Entschlüsselungsalgorithmus, da der für diese Arbeit keine Relevanz hat.

2.5.1 Geschichte

Die erste Blockverschlüsselung, die vom NIST⁴⁰ 1977 offiziell übernommen wurde, war DES⁴¹ und basierte auf einer überarbeiteten Version der Blockverschlüsselung Lucifer. Bei DES betrug die Blockgröße nur 64 Bits, was lange Zeit ausreichend war, aber mit Fortschreiten der Rechenleistungen nicht mehr lange genügen würde gegen Brute-Force Angriffen (weitere Argumente finden sich in [9]). Es gab deswegen die Empfehlung den dreifach DES zu nutzen, aber der war nochmal langsamer, als der einfache DES, weshalb das NIST eine neue Blockverschlüsselung suchte über eine Ausschreibung (1997).

An die neue Verschlüsselung wurden folgende Bedingungen gestellt:

- Blockverschlüsselung mit einer Blocklänge von 128 Bits
- Die drei Schlüssellängen 128, 192 und 256 Bits müssen unterstützt werden
- Sicherheitslevel ähnlich anderer Algorithmen
- Effizient in Soft- und Hardware

⁴⁰National Institute of Standards and Technology

⁴¹Data Encryption Standard

Die Suche nach dem neuen Algorithmus fand öffentlich statt, sodass sich jeder auf der ganzen Welt beteiligen konnte und im August 1999 fünf Finalisten gekürt wurden. Dabei wurden öffentlich die Vor- und Nachteile der einzelnen Algorithmen diskutiert und Ende 2000 ~~hat~~ NIST mit Rijndael den Gewinner bekannt ~~gegeben~~. Dieser wurde von den beiden Belgiern Joan Daemen und Vincent Rijmen entwickelt.

An dieser Stelle sei auch ~~zu~~ ~~erwähnen~~, dass die NSA⁴² sehr viel ~~vertrauen~~ in AES hat. So ~~hat diese~~ die generelle Erlaubnis ~~erteilt~~ als *SECRET* eingestufte Dokumente mit AES zu verschlüsseln und selbst als *TOP SECRET* eingestufte Dokumente dürfen mit einer Schlüssellänge von 192 oder 256Bits verschlüsselt werden. [9]

2.5.2 Mitbewerber

Als die größten Konkurrenten von Rijndael werden “Twofish” und “Serpent” gesehen. Um zu verstehen, warum die Entscheidung am Ende trotzdem auf Rijndael fiel, sollen die beiden Konkurrenten an dieser Stelle untersucht werden. Beide verfolgen einen klassischen Ansatz.

Serpent Entwickelt wurde der Algorithmus von drei bekannten Kryptografen. Sie haben bei der Entwicklung ihres Algorithmus darauf geachtet, dass er noch in mindestens einem Jahrhundert sicher ist und deswegen auch auf Experimente verzichtet. Serpent ist wie AES eine Substitutions-Permutations-Chiffre. Sie ~~nutzen~~ hauptsächlich Substitutionsboxen und einige lineare Funktionen, welche in der Kombination seit den siebziger Jahren bekannt sind und deshalb als gut erforscht gelten. Außerdem nutzen sie 32, statt, wie in anderen Verfahren, die üblichen acht bis 16 Runden, um sicher zu gehen. Serpent hat nach aktuellem Stand einen sehr großen Sicherheitspuffer, da 32 Runden verwendet werden, aber diese hohe Anzahl ~~machen~~ ihn auch zu dem langsamsten ~~von den drei~~ Algorithmen. [10]

Twofish Der Nachfolger von Blowfish ist Twofish entwickelt unter anderem von Bruce Schneier ~~und ist~~ eine Feistel-Chiffre. In Twofish werden vier S-Boxen aus dem Schlüssel generiert, die jeweils zwei mal pro Runde eingesetzt werden. Insgesamt sieht Twofish 16 Runden vor. Wie auch bei Serpent sind abgesehen von den Substitutionsboxen alle Schritte linear. Es werden insgesamt 40 Subschlüssel je 32 Bits Länge benötigt und die

⁴²National Security Agency

S-Boxen müssen generiert werden.

Daraus folgt, dass Twofish mit seiner Geschwindigkeit zwischen Rijndael und Serpent liegt. Auch was die Innovationen im Design angeht, ist Twofish zwischen Serpent und Rijndael anzusehen. Bislang ist nur eine theoretische Schwachstelle bekannt, aber keine konkrete Methode, diese zu nutzen. [10]

2.5.3 Algebraische endliche Körper

Der Algorithmus ist verständlich und anwendbar, auch ohne das Wissen über die mathematischen Grundlagen dazu. Trotzdem beruht die Tatsache, dass der Algorithmus funktioniert, auf Mathematik und diese soll hier erläutert werden. Konkret liegt dem Algorithmus die algebraische Struktur des endlichen Körpers, auch Galois-Feld genannt, zu Grunde. Doch bevor der endliche Körper erklärt werden kann, muss als Basis die algebraische Gruppe definiert werden.

Eine Gruppe ist eine Menge von Elementen G und eine Operation \circ , die zwei Elemente aus G miteinander kombiniert. Es gelten dabei folgende fünf Eigenschaften für die Gruppe [9]:

1. Die Gruppenoperation \circ ist abgeschlossen: $\forall a, b \in G : a \circ b = c \in G$
2. Die Gruppenoperation ist assoziativ: $\forall a, b, c \in G : a \circ (b \circ c) = (a \circ b) \circ c$
3. Es gibt ein neutrales Element e : $\forall a \in G : a \circ e = e \circ a = a$
4. Für jedes Element a existiert ein inverses Element a^{-1} : $\forall a \in G : a \circ a^{-1} = a^{-1} \circ a = e$
5. Eine Gruppe ist außerdem kommutativ, wenn $\forall a, b \in G : a \circ b = b \circ a$

Als Beispiel sei die Menge \mathbb{Z} , mit der Operation $+$ und dem neutralen Element 0 genannt. Dies ist eine kommutative Gruppe mit dem inversen Element $-a$, denn $\forall a \in \mathbb{Z} : a + (-a) = 0$. Mit der Operation $*$ hingegen gibt es kein neutrales Element, welches eine Gruppe ermöglichen würde.

Anzumerken ist, dass die Menge nicht wie \mathbb{Z} unendlich sein muss, damit eine Gruppenbildung möglich ist. Ein Beispiel dafür wird ~~an jedoch erst~~ zu einem späteren Zeitpunkt geliefert.

Die Fortsetzung der Gruppe ist der Körper. Diese Struktur unterstützt die vier arithmetischen Basisoperationen Addition, Subtraktion, Multiplikation und Division. Die Definition für einen Körper K lautet [9]:

1. Alle Elemente aus K bilden eine additive Gruppe mit dem Gruppenoperator “+” und dem neutralen Element 0.
2. Alle Elemente aus K , außer 0, bilden eine multiplikative Gruppe mit dem Operator “ \times ” und dem neutralen Element 1.
3. Wenn beide Gruppenoperatoren gemischt werden, gilt das Distributivgesetz:
 $\forall a, b, c \in K : a \times (b + c) = (a \times b) + (a \times c)$

Ähnlich wie bei Rijndael spielen im Großteil der Kryptographie die endlichen Körper, oder Galois-Feld genannt, eine große Rolle. Die endliche Menge, die einen Körper zu einem endlichen Körper machen, ist zählbar und die Anzahl der Elemente wird “Ordnung” oder “Kardinalität” genannt. Ein Körper kann jedoch nur mit folgendem Theorem endlich sein [9]:

Ein Körper mit einer Ordnung m existiert nur, wenn m eine Primzahlpotenz ist: $m = p^n$, mit einer positiven ganzen Zahl n und einer Primzahl p . p heißt “Charakteristik” von K .

Daraus folgt, dass ein endlicher Körper mit der Kardinalität von $11 = 11^1$ oder $81 = 3^4$ existiert, aber nicht 12, da $12 = 2^2 * 3$. Die einfachsten Körper sind dabei wohl die mit einer Primzahlcharakteristik, also mit $n = 1$, sogenannte “Primkörper”.

Die Elemente des Galois-Felds $GF(p)$ werden mit ganzen Zahlen repräsentiert: $0, 1, \dots, p-1$. Die arithmetischen Rechenoperationen werden *modulo* p betrieben. Dadurch ist das additive Inverse eines Elements a gegeben mit $a + (-a) = 0$ modulo p und das multiplikative Inverse jedes Elements a außer 0 ist definiert als $a * a^{-1} = 1$. Nachfolgend ist ein kleines Beispiel mit $GF(5)$ zum besseren Verständnis:

Addition:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Additives Inverses:

$$\begin{aligned} -0 &= 0 \\ -1 &= 4 \\ -2 &= 3 \\ -3 &= 2 \\ -4 &= 1 \end{aligned}$$

Multiplikation:

\times	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Multiplikatives Inverses: 0^{-1} existiert nicht

$1^{-1} = 1$

$2^{-1} = 3$

$3^{-1} = 2$

$4^{-1} = 4$

Ein wichtiger Primkörper ist $GF(2)$, welcher der kleinste endliche Körper ist, der existiert und wichtig für AES ist. Die Addition kommt in diesem Körper der booleschen XOR-Operation gleich. Gleichzeitig produziert die Multiplikation das selbe Ergebnis wie die boolesche Und-Operation:

Addition:

+	0	1
0	0	1
1	1	0

Multiplikation:

\times	0	1
0	0	0
1	0	1

Als letzte Struktur sei die Körpererweiterung erwähnt, welche als $GF(2^m)$ geschrieben werden kann, mit $m > 1$. AES hat seinen Algorithmus auf der Körpererweiterung $GF(2^8)$ aufgebaut, da sich damit alle 256 Werte eines Bytes repräsentieren lassen. Da 256 offensichtlich keine Primzahl ist, können die Additionen und Multiplikationen nicht repräsentiert werden durch die Ganzzahlige Addition und Multiplikation modulo 2^8 . Deshalb wird zum einen eine andere Notation für die Elemente des Körpers benötigt und zum anderen neue Regeln für die Durchführung der arithmetischen Rechenoperationen. Nachfolgend soll aufgezeigt werden, dass die Elemente als "Polynome" dargestellt werden können und darauf aufbauend "polynome Arithmetik" durchgeführt werden kann [9]. Jedes Polynom $A \in GF(2^m)$ wird dargestellt als:

$$A(x) = a_{m-1}x^{m-1} + \dots + a_1x + a_0, \quad a_i \in GF(2)$$

Die polynome Arithmetik wird nun im folgenden Kapitel ~~erklärt~~ speziell für die Körpererweiterung $GF(2^8)$.

2.5.4 Polynome Arithmetik in $GF(2^8)$

Die Addition bei einem endlichen Körper wird erreicht durch die Addition der einzelnen Koeffizienten. Wie im vorherigen Kapitel gezeigt, bedeutet das, dass die jeweiligen Koeffizienten über eine XOR-Operation verrechnet werden. Die Addition ist wie folgt definiert [9]:

Sei $A(x), B(x) \in GF(2^8)$, dann wird die Summe dieser wie folgt berechnet:

$$C(x) = A(x) + B(x) = \sum_{i=0}^7 c_i x^i, \quad c_i \equiv a_i + b_i \pmod{2}$$

Nachfolgend ist ein kurzes Beispiel der Addition mit unterschiedlichen Notationen aufgeführt:

$$\begin{aligned} (x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) &= x^7 + x^6 + x^4 + x^2 && \text{(polynome Notation)} \\ 01010111_2 \oplus 10000011_2 &= 11010100_2 && \text{(binäre Notation)} \\ 57_{16} \oplus 83_{16} &= d4_{16} && \text{(hexadezimale Notation)} \end{aligned} \tag{1}$$

In der polynomen Repräsentation entspricht die Multiplikation in $GF(2^8)$ (dargestellt durch \cdot) der Multiplikation von Polynomen ~~module~~ eines “irreduziblen Polynoms” achten Grades. Dabei ist ein Polynom irreduzibel, wenn es nur durch 1 und sich selbst teilbar ist, ~~ähnlich~~ einer Primzahl. Das irreduzible Polynom des AES-Algorithmus ~~ist~~:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{2}$$

Ein Beispiel eines reduzierbaren Polynoms ist $x^8 + x^3 + x^2 + 1$, da

$$x^8 + x^3 + x^2 + 1 = (x^5 + x^2 + 1) \cdot (x^3 + 1)$$

Mit der Kenntnis des irreduziblen Polynoms, lässt sich auch die Multiplikation vollständig definieren:

Sei $A(x), B(x) \in GF(2^8)$, dann wird die Multiplikation von zwei Elementen $A(x), B(x)$ und irreduziblen Polynom $m(x)$ berechnet mit

$$C(x) = A(x) \cdot B(x) \bmod m(x)$$

So ist bei der Multiplikation statt Addition der beiden Polynome aus Gleichung 1 das Ergebnis $x^7 + x^6 + 1$.

Das Multiplizieren zweier Polynome auf dem einfachsten Wege ohne Tricks dauert jedoch ziemlich lange und im Anschluss muss noch durch $m(x)$ dividiert werden. Das führt zwar am Ende zum richtigen Ergebnis, aber ist wie gesagt sehr zeitaufwändig. In der Spezifikation des AES wird deshalb eine Methode angeboten, wie ebenfalls das richtige Ergebnis zustande kommt, aber wesentlich effizienter.

Die Funktion, die in der Spezifikation vorgeschlagen wird, heißt $xtime(A(x))$ und multipliziert $A(x)$ mit x und reduziert durch $m(x)$. Es gilt:

$$C(x) = xtime(A(x)) = A(x) * x = \sum_{i=0}^7 a_i x^{i+1}, \text{ für } A \in GF(2^8)$$

Um bei einer Größe von 8 Bits für $C(x)$ zu bleiben, wird vor der Schiebeoperation überprüft, ob a_7 0 oder 1 ist. Wenn das Bit 0 ist, wird nach anwenden der Funktion $C(x)$ kleiner $m(x)$ sein. In diesem Falle ist im Anschluss keine Modulo-Operation nötig. Diese wird nur nötig wenn a_7 1 ist. Dann muss $C(x)$ noch modulo dem hinteren Teil von $m(x)$ berechnet werden, nämlich $x^4 + x^3 + x + 1$. Es muss nur mit dem hinteren Teil des irreduziblen Polynom $m(x)$ reduziert werden, da x^8 nicht genutzt wird in $C(x)$. [8]

Weiter ist $C(x) = m(x) + r$, wobei $r = A(x) \bmod m(x)$. Daraus folgt:

$$C(x) = xtime(A(x)) \oplus \begin{cases} 0 & \text{wenn } a_7 = 0, \\ 00011011_2 & \text{wenn } a_7 = 1. \end{cases}$$

Die Funktion $xtime()$ kann außerdem rekursiv verwendet werden. Das bedeutet, dass

$$A(x) * x^3 = xtime(A(x) * x^2) = xtime(xtime(A(x) * x)) = xtime(xtime(xtime(A(x))))$$

Das Ergebnis der Berechnung zweier Werte $A(x), B(x)$ kann damit erzielt werden, in

den die einzelnen Zwischenergebnisse von $B(x)$ aufaddiert werden mit $A(x)$. Dazu ein letztes Beispiel:

$$\begin{aligned} A(x) &= x^6 + x^4 + x^2 + x^1 + 1, B(x) = x^4 + x^1 + 1 \\ A(x) \cdot B(x) &= x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1, \text{ weil} \\ A(x) * x^1 &= x^7 + x^5 + x^3 + x^2 + x^1 \\ A(x) * x^4 &= x^2 + x^1 + 1 \\ \Rightarrow (x^6 + x^4 + x^2 + x^1 + 1) \oplus (x^7 + x^5 + x^3 + x^2 + x^1) \oplus (x^2 + x^1 + 1) \\ &= x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 \end{aligned}$$

Damit ist die Berechnung der beiden Elemente deutlich effizienter und dient als gute Grundlage für die Implementierung der AES-Chiffre. Diese wird im folgenden Kapitel beschrieben.

2.5.5 Funktionsweise

Bei der Blockverschlüsselung gibt es fünf verschiedene Betriebsmodi, die in der ISO/IEC 10116:2006 [11] beschrieben sind. Diese Betriebsmodi beschreiben wie ein Text bestehend aus mehreren Blöcken verschlüsselt wird.

Electronic Codeblock (ECB) Jeder Textblock wird unabhängig vom vorherigen Block verschlüsselt. Daraus resultieren vor allem zwei Dinge. Zum einen bewirkt eine Vertauschung der Blöcke im chiffrierten Text zur gleichen Vertauschung im entschlüsselten Text und zum anderen beeinflusst ein Fehler bei der Entschlüsselung eines Blocks nicht die anderen Blöcke. Es wird deshalb davon abgeraten diesen Modus zu verwenden bei mehr als einem Block.

Cipher Block Chaining (CBC) Beim CBC werden die Schwächen von ECB verkleinert, in dem eine Verkettung der Blöcke vorgenommen wird. So gibt es beim ersten Block zusätzlich zum Schlüssel noch einen Initialisierungsvektor. Bei den folgenden Textblöcken wird der Schlüssel mit dem vorhergehenden verschlüsselten Textblock verknüpft. Dadurch können Blöcke nicht mehr vertauscht werden und wenn ein Fehler bei der Ent

schlüsselung in einem Block passiert ist zusätzlich zu diesem auch der nachfolgende Block nicht mehr zu entschlüsseln.

Neben den beiden genannten Modi existieren noch “Cipher Feedback”, “Output Feedback” und “Counter”. Auf diese soll hier aber nicht näher eingegangen werden, da diese Elemente der Stromchiffre nutzen. Ihre Erklärung wäre deshalb für diesen Rahmen zu umfangreich, da beim Mileage-Algorithmus nur Zeichenketten verschlüsselt werden, die nicht länger oder kürzer als ein Block sind. Das ist auch der Grund, warum bei Mileage bedenkenlos der ECB eingesetzt werden kann.

Beim AES-Algorithmus werden nicht nur Teile sondern immer alle 128 Bits bearbeitet, weshalb die Anzahl der Iterationen relativ gering ist und den Algorithmus schnell macht. Bei einer Schlüssellänge von 128 Bits werden 10 Runden benötigt, bei 192 Bits 12 und bei der Schlüssellänge von 256 Bits sind 14 Runden nötig. In jeder Runde werden vier Funktionen auf den Textblock angewendet, um den Textblock gut zu verschlüsseln. In welcher Reihenfolge diese angewendet werden, wird später in diesem Kapitel erläutert.

Damit der Textblock mit einem Schlüssel verschlüsselt werden kann, müssen beide in ein rechteckiges Array aus Bytes transformiert werden. Die Anzahl der Reihen ist immer vier, aber die Anzahl der Spalten variiert je nach Textlänge. Mit der Umwandlung des Textblocks in das Array, spricht man vom Zustand und der umgewandelte Schlüssel ist nun der Chiffrierschlüssel.

Der Zustand ist bei AES immer vier Spalten breit und der Chiffrierschlüssel vier, sechs oder acht Spalten breit. In beiden Fällen wird das Array zeilenweise gebildet. Das bedeutet, dass bei einer Blocklänge von 128 Bits in den ersten vier Arrayelementen des Zustands die ersten 32 Bits des Blocks stehen.


Nachfolgend werden nun die vier Funktionen beschrieben, die den Zustand verändern.

SubBytes Bei der SubBytes-Transformation handelt es sich um eine nicht-lineare Bytesubstitution, welche separat auf jedes Byte des Status angewendet wird. Die Substitutionstabelle, auch S-Box genannt, ist 16 x 16 Felder groß und ist damit eine bijektive Abbildung aller 256 möglichen Eingabemöglichkeiten. Wie die Tabelle gebildet wird, wird an dieser Stelle nicht näher beschrieben, aber im Anhang 17: Substitutionstabelle für AES-Verschlüsselung auf S. 85 befindet sich die Tabelle.

Mit dem Wert des Statuselements wird nun der Wert aus der Substitutionstabelle er-

mittelt. Die ersten vier Bits geben die Zeile vor und die letzten vier Bits die Spalte. So wird beispielsweise der Wert $8D_{16}$ mit $5D_{16}$ substituiert.

ShiftRows Bei dieser Funktion werden die Bytes in jeder Zeile des Zustands-Arrays zirkulär verschoben. Das bedeutet, dass die einzelnen Bytes vorne, also links, “rausgeschoben” werden und hinten/rechts, wieder “reingeschoben”. Dabei ist die Anzahl der Verschiebungen in jeder Zeile unterschiedlich. In Zeile eins wird gar nicht verschoben, in Zeile zwei um ein Byte, Zeile drei um zwei Bytes und in Zeile vier ~~das Array~~ um drei Bytes ~~verschoben~~. Für die Berechnung existiert keine mathematische Grundlage, sondern es geht einfach um die Konfusion des Arrays.

Als Beispiel, sei der String in der vierten Zeile $3B|2A|34|7$  würde dieser durch die Rotation anschließend so aussehen: $71|3B|2A|34$.

MixColumns In der MixColumns-Transformation wird nicht jedes Byte einzeln transformiert, sondern die gesamte Spalte. Die Berechnung ist hierbei etwas umfangreicher, als in den Transformationen zuvor. Nehmen wir eine Spalte j , dann wird folgende Berechnung durchgeführt:

$$\begin{aligned} b_{0,j} &= T_2(a_{0,j}) \oplus T_3(a_{1,j}) \oplus a_{2,j} \oplus a_{3,j} \\ b_{1,j} &= a_{0,j} \oplus T_2(a_{1,j}) \oplus T_3(a_{2,j}) \oplus a_{3,j} \\ b_{2,j} &= a_{0,j} \oplus a_{1,j} \oplus T_2(a_{2,j}) \oplus T_3(a_{3,j}) \\ b_{3,j} &= T_3(a_{0,j}) \oplus a_{1,j} \oplus a_{2,j} \oplus T_2(a_{3,j}) \end{aligned}$$

$T_2(a)$ ist nun definiert als:

$$T_2(a) = \begin{cases} 2 * a & \text{wenn } a < 128, \\ (2 * a) \oplus 283 & \text{wenn } a \geq 128. \end{cases}$$

Die Definition von $T_3(a)$ ist lediglich $T_2(a) \oplus a$.

Durch diese Definition gilt, dass wenn $a = 143$, dann

$$\begin{aligned} T_2(143) &= 5 \\ T_3(143) &= T_2(143) \oplus 143 = 138 \end{aligned}$$

Mathematisch betrachtet beruht diese Funktion auf Polynomen mit Koeffizienten aus $GF(2^8)$. Statt also Koeffizienten, die ein Bit repräsentieren, repräsentiert in diesem Fall ein Koeffizient ein ganzes Byte. Für diese Polynome kann gezeigt werden, dass die Funktion $T_2(a)$ auch beschrieben werden kann, als $2 \cdot a$. Ähnliches gilt für $T_3(a)$ mit $3 \cdot a$ [10].

AddRoundKey In dieser Funktion wird jedes einzelne Zustandsbyte mit einem Byte aus dem Rundenschlüssel bitweise über ein exklusives Oder verrechnet. Der Rundenschlüssel ist ein ~~ein~~-erweiterter Chiffrierschlüssel. Die Berechnung des Rundenschlüssels wird im Anschluss erklärt. In jeder Runde kommt ein neuer Rundenschlüssel zum Einsatz, welcher die selbe Größe, wie das Zustands-Array hat. Die Formel dafür lautet:

$$b_{i,j} = a_{i,j} \oplus rk_{i,j}$$

b ist der neue Zustand, a der initiale und rk ist der Rundenschlüssel.

Der gesamte Ablauf der einzelnen Runden und angewendeten Transformationen ist auch grafisch in der Abbildung 16 im Anhang dargestellt. Dort steht ebenfalls, was bereits erwähnt wurde, nämlich dass aus dem ursprünglichen Chiffrierschlüssel die Rundenschlüssel ~~bilden~~.

Key Schedule Mit dem (Rijndael) Key Schedule werden die Rundenschlüssel ermittelt. Der erste Rundenschlüssel ist einfach der Chiffrierschlüssel. Die folgenden Rundenschlüssel berechnen sich alle gleich. Die Berechnung der ersten Spalte lässt sich wieder am besten mit Hilfe einer kurzen Formel beschreiben:

$$\begin{aligned} rk_{r,0,0} &= rk_{r-1,0,0} \oplus S - box[rk_{r-1,1,3}] \oplus round_const[r] \\ rk_{r,1,0} &= rk_{r-1,1,0} \oplus S - box[rk_{r-1,2,3}] \\ rk_{r,2,0} &= rk_{r-1,2,0} \oplus S - box[rk_{r-1,3,3}] \\ rk_{r,3,0} &= rk_{r-1,3,0} \oplus S - box[rk_{r-1,0,3}] \end{aligned}$$

r ist die aktuelle Runde, $round_const[1] = 1$ und $round_const[r] = T_2(round_const[r - 1])$. Es wird eine Rundenkonstante weniger benötigt, als Runden durchlaufen werden, da der erste Rundenschlüssel der Chiffrierschlüssel ist.

Da die Blocklänge bei AES auf 128 Bits begrenzt ist, hat der Rundenschlüssel eine Größe von 4 x 4 Byte. Die restlichen vier Spalten berechnen sich wie folgt:

$$rk_{r,i,j} = rk_{r-1,i,j} \oplus rk_{r,i,j-1} \quad \text{für } i = 0, 1, 2, 3 \text{ und } j = 1, 2, 3$$

Zusammengefasst wird die erste Spalte des neuen Rundenschlüssels berechnet aus der Verknüpfung von der ersten Spalte des vorherigen Rundenschlüssels mit der substituierten vierten Spalte aus dem vorherigen Rundenschlüssel. Außerdem wird vor der Substitution das oberste Byte zyklisch oben rausgeschoben und unten wieder reingeschoben. Zum Schluss wird das erste Byte noch mit einer Rundenkonstanten ~~er~~ verrechnet. Die restlichen drei Spalten des Rundenschlüssels generieren sich aus der exklusiven Oder-Verknüpfung der jeweiligen Spalte im vorherigen Rundenschlüssel und der vorherigen Spalte im aktuellen Rundenschlüssel.

Rundendurchlauf Es wurde schon erwähnt, dass AES-128, der bei Milenage genutzt wird, 10 Runden durchläuft. Die einzelnen Schritte der Runden und was vorher noch passiert wird dargestellt im Anhang 16: Grafische Darstellung der AES-Verschlüsselung auf S. 84. In der gezeigten Abbildung wird der Rundenschlüssel für die jeweilige Runde berechnet. Alternativ können in Runde 0 aber auch direkt alle Rundenschlüssel berechnet werden. Auf jeden Fall wird aber zu Beginn in Runde 0 die addRoundKey-Transformation durchgeführt.

In den Runden 1 bis 9 werden alle vier Operationen abwechselnd durchgeführt. Erst werden die Bytes substituiert, dann die Zeilen und Spalten vermischt und zum Schluss wieder der Rundenschlüssel addiert. Bis auf MixColumns werden ~~w~~ alle Transformationen auch in der letzten Runde getätigt. Der Grund dafür ist, dass MixColumns und AddRoundKey in der Reihenfolge vertauscht werden können, ohne dass sich das Endergebnis ändert. Nun ist es so, dass MixColumns nicht vom Schlüssel abhängig ~~ist~~, was bedeutet, dass Angreifer MixColumns zurückrechnen können ohne den Schlüssel zu kennen. Die Transformation in der letzten Runde anzuwenden würde also lediglich Rechenzeit benötigen, aber würde den Schlüssel nicht sicherer machen, weshalb die Transformation im letzten Schritt weggelassen wurde. [10]

2.5.6 Bewertung von AES

Wie in den vorherigen Kapitel aufgezeigt, verfolgt Rijndael einen innovativen und eleganten Ansatz gegenüber Twofish und Serpent. Da in der Wettbewerbsphasen keine Sicherheitslücken sichtbar wurden und die Verschlüsselungsgeschwindigkeit sehr schnell ist, setzte sich dieser durch. Nach einigen weiteren Jahren, in denen sich Kryptoanalytiker mit dem Algorithmus beschäftigt haben, ist Kritik an Rijndael aufgekommen. Rijndael gilt immer noch als sicher, aber der Sicherheitspuffer ist beträgt nur drei bis fünf Runden. Konkret bedeutet das, dass ein mit Rijndael verschlüsselter Text bis zur siebten Runde bei einer Schlüssellänge von 128 Bits geknackt werden kann. Bei 192 Bits braucht es acht Runden und bei 256 Bits neun von 14 Runden.

2.5.7 AES Visualization Tool

Die Implementierung des AES-Algorithmus wird neben genannter Literatur auch unter Zuhilfenahme eines Tool durchgeführt: das *AES Visualization Tool* der Michigan Technology Universität⁴³. Diese bietet neben AES auch Visualisierungswerkzeuge für die Kryptoalgorithmen DES, RSA, ECC, VIG und SHA an.

Für drei verschiedene Plattformen wird eine eigene Version bereitgestellt: Microsoft Windows, GNU Linux und Apple Mac OS.

Jede dieser angebotenen Versionen verfügt über zwei verschiedene Betriebsmodi: dem *Demo Modus* und den *Practice Modus*. Beide sind dazu entwickelt, den Ablauf des AES-Algorithmus für einen 128 Bit langen Textblock beziehungsweise Schlüssel zu verdeutlichen[12]. Jeder Modus verfügt über vier verschiedene Module:

- SubstituteBytes
- ShiftRows
- MixColumns
- AddRoundKey

Nachfolgend werden beide Betriebsmodi sowie untergeordnete Module genauer erläutert.

Demo Modus Dieser Modus besteht aus vier verschiedenen Teilkomponenten: dem Overview, der Encryption, der Decryption und der Key Expansion.

⁴³<http://www.cs.mtu.edu/~shene/NSF-4/AES-Downloads/index.html>

Eine zentrale Übersicht liefert die Teilkomponente Overview. Hier wird sowohl die Verschlüsselung als auch die Entschlüsselung schematisch dargestellt. Auf der linken Seite (von oben nach unten gelesen) die Runden 1-10 vom Klartext bis zum Ciphertext. Auf der rechten Seite (von unten nach oben gelesen) ebenfalls die Runden 1-10 vom generierten Ciphertext bis zum Klartext. Dazwischen werden alle notwendigen Zwischenschritte gelistet (SubstituteBytes, ShiftRows, MixColumns, etc.).

Ausgehend von dieser Sicht können alle notwendigen (oben genannten) Teilmodule abgearbeitet werden. Entweder durch die Go!-Funktion Schritt für Schritt oder manuell im entsprechenden Modul.

Die Verschlüsselung verfügt im Modul *SubstituteBytes* über eine Funktion, die automatisch einen zufälligen Klartext sowie Cipherkey generiert. Jeweils wird eine Länge von 128 Bit verwendet. ~~Selbige~~ können mit einem RoundKey zusammengeführt und dann unter Zuhilfenahme der S-Box in das ShiftRows-Modul übergeben werden.

Im *ShiftRows*-Modul wird lediglich visualisiert, welche Zeilen um welchen Wert verschoben werden. Ebenso wird das Ergebnis angezeigt.

Das Ergebnis wird im nachfolgenden Modul *MixColumns* als Eingabematrix mit einer weiteren Matrix gekreuzt. Einzelne Spalten können bei Bedarf separat ausgewählt und in der mathematischen Berechnung nachvollzogen werden. Die Ergebnismatrix übergibt die Daten an das Modul *AddRoundKey*. Hier wird der verschlüsselte Text offengelegt. Erneut besteht die Möglichkeit, einzelne Schritte durch Auswahl der Felder in der Matrix nachzuvollziehen.

Der gleiche Funktionsumfang liegt auch für den Prozess der Entschlüsselung vor. Lediglich in inverser Reihenfolge[12].

Practice Modus Dieser Modus ist dazu entwickelt, ~~um~~ den Ablauf des AES-Algorithmus sprichwörtlich zu üben. Im Ablauf entspricht dieser Modus dem des Demo Modus. Lediglich die Ergebnisse werden abweichend getriggert.

2.6 PPP

Zur Bereitstellung einer Punkt-zu-Punkt-Verbindung als Grundlage des Authentifizierungsvorgangs wird von Providern (ISP⁴⁴) die Implementierungen ~~eines~~ PPP verwendet.

⁴⁴Internet Service Provider

Mit Protokollen dieser Art wurden zum Beispiel schon Modem- oder ISDN-Verbindungen aufgebaut. Heutige Szenarien sind unter anderem auch GPRS- und UMTS-Datenverbindungen - hier hauptsächlich in Form von PPPoE. Auf beide Architekturen wird im folgenden genauer eingegangen.

2.6.1 Architektur PPP

PPP ist Teil der TCP/IP-Protokollsuite und sichert die komplette Funktionalität des Datalink-Layers. Es wurde hauptsächlich für den Betrieb von Modems entwickelt. Jede Maschine, die ein Modem in Betrieb hatte, nutzte bereits PPP um z.B. Internet im lokalen Netzwerk freizuschalten und zu verteilen. Neben der Freischaltung von Internetverbindungen wird PPP von vielen ISPs auch dazu verwendet, Zugriffe zu monitoren, sowie Angriffe durch Intrusion Detection zu vermeiden. In üblichen LAN⁴⁵-Umgebungen ist es notwendig, dass eingesetzte Technologien die Datalink-Layer-Funktion implementieren und darüber hinaus über einen MAC-Mechanismus verfügen, da verschiedene Quellen/-Ziele das selbe Medium teilen könnten. Dieser Regulierungsmechanismus ist bei PPP nicht notwendig, da es sich um eine Punkt-zu-Punkt beziehungsweise Ende-zu-Ende-Verbindung handelt. In jedem Fall handelt es sich um genau zwei Teilnehmer:

- Quelle
- Ziel

Motivation Die Architektur ist **gezielt** sehr simpel gewählt. Es werden lediglich IP-Datagramme zwischen den Endgeräten gekapselt. Vergleichbar ist der Aufbau von PPP mit dem von Ethernet, jedoch ohne die notwendige Behandlung vieler Probleme, die in sonstigen LAN- und Breitbandumgebungen auftreten können. So ist der Header z.B. nur 8 Byte statt 16 Byte lang. Doch dazu später mehr. PPP wurde als Alternative zum bereits bestehenden SLIP⁴⁶ implementiert, welches neben den notwendigen Methoden, dem multiplexen verschiedener Netzwerklayer-Protokolle, sowie mehrere Authentifizierungsmethoden noch zusätzliche Funktionen ermöglichen, die von PPP nicht benötigt werden.

PPP Frame Ein PPP-Frame ist wie folgt aufgebaut:



⁴⁵Local Area Network

⁴⁶Serial Line Internet Protocol

- flag (1 Byte) - hexadezimal - Funktion des Paketdelimiter
- address (1 Byte) - hexadezimal (FF) - Indikator für 'adressiert an alle Stationen'
- control (1 Byte) - hexadezimal (03) - identifiziert Paket als HDLC⁴⁷
- protocol (2 Byte) - hexadezimal - identifiziert erwünschtes bzw. eingesetztes Protokoll
 - 0xxx bis 3xxx : Netzwerklayer-Protokolle
 - 4xxx bis 7xxx : Low Level Netzwerklayer Protokolle ohne NCP⁴⁸
 - 7xxx bis bxxx : Low Level Netzwerklayer Protokolle mit NCP
 - cxxx bis fxxx : Link Layer Protokoll wie LCP und zusätzliche Authentifizierungsprotokolle
- data and pad (variabel, maximal 1.500 Byte)
- frame check sequence (2 Byte oder 4 Byte)
- flag (1 Byte)

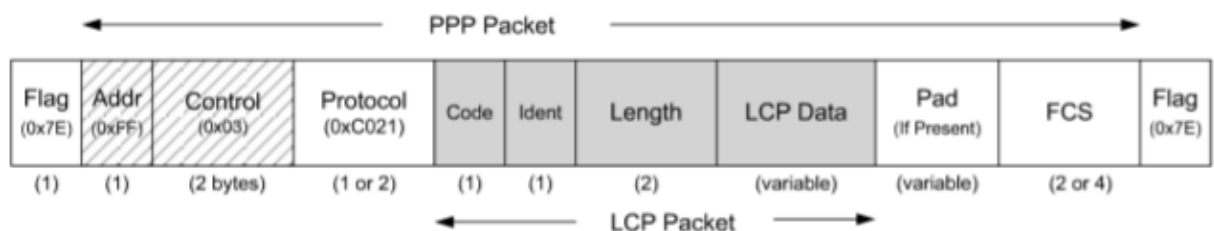


Abbildung 5: Aufbau eines PPP-Frames [13]

Diverse oben genannte Felder können in ihrer Länge variieren, da diese während des Verbindungsaufbaus vom LCP⁴⁹ ausgehandelt werden.

LCP Frame Um die effizienteste Verbindungsart zu finden, benutzen PPP-Systeme immer LCP, welches korrekte Parameter aushandelt. LCP-Nachrichten in ausgetauschten PPP-Frames enthalten somit alle Konfigurationsoptionen für die sich gerade aufbauende Verbindung. Ist eine Konfiguration gefunden, die beide Knoten unterstützen, folgt der

⁴⁷High Level Data Link Control

⁴⁸Network Control Protocol

⁴⁹Link Control Protocol

Link-Establishment-Prozess. Ist dieser erreicht, müssen danach keine weiteren redundanten Paketinformationen im Header mitgetragen werden.

Ein LCP-Frame ist wie folgt aufgebaut:

- code (1 Byte) - hexadezimal - enthält den Messagetyp (als Codes spezifiziert)
- identifier (1 Byte) - hexadezimal - mit diesen werden Anfragen bzw. Antworten mit einzelnen LCP-Transaktionen in Verbindung gebracht
- length (2 Byte) - hexadezimal - beinhaltet die Länge der Nachricht (inklusive code, identifier, length, data)
- data (variabel) - hexadezimal - Nutzdaten

LCP ist so entworfen, dass Hersteller ihre eigenen Optionen einsetzen können, ohne selbige explizit über IANA⁵⁰ spezifizieren zu müssen. Dokumentiert ist dies in **RFC2153**.

Aufbauphasen Nachfolgend werden die verschiedenen Aufbauphasen des PPP-Protokolls erläutert. Im Anhang 18: Aufbauphasen einer PPP-Verbindung auf S. 85 befindet sich eine Abbildung, die diesen Vorgang illustriert.

Link Dead Phase: Quell- und Zielsystem fangen mit dieser Phase an und enden hiermit wieder. Grundlage ist, dass außer (maximal) dem Link auf physischer Ebene, keine Verbindung zwischen beiden Endpunkten besteht. Normalerweise wird nach sicherstellen des physischen Links von einer Seite der Aufbau der Verbindung initiiert. Dies geschieht meist mit einer Form von Modem. Nach Abschluss der Initiierung beginnt die nachfolgende Phase.

Link Establishment Phase: Das initiiierende System sendet eine LCP-Nachricht an das Zielsystem, um Optionen anzufordern, die gesetzt werden sollen. Dazu gehören Netzwerklayer-Protokoll, Authentifizierungsmethode und andere optionale Funktionen. Sofern das Zielsystem alle angeforderten Optionen beherrscht, kann dieses eine Bestätigung (**ACK**) an das Quellsystem senden. Ist dies nicht der Fall, wird eine Antwort verfasst, die sowohl alle *nicht unterstützten* als auch alle unterstützten Optionen enthält, damit das Quellsystem nach Empfang dieser Information eine Verbindung initiieren kann, die in jedem Fall von beiden Seiten unterstützt wird. Das erfolgreiche Abschließen dieser Phase führt zur nächsten Phase.

Authentication Phase: Diese Phase ist optional. Ausgelöst wird sie durch das Vorhan-

⁵⁰Internet Assigned Numbers Authority

densein einer Authentifizierungsoption in der LCP-Konfigurationsnachricht. Zur Auswahl stehen z.B. PAP⁵¹ oder CHAP⁵². Hierbei greift PAP auf Nutzernamen und Passwort, CHAP auf einen komplexeren Informationsaustausch mit einem Challenge-Response-Verfahren zurück. Der Erfolg führt immer zur nächsten Phase ~~führt~~, aber die Reaktion bei Misserfolg des Vorgangs ist protokollabhängig.

Link Quality Monitoring: Diese Phase ist wie ihr Vorgänger ebenfalls optional - ebenfalls ausgelöst durch die gewählte Option in der LCP-Nachricht. Hier wird aus mehreren Protokollen gewählt. Eines davon ist standardisiert: das 'Link Quality Report Protocol'. Registriert werden unter anderem der Linktraffic sowie Fehlermeldungen.

Network Layer Protocol Configuration: Wie bereits erwähnt, unterstützt PPP das Multiplexen von Protokollen auf Netzwerkebene. Für jedes einzelne, das eingesetzt wird, führt das System einen separaten Prozess des Verbindungsaufbaus durch. Jedes Netzwerklayersprotokoll verfügt über ein eigenes NCP sowie IPCP⁵³. Vergleichbar ist dies mit dem Aufbau von LCP - nur spezifischer.

Link Open Phase: Nachdem alle individuellen Optionen und NCP-Exchanges erfolgreich durchgeführt wurden, ist der Verbindungsaufbau komplett und Protokolldaten können jetzt über den aufgebauten Link in beide Richtungen ausgetauscht werden.

Link Termination Phase: Wird die Verbindung absichtlich (Ablauf der Session, Authentifizierungsfehler) oder durch Fehler o.ä. getrennt, wird im Regelfall über LCP eine 'Terminate Request Message' versandt. Diese kann von der Gegenseite angenommen (AKC) werden, sofern die grundlegende Verbindung noch aktiv ist. Beide Systeme sind dann wieder in der ursprünglich genannten 'Link Dead Phase'. Eine Terminierung der Verbindung ist neben LCP auch auf NCP-Ebene möglich, damit die PPP-Verbindung trotz 'Terminierung' bestehen bleibt.

2.6.2 Architektur PPPoE

Zur Realisierung der Verbindung zwischen Authentication Center und Endgerät wurde in diesem Projekt aufgrund der Beschaffenheit des Raspberry Pi eine Ethernetverbindung gewählt. Dieser wird das im vorangegangenen Abschnitt erläuterte PPP-Protokoll zugrunde gelegt. In Ethernetframes werden PPP-Daten als Nutzdaten gekapselt. Diese

⁵¹Password Authentication Protocol

⁵²Challenge Handshake Authentication Protocol

⁵³Internet Protocol Control Protocol

Methode wurde auch im realen Umfeld dazu entwickelt, ISPs die Möglichkeit zu geben, Verbindungen über Kabelmodem oder DSL in Form von Bridged-Topologien zu realisieren. Provider bewerkstelligen so auch die Endpunktidentifikation, Accounting und Rechnungserstellung (beschrieben wird der Standard in RFC2516).

Aufbauphasen Diese sind im Kontext einer PPPoE-Verbindung Discovery und Session. Beide werden nachfolgend erläutert.

Discovery Der Client verwendet PPPoE-Frames in der Discovery-Phase dazu, einen Zugangspunkt zu finden.

Dies geschieht in den folgenden Schritten bzw. Frames:

1. PPPoE Active Discovery Initiation (PADI) - Ein Frame, vom Client gesendet an die Broadcastadresse 0xFF-FF-FF-FF-FF-FF. Falls vorhanden, werden weitere Parameter als Payload mitgeschickt. (Codefeld:9; Session-ID:0)
2. PPPoE Active Discovery Offer (PADO) - Ein Frame, der von der Authentifizierungsstelle an die Unicast MAC-Adresse des initiiierenden Client geschickt wird. Weitere Parameter wie Service-Name o.ä. können ebenfalls mitgeschickt werden. (Codefeld:7; Session-ID:0)
3. PPPoE Active Discovery Request (PADR) - Ein Frame, der vom Client an die Unicast MAC-Adresse der Authentifizierungsstelle geschickt wird. (Codefeld:25; Session-ID:0)
4. PPPoE Active Discovery Session-Confirmation (PADS) - Ein Frame, der von der Authentifizierungsstelle an die Unicast MAC-Adresse des Client geschickt wird. Er enthält alle ausgehandelten Daten mit der zugewiesenen Session-ID. (Codefeld:101; Session-ID:XX)
5. PPPoE Active Discovery Terminate (PADT) - Ein Frame, der von beiden Endpunkten geschickt werden kann. Er signalisiert die gewünschte Verbindungsterminierung des Absenders. (Codefeld:167)

Session In der Session-Phase, ist die PPPoE-Verbindung bereits erfolgreich aufgebaut und Daten können ausgetauscht werden. Dieser Zustand ist erreicht, sobald die Discovery-Phase erfolgreich abgeschlossen ist.

2.6.3 Roaring Penguin PPPoE

Die eingesetzte Software zur Realisierung der PPPoE-Verbindung ist der Roaring Penguin⁵⁴ PPPoE-Server.

Er implementiert den PPPoE-Standard (2.6.2: Architektur PPPoE auf S. 36) auf Basis von *TCP/IP* und *PPP* als eine *Userland*-Anwendung. Dementsprechend ist es nicht notwendig hierfür den Kernel zu patchen und neu zu bauen. Ein einfaches installieren und konfigurieren genügt.

pppd Der im Userland installierte Dienst heißt `pppd`. Um eine Verbindung aufzubauen, erstellt er eine virtuelle Netzwerkschnittstelle (der Form `pppX`) und verknüpft diese mit einem Pseudo-tty⁵⁵. Typischerweise handelt es sich hier um einem seriellen Port [14]). Daten, die von diesem tty-Device empfangen werden, erscheinen durch das `pppX`-Device. Logisch unterhalb des tty-Device wird dann über PPPoE mit einem ebenfalls durch den Server assoziierten Ethernet-Interface kommuniziert (`ethX`).

Der Informationsfluss findet somit in nachfolgender Hierarchie statt:

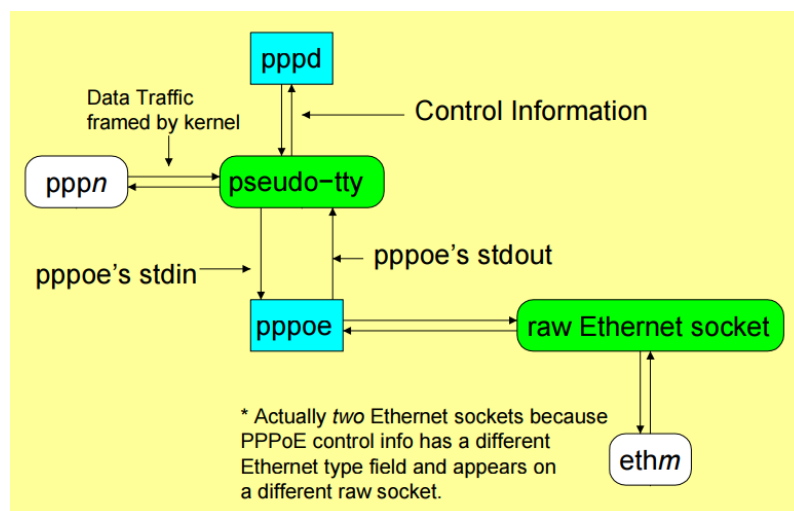


Abbildung 6: Roaring Penguin PPPoE Devicehierarchie [14]

Ausgehend vom `ttyX`-Device kommen Daten in HDLC-Frames zum Ethernet-Device.

⁵⁴<https://www.roaringpenguin.com/products/pppoe>

⁵⁵tele-typewriter

Eingehende Pakete werden beim Ethernet-Device gerahmt und auf dem stdout an das ttyX-Device weitergeleitet.

Kontroll- und Dateninformationen können bei Bedarf über unterschiedliche Ethernet-Devices ausgetauscht werden.

2.7 PC/SC

Multitasking- und Multiuserbetrieb in modernen Betriebssystemen erfordert auch das Bereitstellen eines Standards, um den Zugriff auf Chipkarten mit mehreren (parallel arbeitenden) Teilnehmern zu organisieren. Ein Standard, der sich mit genau dieser Problemstellung befasst, ist der *PC/SC*⁵⁶-Standard. Er abstrahiert die Kommunikation mit der Chipkarte so, dass die Anwendung keine genaueren speziellen Informationen zur verwendeten Karte benötigt. Lediglich die Kommunikation mit dem Standard konformen Lesegerät muss durch den Treiber sichergestellt werden.

Entwickelt wurde PC/SC von verschiedenen Herstellern, hauptsächlich Gemalto, Microsoft, Infineon und Toshiba.

2.7.1 Spezifikation und Aufbau der Schnittstelle

Die Spezifikation definiert Schnittstellen, die den Zugriff auf Chipkarten ausgehend von mehreren Applikationen beziehungsweise Nutzern ermöglichen. Grundlegend dafür benötigt werden neben dem Standardkonformen Treiber ein IFD⁵⁷ und eine *PC/SC*-konforme Chipkarte (ICC⁵⁸ nach ISO7816-1,2 und 3).

Der Unterschied zum ähnlichen CT-API-Standard fängt in der Ebene des *ICC Resource Managers* an. Also der Abstraktion der APDU-Schnittstelle in der Chipkarten-middleware [1]. Selbiger Manager organisiert sowohl die Vergabe von Terminals inklusive der vorhandenen Chipkarte, als auch das Zuordnen zusammenhängender APDU-Kommandoketten.

Losgelöst davon können Daten auf der Chipkarte abgelegt werden. Im Umfeld der SIM-Karte sind dies Kontakte, SMS, o.ä. Hierfür zuständig ist der *ICC Service Provider*.

⁵⁶Personal Computer / Smart Card

⁵⁷Interface Device

⁵⁸Integrated Chip Card

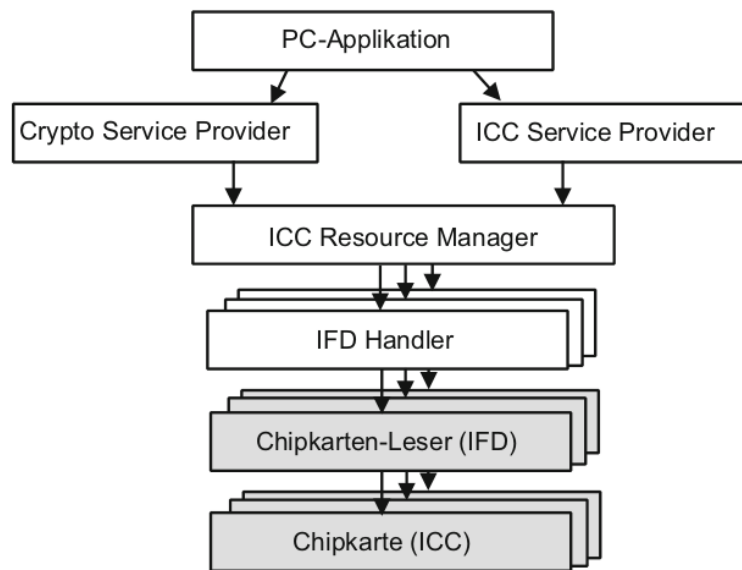


Abbildung 7: Architektur des PC/SC-Standards [1]

Ebenfalls dazu gehören administrative Dateisystemzugriffe wie das Ändern des PIN-Codes.

Der *Cryptographic Service Provider* ist die ausführende Komponente der auf der Chipkarte implementierten Algorithmen zur Authentifizierung mittels Kryptoalgorithmen. Auslöser für die Aufteilung in zwei Schnittstellen - *Crypto Service Provider* und *ICC Service Provider* sind rechtliche Grundlagen. In manchen Staaten ist der Import von Verschlüsselungstechnologien auf diesem Weg untersagt.

Unter Einhaltung des Standards ist es mit oben genannter Architektur möglich, dass parallel aus verschiedenen Anwendungen auf unterschiedliche Konstellationen von Chipkarten und Lesern zugegriffen wird.

Sowohl unter Windows als auch in vielen Linux-Distributionen sind *PC/SC-Treiber* mittlerweile verbreitet.

2.7.2 PCSClite

Die unter Linux-Distributionen verfügbare Version ist PCSClite⁵⁹. Das von David Corcoran gestartete Projekt M.U.S.C.L.E (Movement for the Use of Smart Cards in a Linux Environment) wird mittlerweile hauptsächlich von Ludovic Rousseau weitergeführt[15].

Neben diversen Linux-Distributionen werden auch Mac OSX und diverse BSD- sowie Unix-Derivate unterstützt.


2.8 Die Sprache C

C ist eine prozedurale Programmiersprache, die nicht für einen speziellen Anwendungsfall entwickelt wurde. Sie hat einen hohen Verbreitungsgrad und ist laut IEEE⁶⁰ Spectrum⁶¹ und RedMonk⁶² unter den 10 beliebtesten Programmiersprachen. C ist eine sehr hardwarenahe Sprache und wird deshalb oft als Ersatz für Assembler eingesetzt. In diesem Kapitel soll kurz auf die Geschichte eingegangen werden, was es bedeutet, dass die Sprache prozedural ist und welche Besonderheiten C aufweist.

2.8.1 Geschichte

Die Entwicklung von C hängt stark zusammen mit der Entwicklung des Unix-Systems. Dieses wurde ursprünglich in Assembler geschrieben und nun wurde überlegt, den Code neu zu schreiben, allerdings in B. Mit B ließen sich jedoch einige der Unix-Features nicht umzusetzen, wie beispielsweise die Byte-Adressierung.

Deshalb wurde 1972 die Sprache C von Dennies Ritchie bei Bell Labs entwickelt. Die Portabilität war dabei zu Beginn nicht beabsichtigt, stattdessen war C nur für Unix-Systeme entwickelt worden. Allerdings wurden schnell Compiler auch auf andere Systeme portiert. 1973 war C dann leistungsfähig genug, ~~dass der Großteil des Unix-Kernels in C geschrieben war.~~

C wurde Ende der 70er auf verschiedenen Rechensystemen, wie Mainframecomputer und Heimrechnern von plementiert. Dies bewegte vermutlich das American National

⁵⁹<https://pcsc-lite.alioth.debian.org/>

⁶⁰Institute of Electrical and Electronics Engineers

⁶¹<http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>
[besucht am 17.05.16]

⁶²<http://redmonk.com/sograde/2016/02/19/language-rankings-1-16/> [besucht am 17.05.16]

Standards Institute dazu, 1983 eine Komitee zu gründen, was sich um eine Standardisierung von C kümmert. Das Komitee, X3J11, veröffentlichte 1989 seinen ersten Standard, der oft auch als ANSI C, Standard C oder C89 bekannt ist.

Nach der Weitergabe des Standards an die ISO⁶³ 1990 wurde dieser weiterentwickelt und führte zu zwei weiteren Standards in 1999 (C99) und 2011 (C11). Unterstützt wird von den Compilern C89 und oft auch C99, wohingegen C11 nicht sehr verbreitet ist.

2.8.2 Prozedurales Programmierparadigma

In der Programmierung gibt es verschiedene Stile, wie Programmcode geschrieben und strukturiert werden kann. Dabei gibt es einige fundamentale Stile, die als Programmierparadigmen bezeichnet werden. Die vier wichtigsten Paradigmen sind das imperative, funktionale, logische und objekt-orientierte Programmierparadigma [17]. Das prozedurale Paradigma ist eine Spezialisierung des imperativen Paradigma. Programmiersprachen können teilweise nur mit einem Paradigma entwickelt werden, aber oft gibt es auch mehrere Ansätze. So kann C beispielsweise auch objekt-orientiert geschrieben werden, aber eigentlich gilt es als prozedurale Programmiersprache.

Die Idee der prozeduralen Programmierung ist, dass Aufgaben nacheinander erledigt werden, wie sie aufgeschrieben sind. Im Unterschied dazu wird zum Beispiel in der objekt-orientierten Programmierung das Programm in Klassen aufzuteilen, welche Funktionen zur Verfügung stellen. Bei einer prozeduralen Sprache werden die Funktionen in Bibliotheken nach ihren Aufgabengebieten gruppiert und aus dem Hauptprogramm heraus abgerufen. Bei C ist dieses Programm in einer "main"-Funktion enthalten und über diese Funktion wird das Programm gestartet.

2.8.3 Besonderheiten

C ist eine "low-level"-Sprache, was an sich nichts schlechtes ist, aber es bedeutet, dass C mit den Objekten arbeitet, mit denen auch die meisten Computern arbeiten. Konkret bedeutet das, dass C nur Zeichen, Zahlen und Speicheradressen nutzt. Durch arithmetische und logische Operationen, die in der Maschine implementiert sind, lassen sich diese Datentypen kombinieren und bewegen. Daraus resultiert, dass keine Zeichenketten wie in höheren Programmiersprachen existieren oder Funktionen, die auf Zeichenketten und

⁶³Internationale Organisation für Normung

Arrays angewendet werden können. [18]

Welche Besonderheiten diese Dinge mit sich bringen soll im folgenden aufgezeigt werden.

Pointers Pointers sind Variablen, die die Speicheradressen von Variablen speichern. Sie werden in C häufig angewendet, da sie oft die einzige Möglichkeit sind eine Berechnung durchzuführen und teilweise auch, weil sie zu kompakterem und effizienterem Code führen als andere Ansätze. [18] Pointers sind also eine große Hilfe, aber da sie Speicheradressen speichern sind sie auch nicht ganz einfach zu verwenden und können zu unerwarteten Ergebnissen führen, wenn auf eine falsche Speicheradresse zugegriffen wird. Wie Abbildung 8 dargestellt ist in der Variable *c* ein Wert abgespeichert und in der Pointervariable *p* ist die Adresse in der die Variable *c* liegt gespeichert.



Abbildung 8: Konzept von Pointers [18]

Im C-Code wird das wie folgt realisiert:

```
int c = 12;
int* p = &c;
```

Durch den `*` wird ein Pointer deklariert, in diesem Falle auf eine Integer-Variable. Der `&`-Operator liefert dann die Adresse der Variable *c*.

Bitweise Operatoren Um einzelne Bits zu manipulieren ist es am einfachsten mit bitweisen Operatoren zu arbeiten. In vielen Sprachen existieren Möglichkeiten, um diese durchzuführen, aber sie werden selten genutzt. Der Grund dafür ist, dass in der Regel keinen einzelnen Bits manipuliert werden müssen, um eine gegebene Aufgabe zu erledigen. Da es in C aber nur wenige Funktionen gibt, sind sie in den meisten Programmen zwingend erforderlich. Es werden die sechs Operationen Und (`&`), Oder (`|`), Exklusiv-Oder (`^`), Verschiebung nach links (`<<`) oder rechts (`>>`) und das Komplement (`~`) von C unterstützt.

Bibliotheken Zum Schluss sollte noch etwas zu den Bibliotheken gesagt werden. In prozeduralen Sprachen lassen sich Funktionen in Bibliotheken bündeln. Da C jedoch von Haus aus nur wenige Funktionen bereitstellt, gibt es nur wenige C-Programme, die ohne eine externe Bibliothek auskommen. So gibt es einige sogenannte Standardbibliotheken, die von der ISO und ANSI C spezifiziert wurden. Durch diese Bibliotheken wird der Umfang von C erweitert und abhängig vom Betriebssystem gibt es noch andere Bibliotheken für verschieden Anwendungsfälle.

Zusätzlich zu der Tatsache, dass C viele Bibliotheken einbindet, besitzen diese typischerweise auch eine Header-Datei. Die Header-Datei enthält die Prototypen der Funktionen, die vom Programm genutzt werden können, denn auch eine Bibliothek kann Hilfsfunktionen enthalten. Diese werden dann nicht in der Header-Datei notiert. Neben den Funktionsprototypen enthält die Header-Datei spezielle Datentypen und Konstanten, die für die Verwendung der Bibliothek notwendig sind. Die Header-Datei wird zu Beginn eines C-Programms eingebunden und der Compiler kann auf Basis der Header-Datei eine Überprüfung durchführen, ob alle genutzten Funktionen seitens des Programms in einer eingebunden Bibliothek zur Verfügung stehen.

2.9 Die Sprache Python

2.9.1 Grundlagen

Python ist eine seit 1991 frei veröffentlichte Programmiersprache. Sie zählt zu den interpretierten Programmiersprachen. Primäre Ziele der Sprache sind zusammengefasst:

- Lesbarkeit
- Kompaktheit

Somit wird neben etablierten Sprachen eine Alternative für sowohl umfangreiche Softwareprojekte als auch kleinere Einsatzgebiete wie Skripting im Heimcomputerbereich angeboten.

Python konzentriert sich nicht ausschließlich auf ein einziges Programmierparadigma. Es unterstützt objektorientierte, funktionale sowie prozedurale Programmierung. Zusätzliche Paradigmen (unter anderem auch logische) können über Erweiterungen nachgerüstet werden. Weiterhin verfügt *Python* auf dynamische Typisierung und automatisches Speichermanagement. Aus diesen Gründen ist *Python* vor allem auch für Einsteiger in das Thema

der Programmierung gut geeignet. ~~Nichts desto Trotz~~ ist *Python* ~~auch~~ aufgrund seiner hohen Flexibilität ~~bei~~ erfahrenen Entwicklern verbreitet.

In vielen Linux-Distributionen gehört *Python* im gewissem Umfang zur Standardinstallation. Mit dem ausgelieferten Interpreter können ~~plattformübergreifend~~ Programme ausgeführt werden. Bei Bedarf ist es möglich ~~auf~~ eine Große Anzahl von Paketen in Standardbibliotheken zuzugreifen.

In seltenen Fällen kann es auch notwendig sein ~~Binärcode~~ auszuliefern, ohne den *Python*-Interpreter zu installieren. Hierfür existieren Tools von Drittanbietern, die aus vorliegendem *Python*-Code ausführbare Programme generieren. Ein Beispiel hierfür ist *py2exe*⁶⁴.

2.9.2 Motivation zur Nutzung

Die Entscheidung für diese Programmiersprache fällt vor allem ~~aus dem Grund, dass~~ bereits eine Vielzahl von Skripten aus kleineren Projekten im Bereich der Kommunikation mit *SIM*-Karten (also auf der Seite des UE) existieren. An diese Skripte kann angeschlossen werden. Nachfolgend werden die zugrunde liegende Skripte näher erläutert. Diese werden für die Umsetzung dieser Studienarbeit verwendet sowie in ihrer Funktion erweitert.

2.10 Bibliotheken

2.10.1 pysim

Pysim⁶⁵ ist ein Python-Skript, geschrieben von Kevin Prince, welches Operationen direkt auf der *SIM*-Karte implementiert. ~~Sowohl~~ Kryptoalgorithmen wie Milenage als auch administrative Dateisystemzugriffe bis hin zur Verwaltung von Telefonbuch, SMS oder ähnlichem. Unterstützt wird der GSM-Authentifizierungsvorgang.

Pysim kennt verschiedene Modi und nimmt dementsprechend unterschiedliche Parameter entgegen. Verfügbare Parameter sind Chipkartentyp, verwendetes Device (im Betriebssystem `/dev/ttyX`) und Baudrate[19].

Über den normalen Betrieb einer *SIM*-Karte (also Authentifizierung, Auslesen, etc.) ist es mit Pysim auch möglich ~~Werte~~ auf programmierbaren *SIM*-Karten zu setzen.

⁶⁴<http://www.py2exe.org/>

⁶⁵<https://github.com/kevinprince/pysim>

Neben abgeschlossenen Skripten ist auch die interaktive Nutzung auf der Python-Shell verwendbar.

2.10.2 osmo-sim-auth

*Osmo-sim-auth*⁶⁶ überschneidet sich im Umfang angebotener Funktionen mit *pysim*. Hauptsächlich ist es, genau wie *pysim* eine Implementierung in Python, um den Authentifizierungsprozess einer Chipkarte zu steuern. Es basiert ebenfalls auf der Grundlage des *PCSClite*-Treibers und der *pyscard*-Library.

Ein Unterschied zwischen beiden python-Skripten stellte sich bei der Realisierung der Kommunikation in der Praxis heraus: Während *pysim* auch die Programmierung von Chipkarten anbietet, ist *osmo-sim-auth* dazu in der Lage, über GSM- hinaus auch eine UMTS-Authentifizierung zu leiten. Der jeweilige Modus wird über Parameter übermittelt. Dementsprechend verfügt *osmo-sim-auth* auch über die Funktion der Resynchronisation bezüglich der SQN-Nummer[20].

Entwickelt wird es vom Osmocom OpenBSC-Projekt, welches sich damit beschäftigt, eine (A)GPL-lizenzierte Implementierung für den GSM/3GPP-Protokollstack zu erzielen. Neben der Authentifizierung in direkter Kommunikation mit der SIM-Karte werden auch folgende Funktionalitäten durch das Projekt realisiert: BSC⁶⁷, MSC, HLR, AuC, VLR, und EIR⁶⁸[21].

2.10.3 pyscard

Zur Verwendung des Tools *pysim* oder *osmo-sim-auth* wird eine Python-Bibliothek benötigt, die Smartcard-Support für Python bereitstellt. Diese Bibliothek ist *pysim*. Sie fungiert als Schnittstelle zwischen dem Treiber *PC/SC* beziehungsweise *PCSClite* und der Python-Anwendung (dargestellt im Anhang 19: Architektur der *pyscard*-Bibliothek auf S. 86).

Implementierte Funktionen sind das Prüfen der gesteckten bzw. nicht gesteckten Chipkarte, der darauffolgende Verbindungsaufbau sowie das Versenden und Empfangen von APDUs in hexadezimaler Darstellung. Nach erfolgreichem Verbindungsaufbau kann somit eine Kommunikation via APDUs stattfinden.

⁶⁶<http://openbsc.osmocom.org/trac/wiki/osmo-sim-auth>

⁶⁷Base Station Controller

⁶⁸Equipment Identity Register

Veröffentlicht wurde pycard als freie Software unter der *GNU Lesser General Public License*.

2.11 Ubuntu

Als Grundlage für die Umsetzung des Providers wird die Linuxdistribution Ubuntu⁶⁹ gewählt. In Desktop- sowie Serverumgebungen wird es zunehmend eingesetzt. Unter anderem, da eines der Ziele des Projektes ist, eine stabile 'Out-Of-The-Box'-Installation zu liefern. Neben der intuitiven Installation und dem Betrieb wird auch jeweils eine *LTS*-Version der aktuellen Veröffentlichung angeboten. Hierbei steht das Akronym *LTS* für Long Term Support und garantiert dem Anwender eine Updateversorgung (an Paketen) über insgesamt fünf Jahre hinweg. Ebenso kann bei Bedarf separat technischer Support direkt von Canonical Ltd. bezogen werden. Ubuntu's initiale Veröffentlichung (Version 4.10) geht bis ins Jahr 2004 (20. Oktober) zurück und stammt von Debian GNU/Linux ab - es zeichnet sich allerdings durch eine ebenso breite, allerdings darüber hinaus aktuellere Paketauswahl aus.

Motivation zum Einsatz von Ubuntu in der Version 14.04 LTS ist die bestehende Erfahrung mit dieser Version in Verbindung mit dem PPPoE-Server von Roaring Penguin (Version 4.11).

2.12 Raspberry Pi

Die Hardwaregrundlage für das Endgerät bietet die Plattform eines Raspberry Pis, einem Kleincomputer.

2.12.1 Raspberry Pi Foundation

Entwickelt wird der Raspberry Pi von der Raspberry Pi Foundation⁷⁰ (registriert in Großbritannien). Ziel der Organisation ist es, seit Veröffentlichung des ersten Modells, einen kostengünstigen Computer vorrangig für Bildungszwecke zu entwerfen. Auf diesem Weg soll sowohl Erwachsenen als auch Kindern der Zugang zum Programmieren oder anderen wissenschaftlichen Anwendungsgebieten erleichtert werden.

⁶⁹<http://www.ubuntu.com/>

⁷⁰<https://www.raspberrypi.org>

Bisher wurden seit der initialen Veröffentlichung im Februar 2012([22]) drei Generationen in unterschiedlichen Ausführungen entwickelt. Die Bezeichnung A(+) beziehungsweise B(+) gibt jeweils Aufschluss über die jeweilige Ausführung.

2.12.2 Hardware Modell B

Das in diesem Projekt eingesetzte Modell B der ersten Generation verfügt über folgende Hardwarekomponenten:

- CPU - 700 MHz Singlecore ARM1176JZF-S
- RAM - 512 MB
- Speicherslot - SDHC
- Grafikprozessor - Broadcom VideoCore IV

2.12.3 Betriebssystem

Als Betriebssystem gibt es für den Raspberry Pi eine breite Auswahl. Neben einer Vielzahl von Media-Center-Plattformen sind auch alle Desktop- beziehungsweise Server-Versionen gängiger Linux-Derivate verfügbar: Arch Linux, Puppy Linux, Raspbian, openSUSE, Gentoo Linux, Ubuntu Mate, CentOS, Slackware, ...

Raspbian Aufgrund hoher Stabilität und Verfügbarkeit aller benötigten Treiber (u.a. dem SIM-Kartenleser) wurde die auf *Debian GNU Linux*⁷¹ basierende Distribution *Raspbian*⁷² ausgewählt. Es erbt somit alle Eigenschaften vom übergeordneten Debian-projekt. So auch den Paketmanager *dpkg* mit ca. 35.000 vorkompilierten Softwarepaketen - in einigen Fällen für den Betrieb mit dem Raspberry Pi optimiert. Nicht vorpaketierte Software kann auf dem Raspberry Pi durch Vorhandensein einer Vielzahl von Libraries und Build-Tools für die ARM-Architektur kompiliert werden.

Über den rein funktionalen Konsolenbetrieb hinaus, wird Raspbian standardmäßig mit den Windowmanagern XFCE oder LXDE ausgeliefert.

Nachdem im Juni 2012([23]) die erste Version fertiggestellt wurde ist Raspbian nach wie vor aktiv in Entwicklung. Momentan im stabilen Debian-Release *Jessie*. Selbiges wird

⁷¹<https://www.debian.org/>

⁷²<https://www.raspbian.org/>

auch zur Umsetzung des Endgerätes eingesetzt. Die vorkompilierten Pakete stammen dementsprechend aus dem aktuellen *stable*-Zweig, der häufig vor allem im Umfeld des Serverbetriebes anzutreffen ist. Die Ursache hierfür ist die Tatsache, dass das Projekt Pakete erst nach einem langen Testprozess im *unstable*- sowie darauffolgenden *testing*-Zweig für den *stable*-Zweig freigibt. Im Kontrast zu gängigen Desktop-Distributionen, die im Vergleich zu Debian stärker auf Aktualität achten.

Neben den Zielen der Raspberry Pi Foundation verfolgt das Projekt auch die Ziele des Debian/GNU-Projekts. Es ist dementsprechend unter den *Debian Free Software Guidelines*⁷³ lizenziert und durch die Raspbian-Community unabhängig entwickelt.

2.13 VirtualBox

Die eingesetzte Virtualisierungsplattform zur Simulation des Providers ist VirtualBox⁷⁴. Das Betriebssystem Ubuntu, der PPPoE-Server und weitere Software für das Bereitstellen des Providers wird innerhalb einer virtuellen Maschine realisiert.

2.13.1 Grundlage

VirtualBox ist ein open-source Typ-2 Hypervisor (2.13.2: Hypervisor auf S. 50), der ursprünglich von der *innotek GmbH*, gefolgt von *Sun Microsystems* entwickelt wurde. Damals noch *Sun VirtualBox* genannt, wurde es von *Oracle* übernommen und ist nach wie vor frei zugänglich (nach der GNU GPL2 Lizenz) [24].

VirtualBox bietet den Hypervisor für Hostplattformen, eine API und ein SDK zur Bereitstellung von Werkzeugen für Gäste an. Es wird als Wirtssystem auf sowohl 32-Bit als auch 64-Bit für x86-Architekturen angeboten.

Nachfolgend werden die wichtigsten Komponenten zur Realisierung der VirtualBox-Umgebung erläutert.

Wirtssystem wird das Betriebssystem genannt, welches die Grundlage für den Hypervisor liefert. Es gehört entweder selbst zum Hypervisor oder intergeriert diesen so, dass Gastssysteme auf ihm betrieben werden können. Beide Typen werden unter 2.13.2: Hypervisor auf S. 50 genauer erläutert.

⁷³<https://wiki.debian.org/DFSGLicenses>

⁷⁴<https://www.virtualbox.org/>

Bezeichnung	Beschreibung
vboxsrv	Der Prozess, der die Allokation aller Ressourcen verwaltet.
VBoxSVC	Der VirtualBox-Serviceprozess. Er überwacht den laufenden Betrieb von virtuellen Maschinen
vboxzoneaccess	Der solarisspezifische Daemon, um VirtualBox-Devices von Oracle Solaris Containern aus anzusprechen.
VBoxXPCOMIPCD	Der Prozess, der (nicht auf Windows-Hostsystemen) Interprozesskommunikation abwickelt.
VirtualBox	Der Clientdienst, der Ressourcenlimits definieren lässt und diese verwaltet.

Tabelle 3: Prozessarchitektur von VirtualBox

Gastsystem wird das Betriebssystem genannt, welches virtualisiert auf dem Hypervisor abgebildet wird. Es spricht die abstrahierten, virtuellen Hardwarekomponenten an, als würde es sich um ein physikalisches System handeln.

Virtuelle Maschine (VM) benennt die Instanz aus abstrahierter Hardware und einem darin installierten Gastsystem. Hier werden die virtuellen Komponenten (CPU, RAM, HDD, etc.) definiert und dem Gastsystem bereitgestellt.

Gasterweiterungen werden dazu benötigt, die Integration der Gastsysteme zu verbessern. Diese sind von Gastsystem zu Gastsystem verschieden und müssen deshalb gesondert ausgeliefert werden. Dazu gehören beispielsweise Treiber für die graphische Ausgabe, die Integration von Mauspuffern oder dem Weiterleiten von USB-Geräten.

Softwarekomponenten von Virtualbox sind in verschiedene Daemon-Prozesse aufgeteilt. Nachfolgende werden diese aufgelistet und beschrieben [25]:

2.13.2 Hypervisor

Ein *Hypervisor* oder auch VMM⁷⁵ ist die Softwarekomponente, die es ermöglicht, mehrere Gastbetriebssysteme (in VMs) gleichzeitig zu betreiben.

Bei *Hypervisor* existieren zwei verschiedene Architekturen: *Typ1* und *Typ2* [24].

⁷⁵Virtual Machine Monitor

Typ1 wird oft auch als 'bare metal'-*Hypervisor* bezeichnet, da dieser direkt auf der Hardware agiert. Auf ihm werden ohne weitere Zwischenschicht die virtuellen Maschinen betrieben (vgl. Anhang 14: Architektur eines Typ-1-Hypervisors auf S. 83). Beispiele für einen *Hypervisor* dieses Typs sind Microsoft Hyper-V⁷⁶ oder VMware ESX(i)⁷⁷.

Typ2 wird oft auch als 'hosted'-*Hypervisor* bezeichnet, da dieser auf einem Hostsystem (inklusive Betriebssystem) aufbaut. Es existiert also eine weitere Schicht zwischen *Hypervisor* und Hardware im Vergleich zu Typ 1 (vgl. Anhang 15: Architektur eines Typ-2-Hypervisors auf S. 83). Beispiele für einen *Hypervisor* dieses Typs sind (neben Oracle VirtualBox) Microsoft Virtual PC⁷⁸ und VMware Workstation⁷⁹.

Ressourcenverwaltung ist einer der wichtigsten Aspekte für virtualisierte Umgebungen. Verschiedene Ressourcen müssen unter virtuellen Maschinen aufgeteilt werden, sodass weder für die Teilnehmer ein Leistungseinbruch entsteht noch die Ressourcen ineffizient genutzt werden. Eine zentrale Rolle spielen die Belegung von Prozessor-, Arbeitsspeicher-, Swapspeicher- sowie Netzwerk-Kapazität. Um diese Ressourcen zu verwalten existieren verschiedene Ansätze.

Eine Methode ist alle Ressourcen in diskrete Teile aufzuteilen und selbige einzeln den Teilnehmern zuzuordnen. Diese Methode bietet sich beispielsweise bei Ressourcen mit endlicher Größe an (CPU, RAM). Während die Umsetzung einfach zu realisieren ist, birgt dieser Ansatz einen Nachteil. Auf diesem Weg werden einzelne Hardwareteile unter Umständen nicht effizient genutzt, da sie aktuell nicht vom besitzenden Teilnehmer in Verwendung sind, jedoch persistent zugewiesen wurden.

Aus diesem Grund wurde eine weitere Methode entwickelt, die eine software-regulierte Grenze setzt. Hier sorgen entweder Hypervisor oder Betriebssystem dafür dass der Teilnehmer nicht mehr als diese bestimmte zugewiesene Obergrenze der Ressourcennutzung erreicht. Verbleibende, aktuell nicht verwendete Ressourcen können von anderen Teilnehmern beansprucht werden. Dies im Rahmen ihrer eigenen Ressourcenobergrenze[25].

Die dritte verbreitete Methode ist der (auch in Betriebssystemen eingesetzte) *Fair Share Schedule (FSS)*. Solange ausreichend Kapazitäten zur Verfügung stehen, erhält jeder

⁷⁶<https://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx>

⁷⁷<https://www.vmware.com/products/esxi-and-esx/overview>

⁷⁸<https://www.microsoft.com/en-us/download/details.aspx?id=3702&751be11f-ed8-5a0c-058c-2ee190a24fa6=True>


⁷⁹<https://www.vmware.com/products/workstation>

Teilnehmer so viel Ressourcen, wie er anfordert. Ist die tatsächliche Gesamtgrenze der Ressource erschöpft, erhält jeder Teilnehmer ~~nur noch~~ den Teil, den er als Minimum zugesichert bekam. Diese Methode erreicht die effizienteste Ressourcenausnutzung [25].

Zu beachten ist, dass fehlende Arbeitsspeicherressourcen bei einem Teilnehmer sich negativer auswirken ~~kann~~, als fehlende Prozessorressourcen. Während 10% weniger Prozessorkapazität einen Performanceverlust von ebenso 10% bewirkt, kann 10% weniger Arbeitsspeicher als erforderlich zu erweitertem Paging-Aufwand führen. Dies kann unter Umständen auch andere Teilnehmer beeinflussen. Aus diesem Grund muss auch der Speicher ausreichend restriktiv partitioniert beziehungsweise zugewiesen werden.

Sicherheit und Isolation wird bei VirtualBox in erster Linie dadurch umgesetzt, dass es, wie alle anderen Anwendungen für Endanwender auf der x86-Architektur, sich auf Ring 3 konzentrieren. Dieser Ring repräsentiert die geringste Privilegierungsstufe auf dem x86-Prozessor. Der Kernel beispielsweise wird in Ring 0 betrieben. Dazwischen existieren zwei weitere Privilegierungsstufen.

Für jeden Teilnehmer wird ein einziger Prozess eröffnet. Jeglicher Code von virtuellen Teilnehmern wird in diesem Rahmen in Ring 3 ausgeführt. So erreicht jede Applikation in einer virtuellen Maschine nahezu die gleiche Performance als würde sie nativ auf dem Host betrieben.

Anders wird der Kernelcode der virtuellen Maschine behandelt. Sollte keine Hardwarevirtualisierung zur Verfügung stehen, wird dieser nicht in Ring 1 (wie vorgesehen), sondern lediglich in Ring  ausgeführt. Dies kann insofern problematisch sein, da Befehle existieren, die ausschließlich in Ring 0 funktionieren oder sich auf Ring 1 abweichend verhalten. Aus diesem Grund überprüft VirtualBox dauerhaft Ring 1 nach Befehlen dieser Art. Wird ein Befehl gefunden, der von dieser Problematik betroffen ist, setzt VirtualBox diesen direkt in einen geeigneten Hypervisorbefehl um.

In manchen Fällen ist VirtualBox nicht dazu in der Lage den entsprechenden Code korrekt zu interpretieren. Deshalb ist es unter Umständen notwendig gewisse Befehle in einer emulierten Umgebung auszuführen. Hierzu wird (unter Performanceeinbrüchen) Gebrauch von *QEMU* gemacht.

Verfügt das Hostsystem jedoch über Hardwarevirtualisierung auf Prozessebene (z.B. intel VT-x⁸⁰), kann entsprechender Kernelcode bis auf Ring 0 reichen und dort ord-

⁸⁰<http://ark.intel.com/Products/VirtualizationTechnology>

nungsgemäß ausgeführt werden. Auf diesem Weg ist zusätzlich ein Performancezuwachs beziehungsweise kein Performanceverlust zu verzeichnen [25].

2.13.3 Motivation

VirtualBox ermöglicht es auf einem Hostsystem mehrere virtuelle Maschinen nebeneinander zu betreiben. Unter Berücksichtigung der Tatsache, dass viele Applikationen nicht immer ihre zur Verfügung stehenden Ressourcen (CPU, RAM, HDD, etc.) benötigen, können diese dynamisch verwaltet werden. So ergibt sich eine effiziente Ressourcennutzung für alle Komponenten, die ursprünglich dedizierte Hardware erforderten und für sich alleine beanspruchten.

Darüber hinaus sind virtuelle Maschinen sehr portabel und können auf anderen Hostsystemen als ihrer ursprünglichen gestartet und betrieben werden. Ebenso ist es möglich (virtuelle) Hardwarekonfigurationen dynamisch anzupassen. Aus diesen Gründen eignen sich virtuelle Maschinen sehr gut zu Test- und Entwicklungszwecken. Auch das Betreiben von Legacy-Betriebssystemen ist in einer virtuellen Umgebung möglich. Systeme können flexibel für unterschiedliche Situationen angepasst (oder auch isoliert) werden, ohne dabei großen Aufwand zu generieren. Ebenso flexibel können fertig entwickelte und getestete Systeme reproduziert (provisioniert) werden.

Da die Verfasser dieser Arbeit über Ressourcen auf ihren Arbeits-PCs verfügen und *VirtualBox* kosteneffizient und vergleichbar ein Serversystem betreiben kann, wird selbiges eingesetzt.

2.14 Projektumsetzung

2.14.1 Anforderungen

Zu Beginn, und teilweise auch während des Projektes, wurden Anforderungen ~~an das Projekt~~ gestellt, die umgesetzt werden sollen und Grenzen gesetzt, die dem Projekt einen Rahmen vorgeben. Außerdem wurden Meilensteine definiert, um der Projektumsetzung mehr Struktur zu geben.

In Kapitel 1.1: Idee zur Arbeit auf S. 2 wurde bereits angesprochen, dass hardwareseitig ein Raspberry Pi mit einem Kartenleser als UE dienen soll und der Netzprovider ~~simuliert wird~~ auf einer virtuellen Maschine. Die Kommunikation der beiden Geräte wird über

eine PPPoE-Verbindung realisiert. Was nicht simuliert wird, ist der MSC, da dafür eine weitere virtuelle Maschine aufgesetzt werden muss und die Funktionalität des Knotens darauf reduziert wäre. RES und XRES (vgl. Kapitel 2.3) miteinander zu vergleichen.

Der Vergleich der beiden Werte RES und XRES wird allerdings auch nicht an anderer Stelle umgesetzt, sondern fällt weg, ebenso wie bei der ~~Resynchronisation~~ der Vergleich von MAC-S mit XMAC-S. Beide Vergleiche werden gemacht um sicher zu stellen, dass der jeweils andere Kommunikationspartner nicht korrupt ist. Da in diesem Projekt eine PPPoE-Verbindung genutzt wird, ist sichergestellt, dass beide Partner nicht korrupt sind und eine Überprüfung deshalb überflüssig.

Da das Kernziel eine erfolgreiche Authentifizierung ist, werden die nötigen Funktionen für sowohl die Synchronisation, als auch die Resynchronisation umgesetzt. Die Datenbank, in der der aktuelle SQN, beziehungsweise SEQ, gespeichert werden fällt jedoch weg, da diese bei nur einer USIM wieder zusätzlicher Aufwand bedeutet.

Da es außerdem nur um die Authentifizierung geht, aber nicht die Kommunikation zwischen beiden Seiten, wird vom simulierten AuC kein CK oder IK generiert.

Damit bleibt, dass bis auf f_3 und f_4 , alle Funktionen, die in Kapitel 2.4: Milenage Algorithmus auf S. 14 genannt wurden, korrekt umgesetzt werden sollen.

Zur korrekten Umsetzung der Milenage-Funktionen gehört ebenfalls eine Eigenimplementierung des AES-Algorithmus, aber nur für eine Blocklänge von 128 Bits, da der Milenage-Algorithmus keine anderen Blocklängen benötigt.

Bei beiden Algorithmen geht es zudem nicht um Geschwindigkeit oder Ressourcenverbrauch. Das soll heißen, dass bei der Implementierung dieser nicht daraufhin optimiert wird, dass wenig Speicherplatz benötigt wird oder das Programm besonders schnell ist. Stattdessen ist lediglich wichtig, dass beide Algorithmen korrekt implementiert werden und am Ende die Werte generieren, die für die Eingabeparameter zu erwarten sind.

Meilensteine Die Meilensteine, die für das Projekt definiert wurden, lauten wie folgt:

27.10.2015 Die Grundlagen für die Authentifizierungsalgorithmen sollen bekannt und verstanden sein

07.12.2015 Das Auslesen einer USIM soll funktionieren, alternativ soll eine USIM emuliert werden können

01.02.2016 Der virtuelle Netzbetreiber soll gemäß der Spezifikationen ~~funktionieren~~, für die nötigen Funktionen

29.02.2016 Die USIM soll in den Authentifizierungsprozess eingebunden werden, also mit dem virtuellen Netzbetreiber kommunizieren

17.04.2016 Die Studienarbeit ist bis zu diesem Zeitpunkt fertig geschrieben

30.05.2016 Die Studienarbeit muss abgegeben werden

Durch die Meilensteine ist eine klare Struktur gegeben und es gibt einen Zeitpuffer am Ende, um eventuell auftretende Komplikationen und Verzögerungen ausgleichen zu können ohne den Erfolg des Projektes zu gefährden.

2.14.2 Unterstützende Tools

Im Projekt werden einige Tools genutzt, die die Arbeit unterstützen und die Kommunikation unter den beiden Autoren vereinfacht. Trello und Git(hub) sollen hier kurz vorgestellt werden, da sie eine große Hilfe waren.

Trello Die web-basierte Projektmanagementsoftware Trello wird vom US-amerikanischen Unternehmen “Fog Creek Software” betrieben. Gegründet wurde es am 13. September 2011 und erfreut sich in letzter Zeit immer ~~mehr~~ Beliebtheit, sodass es seit Mitte 2015 auch auf Deutsch existiert.

Wenn man bei Trello angemeldet ist, kann man mehreren sogenannten Boards zugeordnet sein. In diesen Boards gibt es dann Listen. So kann mit Trello schnell ein Scrum- oder Kanban-Board erstellt werden. Man ist darauf jedoch nicht festgelegt, weshalb die Software ziemlich nutzungsoffen ist. In den Listen lassen sich dann Karten anlegen, welche sowohl Text, als auch Bilder und Checklisten enthalten können.

Git Um den Programmcode und das Schreiben dieser Arbeit zu vereinfachen, wurde Git genutzt. Dies ist eine quelloffene verteilte Versionsverwaltung, die vom Linux Gründer Linus Torvalds entwickelt wurde. Der Grund dafür war, dass die Entwickler des Linuxkernels lange BitKeeper genutzt haben für die Versionsverwaltung, aber dies konnte durch eine Lizenzänderung nicht mehr genutzt werden, weshalb Torvalds sich entschied seine eigene Verwaltung zu bauen, die seine Anforderungen erfüllt.

Im Gegensatz zu beispielsweise Subversion, gibt es nicht einen zentralen Server, auf dem die komplette Historie gespeichert ist, sondern jeder Client ist auch ein Server und hält die Historie bereit. Dies ermöglicht dem Nutzer einige Interaktionen durchzuführen ohne an das Internet angebunden zu sein.

Git hat bewiesen, dass es stabil ist, da viele größere Projekte wie Android oder Eclipse damit ihre Projekte entwickeln.

3 Tätigkeit

3.1 Stellen des Netzproviders

3.1.1 Ubuntu


Wie in 2.11: Ubuntu auf S. 47 beschrieben wird Version 14.04 LTS der Linuxdistribution Ubuntu eingesetzt. Das bei der Herstellerwebsite⁸¹ für Serverbetrieb erhältliche Image wird in einer virtuellen Maschine betrieben. Diese weist folgende Eigenschaften auf:

Gerät	Typ
CPU	1x 64 Bit
RAM	1x 1024 MB
HDD	1x 10 GB
NIC	2x Bridged (1x auf enp0s25; 1x auf wlp3s0)

Tabelle 4: Eigenschaften der virtuellen Maschine

Aufgrund der Tatsache dass nur ein einziger Client über den Zugang von *PPPoE* zu erwarten ist, sind die Anforderungen an CPU, Arbeits- und Festplattenspeicher recht gering. Wichtig ist das Vorhandensein zweier Netzwerkschnittstellen, die auf jeweils eine physikalisch Schnittstelle geleitet werden. Dies ist notwendig, da sowohl die UE über eine Ethernetverbindung mit der virtuellen Maschine angebunden sein wird, als auch eine Dauerhafte Internetverbindung mit der verbleibenden Schnittstelle aufrecht erhalten werden muss. So ist es möglich nach erfolgreicher Authentifizierung des UE den Zugang zum Internet freizuschalten.

3.1.2 PPPoE

 Einrichtung von *PPPoE* basierend auf der vollständigen Ubuntuinstallation realisiert werden. Hierzu wird die aktuelle Version 3.11 auf der Herstellerwebsite⁸² bezogen. Zusätzlich wird das Ubuntu-Paket zur Unterstützung des Point-To-Point-Protocol (als *ppp* erhältlich) benötigt. Sind diese beiden Grundvoraussetzungen abgedeckt wird der Roaring Penguin PPPoE-Server ohne spezielle Änderungen aus dem Quellcode kompiliert.

⁸¹<http://releases.ubuntu.com/14.04/>

⁸²<https://www.roaringpenguin.com/products/pppoe>

In der Datei `/etc/ppp/pppoe-server-options` werden die notwendigen Parameter gesetzt. Anzugeben sind die DNS-Server sowie die CHAP-Authentifizierung. Der User für die Authentifizierung wird unter `/etc/ppp/chap-secrets` definiert.

Der zu verwendende Adresspool an IP-Adressen ist ebenfalls (unter `/etc/ppp/allip`) anzugeben.

Abschließend fehlt noch die Deklaration der zu verwendenden Netzwerkschnittstelle (unter `/etc/network/interfaces`), die zum Betrieb mit PPPoE eingesetzt werden soll. Ihr muss eine valide IP-Adresse aus dem definierten Pool sowie eine Subnetzmaske zugewiesen werden.

Alle Konfigurationsdateien befinden sich im Anhang.

Damit die Internetverbindung nach erfolgreicher Authentifizierung des UE auch korrekt von Provider freigeschaltet und weitergeleitet wird, muss eine IPTables-Regel folgender Form eingerichtet werden:

```
$ iptables -t nat -A POSTROUTING -s 192.168.178.0/24 -o enp0s25 -j MASQUERADE
```

Sie nimmt den Verkehr auf dem Device `enp0s25` an und maskiert (vgl. NAT⁸³) diesen für den weiteren Betrieb (wie es bei einem realen Provider auch wäre). Dies geschieht für eingehenden sowie ausgehenden Verkehr. Die auf den zweiten Netzwerkadapter geleitete Netzwerkschnittstelle bleibt mit Defaultwerten konfiguriert und fungiert als Zugangspunkt zum Internet.

Der Server kann auf der gewünschten Netzwerkschnittstelle mit einer eigenen IP-Adresse gestartet und genutzt werden:

```
pppoe-server -C isp -L 192.168.178.254 -p /etc/ppp/allip -I enp0s25
```

Ein Beispiel-Initskript, welches diesen Zweck erfüllt, befindet sich im Anhang unter 6.2: Listings auf S. 103.

Korrekt konfigurierte Clients, die über die Zugangsdaten verfügen, können sich über diese Verbindung anmelden und Internetzugriff erhalten.

⁸³Network Address Translation

3.1.3 Implementierung AES

Die Blockchiffre AES wurde in der Datei *rijndael.c* geschrieben. Der Code wurde in drei logische Teile unterteilt, nämlich die Variablendeklaration, die Main-Funktion, die die einzelnen Runden steuert und Funktionen, die die ~~konkrete Implementierung der~~ jeweiligen Transformation bereitstellen. Des weiteren wurde ein eigener Variablentyp *u8* definiert, der eine verkürzte Form des unsigned integer ist.

Der gesamte Code steht im Anhang **???** und wird in den folgenden Paragraphen näher erläutert.

Variablen Die zu Beginn initialisierten Variablen sind ~~einige~~ Zählervariablen, die für spätere Schleifen reserviert werden. Zusätzlich werden jedoch auch einige Konstanten definiert. Da keine Optimierung bezüglich des Speicherplatzverbrauchs getätigt werden muss, wird die Substitutions-Tabelle, also die S-box, nicht zur Laufzeit erst berechnet, sondern zu Beginn definiert. Ähnlich sind die ~~Runden~~konstanten für die Generierung des Rundenschlüssels schon vorher in einer Konstante definiert. Des weiteren existiert die *mixMatrix*, **genauer gesagt** ein 4 x 4-Array, welches die MixColumns-Transformation vereinfachen wird.

tmp ist ein Hilfs-Array, in *key* wird der Chiffrierschlüssel gespeichert, in *roundKey* die **Runden**schlüssel für alle zehn Runden, ~~also~~ inklusive der 1. Runde, und in *state* wird der Zustand hinterlegt.

encrypt Die Funktion, die vom Milenage-Algorithmus aufgerufen wird, ist *encrypt* und braucht drei Eingabeparameter. *input* ist der Textblock, *keyStr* der Schlüssel und output die Variable, in die der chiffrierte Status am Ende geschrieben werden soll.

Da Textblock und Schlüssel ~~zu erst~~ noch in ein Array konvertiert werden müssen, wird bei beiden erst *convert2array* aufgerufen. Wie die Funktion erkennen lässt, konvertiert sie den Inhalt des ersten Eingabeparameter in ein Array und speichert das Ergebnis in der zweiten übergebenen Variable. Entgegen der Spezifikation wird ~~das~~ der Textblock und der Schlüssel aber nicht zeilenweise, sondern spaltenweise in das Array umgewandelt.

```
void convert2array(u8 input[16], u8 output[4][4]) {
    for (j = 0; j < 4; j++) {
        for (i = 0; i < 4; i++) {
            output[i][j] = input[(j * 4 + i)];
        }
    }
}
```



j ist die Spalte in dem Array und i die Zeile. Durch die einfache Rechnung $j * 4 + i$ kann das Array nacheinander durchgegangenen ~~werden~~ und spaltenweise abgespeichert werden.

Anschließend wird gemäß der Spezifikation der Rundenschlüssel generiert, allerdings wie schon erwähnt für alle zehn Runden und nicht für jede Runde einzeln. Wie im Anhang 16: Grafische Darstellung der AES-Verschlüsselung auf S. 84 gezeigt, wird dann der erste Rundenschlüssel mit dem Zustand verknüpft.

Im Anschluss folgen die ersten neun Runden durch eine einfache for-Schleife. Die folgenden beiden verschachtelten Schleifen sind schon die SubByte-Transformation, welche nicht in eine eigene Funktion ausgegliedert ist. *getSboxValue* dient nur zur besseren Lesbarkeit des Codes.

Nachfolgend werden auch die anderen Transformationen durchgeführt und nach den neun Runden die zehnte ohne eine MixColumns-Transformation.

Der finale Zustand wird dann wieder in eine Zeichenkette konvertiert und zurückgegeben.

generateRoundKey In der ersten doppelten for-Schleife von generateRoundKey wird der Chiffrierschlüssel in den ersten Block des roundKey übertragen.

Anschließend werden die restlichen Rundenschlüssel gemäß Spezifikation generiert. Um die spaltenweise Rotation durchzuführen, wird das erste Byte der Spalte, das von oben nach unten rotiert, in das temporäre Array geschrieben. Anschließend überschreibt der zweite Wert den ersten, dann der dritte Wert den zweiten und im vorletzten Schritt das vierte Byte in der Spalte. Da nun der Wert im ersten Byte nicht mehr der selbe wie vor der Rotation ist, wird das vierte Byte ~~nun~~ mit dem Wert aus dem temporären Speicher gefüllt.


Da die erste Spalte des Arrays anders berechnet wird als die folgenden, gibt es zwei Schleifenblöcke. Der erste berechnet die erste Spalte und der zweite Block, die restlichen drei, da diese nach dem gleichen Schema berechnet werden.


shiftRow Die Funktion shiftRow ist etwas komplexer, auch wenn sie nur wenige Zeilen lang ist. Die erste Schleife zählt die Spalten des Zustands (*state*) durch und die beiden folgenden Schleifen die Zeilen.

In der ersten inneren Schleife wird die aktuelle Zeile komplett in ein temporäres Array gespeichert und in der zweiten Schleife findet die Rotation statt.

```
for (i = 0; i < 4; i++) {  
    for (j = 0; j < 4; j++) {  
        tmp[j] = state[i][j];  
    }  
    for (j = 0; j < 4; j++) {  
        state[i][j] = tmp[(j + i) % 4];  
    }  
}
```

In der zweiten inneren for-Schleife wird dem Zustands-Array Spalte für Spalte und Zeile für Zeile ein neuer Wert zugewiesen, der aus dem temporären Array geliefert wird. Durch die Addition von j und i wird bei der ersten Zeile auch mit dem ersten Byte angefangen, bei der zweiten Zeile mit dem zweiten Byte und so weiter. Das Byte, mit dem angefangen wird, wird aber immer an ~~die erste~~ Stelle des Zustands-Array gespeichert. Damit wird die Rotation um null, eins, zwei und drei Bytes ermöglicht. Da aber zum Beispiel in der dritten Zeile ~~das~~ das erste Byte rechts wieder reingeschoben werden soll, ist die modulo 4-Operation nötig.

$i\%4$ reicht übrigens nicht aus, da dann zwar beispielsweise beim dritten Byte in der dritten Zeile begonnen werden würde, aber nicht die durch die anderen Bytes in der Zeile iteriert wird. 

mixColumn Im Code wurde mixColumns nicht mit der zugrunde liegenden Mathematik, also der Polynome mit Koeffizienten aus $GF(2^8)$ gelöst, sondern weniger effizient über einen switch-case Block. Die Umsetzung ist vermutlich weniger effizient, aber trotzdem erfolgreich und einfacher zu schreiben.  ~~Se~~ wird gemäß der Beispielimplementierung die MixColumns-Transformation durchgeführt für jedes Byte in jeder Spalte. Das Array *mixMatrix* gibt vor, ob die Funktion T_2 oder T_3 angewendet werden muss. In der Implementierung wurde jedoch nur eine t2-Funktion geschrieben und T_3 wird mit Hilfe der t2-Funktion umgesetzt, ähnlich der mathematischen Definition in Kapitel 2.4.2: Funktionsweise auf S. 15.

3.1.4 Implementierung Milenage

Die Implementierung des Milenage Algorithmus wurde nicht auf Basis der Funktionen strukturiert sondern der Ausgabeblöcke. Die Funktionen $f2$ und $f5$ sind deshalb in einer Funktion zusammengefasst, ebenso wie $f1$ und $f1^*$. Die beiden übrigen Funktionen $f3$ und $f4$ wurden hingegen gar nicht implementiert, da diese nicht für den reinen Authentifizierungsvorgang nötig sind und deshalb nicht gefordert sind.

In den folgenden Paragraphen werden wieder die Hauptfunktionen erklärt für ein besseres Verständnis des Codes.

Variablen Ähnlich wie im AES-Programmcode sind auch bei Milenage einige Variablen zu Beginn deklariert. Dazu gehören alle Werte, die SIM-Karten spezifisch sind und in der Realität in einer Datenbank gespeichert werden für jeden Netzteilnehmer. So sind also die Rundenkonstanten $c1$ bis $c5$, SQN, AMF und OPc schon fest definiert. Zusätzlich sind die Adressräume für die Werte, die generiert werden reserviert, wie AK oder RES und i als Zählervariable.

f1 Die Funktion benötigt als Eingabeparameter den Schlüssel K und den RAND. Der RAND wird in der Funktion genRand generiert auf Basis der aktuellen Zeit.

Für Demozwecke werden dann zu Beginn der Funktion die beiden Werte SQN und AMF ausgegeben, so wie in der restlichen Funktion auch andere Parameter in der Konsole ausgegeben werden.

Die beiden folgenden Schleifen sind zur Generierung des IN1, welcher ein konkatenierter Wert aus SQN und AMF ist. Um nicht in vier Schleifen abwechselnd durch beide Werte zu iterieren wird nur jeweils einmal durch beide Werte iteriert. Im Falle von SQN wird das erste Byte also direkt an den Anfang des IN1-Arrays und an achte Stelle geschrieben. Ähnlich funktioniert auch das hinzufügen von AMF zu IN1.


Die Variable *toEncrypt* ist nur eine temporäre Variable in der der OPc mit dem RAND verknüpft wird über XOR. Diese wird dann verschlüsselt mit der in Kapitel 3.1.3: Implementierung AES auf S. 59 erklärten Funktion encrypt und der verschlüsselte Wert ist dann der TEMP-Wert. Dieser wird nachher auch von den anderen Funktionen benötigt, aber da $f1$ die erste Funktion ist die aufgerufen wird, wurde die Generierung der TEMP-Funktion nicht ausgegliedert in eine eigene Funktion.



Anschließend fängt die Berechnung des ersten Blocks OUT1 an aus dem sich MAC-A und MAC-S ableiten. Nach der Verknüpfung von OPc und IN1 wird rotiert. Anders als

bei AES und der shiftRows-Transformation ist die Rotation nicht byteweise, sondern bitweise. Die Strings sind aber als Byte-Arrays gespeichert, weshalb sie für eine bitweise Rotation erst zu einem binären Array konvertiert werden. Diese Konvertierung geschieht über eine Bitmaske. In jedem Byte des *hexArr* werden alle acht Bit durchiteriert und je nach dem, ob **da ne 1** oder 0 steht wird diese in das *binArr* geschrieben. Die nachfolgende Rotation wird in einem der folgenden Paragraphen noch näher erläutert. Die Rückkonvertierung des binären Arrays in das Byte-Array erfolgt wieder über Bitoperationen. Das entsprechende Byte wird erst um **1** Stelle nach links verschoben und es wird rechts mit einer 0 aufgefüllt. Durch die Oder-Operation mit dem Bit-Array wird diese zu einer 1, wenn eine 1 im Bit-Array steht oder bleibt eine 0.

Die restliche f1-Funktion verläuft wie in der Spezifikation definiert. Der rotierte OUT1 wird mit der Konstante c1 und TEMP verknüpft, verschlüsselt und zum Schluss noch mit OPc verknüpft. Die beiden letzten Schleifen geben nur noch MAC-A und MAC-S aus.

f2_5 Die Funktion f2_5 braucht zwei Eingabeparameter, **nämlich** den Schlüssel und *response_arr* welches die Werte enthält, die nach der Registrierungsanfrage an die SIM-Karte zurückgeschickt werden.


Ansonsten ist die Funktion sehr ähnlich der schon beschriebenen f1-Funktion aufgebaut. loch wird IN1 nicht benötigt und TEMP wurde schon generiert, **weshalb diese Werte in f2_5 nicht nochmal generiert werden müssen. So wird also** gemäß der Spezifikation TEMP mit OPc verknüpft, dieser Wert rotiert und mit der Konstanten c2 verknüpft. Das Ergebnis ist OUT2, aus welchem sich AK und RES ableiten, welche in jeweils einer for-Schleife auch wieder in der Konsole ausgegeben werden. Da RES auch an die USIM geschickt wird **als Antwort**, wird diese in das *response_arr* geschrieben, **wofür die Standardfunktion sprintf genutzt wird.**

f5star Diese Funktion benötigt ebenfalls zwei Werte. s eine ist wieder der Schlüssel und **das** andere ist der SQN verknüpft mit AK, der von der SIM-Karte geschickt wird für die Resynchronisation. Die Funktion berechnet erst den AK aus *f5** anhand der Definition des 3GPP. Dafür wird wieder OPc mit TEMP verknüpft über die XOR-Operation, das ganze rotiert it c5 verknüpft und verschlüsselt. Zum Schluss noch einmal **wieder** mit OPc verknüpft und damit ist OUT5 **fertig** berechnet.

Die erste Hälfte, die AK ergibt wird nun benötigt, um die SQN aus der Variable *sqn_ak*

zu ermitteln. Diese SQN ist allerdings keine, die für die Berechnung der neuen Authentifizierungsvektoren genutzt werden kann, da sie schon benutzt wurde und der Netzprovider deshalb eine neue SQN generieren muss. Bei der USIM in diesem Projekt wird eine komplett zählerbasierte SQN verwendet mit einer Arraygröße von 32. Wie in Kapitel 2.4.2: Funktionsweise auf S. 15 näher erläutert, bedeutet das, dass die hinteren fünf Bits der IND sind und die vorderen 43 die SEQ. Da die SQN ein Bytearray ist, wird aus dem letzten Byte mit einer Bitmaske die IND extrahiert und die vorderen drei Bits des letzten Bytes als SEQ abgespeichert. Bevor die SEQ hochgezählt wird, wird der IND erhöht. Da das SQN-Array nur 32 Werte speichert, wird IND nicht beliebig groß und fängt durch modulo 32 wieder bei 0 an, wenn die 31 überschritten wird. Das letzte Byte wird nun auf 0 gesetzt und die letzten fünf Bits gleich dem neuen Wert von IND gesetzt.

Im zweiten Schritt wird auch SEQ um eins erhöht, aber da SEQ beim sechsten Bit anfängt, wird SEQ nicht mit 1 sondern mit 32_{10} (00100000_2) addiert. Wenn jedoch die vorderen drei Bits des letzten Bytes 111_2 sind und jetzt mit 1 addiert wird, muss ein Übertrag in das vorletzte Byte vorgenommen werden. Diesen Überlauf überprüft die if-Abfrage und addiert ~~im Falle~~ eine 1 auf das vorletzte Byte. Unabhängig vom Überlauf wird der Wert von *seq* auch wieder in das letzte Byte durch die oder-Operation übertragen.

rotWord Diese Funktion nutzt einen effizienten Algorithmus, um ein Array oder eine Zeichenkette zu rotieren. Die Idee dahinter ist, dass, wenn wie in Abbildung 9 eine Zeichenkette 10 Zeichen hat und um 3 Zeichen nach links rotiert werden soll, ~~dann werden~~ die ersten drei Zeichen vertiert, anschließend die letzten sieben Zeichen und in einem dritten Schritt der gesamte String.

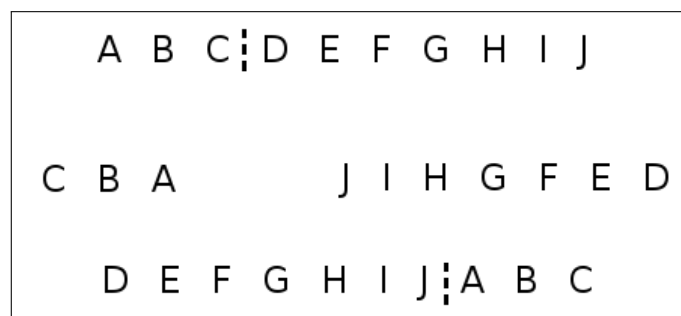


Abbildung 9: Schematische Darstellung des Rotations-Algorithmus

In der konkreten Implementierung wird zum ~~invertieren~~ die Funktion *reverse* aufgerufen,

die als Übergabeparameter ein Array übergeben bekommt und die Größe des Arrays, dass invertiert werden muss. Das vordere und hintere Element werden dann mit Hilfe der *tmp*-Variable ausgetauscht, bis die Schleife in der Mitte des Arrays angekommen ist und das Array damit invertiert wurde.

3.2 Erstellen des UE

3.2.1 Aufsetzen des Raspberry Pis

Raspbian wird über die von der Webpräsenz⁸⁴ des Projekts gehosteten vorgefertigten Builds aufgesetzt.

Die SD-Karte muss nicht weiter formatiert und für die Installation vorbereitet werden. Das bereitstehende ISO-Image ist ein Abbild eines bereits korrekt formatierten sowie installierten Raspbian (Jessie). Nach (blockweisem) Kopieren des Images auf die SD-Karte kann von dieser direkt gebootet werden.

Nach dem initialen Bootvorgang können Userspezifische Änderungen durchgeführt werden. Hierzu gehört das Anpassen von Netzwerk- und Hostinformationen. Um später eine Administration via ssh⁸⁵-Remotezugang zu ermöglichen wird auch dieser in der entsprechenden Konfigurationsdatei zum automatischen Starten aktiviert.

Für den weiteren Betrieb müssen Pakete aus dem offiziellen Repository nachinstalliert werden. Spezielle Pakete, die dort nicht verfügbar sind, werden manuell kompiliert.

SIM-Kartenleser Der benötigte (herstellerspezifische) Cherry-Treiber für den Kartenleser TC1100 ist nicht im offiziellen Repository enthalten, kann jedoch manuell kompiliert werden. Auf der Webpräsenz des Herstellers⁸⁶ wird ein Treiber für den Betrieb mit 64-Bit Linuxdistributionen zu Verfügung gestellt. Selbiger kann nach Installation der nativen *pcsc*-Pakete (aus dem Repository) kompiliert und eingesetzt werden.

Sind diese Softwarepakete eingerichtet, kann der Leser via USB verbunden, die SIM-Karte eingesetzt und erste Kommunikationsversuche durchgeführt werden. Hier bietet sich der vom Treiber mitgelieferte Scanmechanismus für neue Devices an. Da der Treiber Hot-

⁸⁴<https://www.raspberrypi.org/downloads/raspbian/>

⁸⁵Secure Shell

⁸⁶<http://www.cherry.de/cid/download.php>

plugfähig ist, kann direkt mit der SIM-Karte gearbeitet werden, sobald diese eingesetzt wird.

pysim kann von Kevin Prince's Githubrepository⁸⁷ bezogen werden. Es benötigt ebenso wie andere nachfolgend genannte Python-Skripte noch das Paket *pysim*, welches in den offiziellen Raspbian Paketquellen zur Verfügung steht. Nach erfolgreicher Installation von Pysim und `pullen` des Skriptes wird selbiges über Python verwendet.

osmo-sim-auth ist vom Gitrepository⁸⁸ des Projekts Osmocom zu beziehen. Genau wie auch pysim kann dieses nach dem `pullen` über Python verwendet werden. Entweder als Bibliothek oder direkt im Interpreter.

3.3 Integration mit PPPoE

Um den Verbindungsaufbau zwischen UE und AuC korrekt abzuwickeln, muss sowohl der Client als auch der Server für PPPoE konfiguriert sowie gestartet sein.

3.3.1 Server

Hier muss initial der Daemon für den PPPoE-Server gestartet sein. Anzugeben ist das gewünschte Netz mit zugehöriger Netzwerkschnittstelle. Ebenso muss die IPTables-Regel zur Weiterleitung des Internetverkehrs gesetzt sein.

3.3.2 Client

Hier müssen einmalig über das Tool *pppoe-conf* die Parameter und Authentifizierungsdaten für die PPPoE-Verbindung eingepflegt werden.

Das Tool bietet die Möglichkeit, selbige Informationen zu speichern und automatisch den Verbindungsaufbau mit dem Server aufzunehmen. Bei Neustart des Systems oder Verbindungsabbruch (regulär oder durch technische Zwischenfälle) greift dieser Mechanismus ebenfalls.

⁸⁷<https://github.com/kevinprince/pysim>

⁸⁸<http://cgkit.osmocom.org/cgit/osmo-sim-auth>

3.4 Implementierung der Server-Client Architektur

3.4.1 Implementierung des Servers

Das Hauptprogramm, das am Ende in `main.c` gestartet wird, ist der eigentliche Netzprovider beziehungsweise die Authentifizierungsstelle. Damit diese Daten dynamisch empfangen kann, wird ein Serversocket aufgebaut mit dem sich Clients, also die USIM, verbinden und kommunizieren können. Um diesen Socket aufzubauen werden die beiden Bibliotheken `sys/socket` und `arpa/inet` benötigt. Zusätzlich werden zu Beginn wieder einige Variablen deklariert, die benötigt werden, um eine gute Kommunikation mit dem Server mit der SIM-Karte zu ermöglichen.

Die Konstanten mit Präfix `ANSI` wurde definiert, damit die Konsolenausgaben öffentlich sind, aber die Lesbarkeit des Codes darunter nicht leidet.

Kommunikationsaufbau Beim Start des Programms muss dem Server ein Socket zur Verfügung gestellt werden, was mit der Funktion `socket` erfolgreich geschieht, sofern sie keinen Fehlercode zurückgibt, der in der `if`-Abfrage abgefangen wird. Anschließend muss der Serversocket konfiguriert werden. Er wird mit der Funktion `bind` so eingestellt, dass er auf jede IP-Adresse mit Port 12345 hört. Nachdem dies konfiguriert ist, wartet das Programm auf eine Verbindungsanfrage durch die Funktion `listen`. Neben der Socket-Beschreibung muss der Funktion `listen` mitgeteilt werden, wie viele Verbindungen der Server maximal zulässt. Da in diesem Projekt nur mit einer SIM-Karte gearbeitet wird, ist der Wert also auf 1 gesetzt. Nach diesen Schritten akzeptiert der Server einen Client und speichert die Daten des Clients in `client_sock`. Wenn ein Client nun eine Verbindung aufbaut, wird dies vom Server registriert und die Infos des Clients abgespeichert, damit der Server weiß mit wem er kommuniziert.

In der `while`-Schleife wird nun darauf gewartet, dass die USIM eine Nachricht schickt, die mit der Funktion `recv` empfangen wird. Die Funktion `recv` benötigt den Client von dem eine Nachricht erwartet wird, sowie ein Array in den die Nachricht dann geschrieben werden soll. Außerdem wird angegeben, wie lang die Nachricht sein wird. Diese Zahl darf maximal so groß sein, wie die Größe des Arrays, in dem die Nachricht gespeichert wird. Zusätzlich können noch Flags gesetzt werden, was in diesem Fall aber nicht nötig ist. Wenn die Funktion `recv` nun einen Wert zurückgibt kann dieser -1, 0 oder größer 0 sein, wobei letzteres die Länge der empfangenen Nachricht bedeutet. Die anderen beiden Fälle werden am Ende dieses Kapitels erklärt.

Verarbeiten der Nachrichten Wenn eine Nachricht empfangen wurde, ist sie im übergebenen Array *client_message* eingetragen. Dieses Array ist eine Zeichenkette und kein Array aus unsigned chars. Außerdem sind in dieser Nachricht einige Werte konkateniert gespeichert, weshalb diese nun getrennt werden. Zu erst wird das Statusbyte in eine temporäre Variable gespeichert mit Hilfe der Standardfunktion *strcpy* und anschließend wird der Wert umgewandelt in eine Dezimalzahl mit *strtol*. Die Konsolenausgabe ist abhängig davon, ob eine erste Anfrage oder eine Resynchronisation nötig ist. Als nächstes wird die IMSI extrahiert, welche unbearbeitet in *response_message* später wieder eingetragen wird und der Status in *response_message* wird standardmäßig auf 50 gesetzt. 50 ist dabei die hexadezimale Darstellung des Zeichens "2".

Wenn die Nachricht, die empfangen wurde, den Status 3 hat, dann bedeutet das, dass Resynchronisiert werden muss. In dem Fall wurde von der USIM der AUTS geschickt und dieser wird in entsprechende Variable übertragen. Nach der Extraktion wird mit AUTS die Funktion *f5star* aufgerufen, um den Resynchronisations AK zu berechnen sowie die neue und richtige SQN zu kalkulieren. Des weiteren wird der Status der Antwortnachricht auf 52, also "3" gesetzt.

Im letzten Schritt wird RAND generiert, *f1*, *f2* und *f5* berechnet, sowie aus den generierten Werten der AUTN gebildet. Die Werte werden alle in der Antwortvariable eingetragen, damit dieser dann an die USIM geschickt werden kann. Das passiert über die Funktion *write*, die wie *recv* die Daten des Clients braucht, die eigentliche Antwortnachricht und die Länge dieser Nachricht. Für die bessere Visualisierung wird die Antwort am Ende auch in der Konsole ausgegeben.

Beenden des Servers Wie schon erwähnt kann *recv*, auch die Werte 0 und -1 zurückgeben. Im Falle von 0 bedeutet das, dass der Client die Verbindung unterbrochen hat und deshalb keine Kommunikation mehr möglich mit diesem. Bei -1 hingegen bedeutet es, dass die Nachricht des Clients nicht gelesen werden konnte.

In beiden Fällen wird das Programm und damit der Server beendet.

3.4.2 Implementierung des Clients

Wie bereits erläutert, existiert neben dem Server (vgl. 3.4.1: Implementierung des Servers auf S. 67) auch ein Client, damit die Kommunikation wie erwünscht realisiert werden kann. Hier handelt es sich um die Komponente, die Zugriff auf die SIM-Karte hat (*UE*)

und alle notwendigen Informationen an den Server sendet. Alle notwendigen Funktionen sind entweder auf der SIM-Karte persistent gespeichert oder werden durch das Triggern der integrierten Funktionen generiert.

Der Client wird in Python entwickelt und verwendet grundlegend die Bibliothek *osmo-sim-auth* (vgl. 2.10.2: *osmo-sim-auth* auf S. 46). Darüber hinaus kommen weitere Bibliotheken zum Einsatz, die in Python den Kommunikationsaufbau mit Chipkarten ermöglicht (card-Utills, card-Sim und card-USIM).

Aufruf der Authentifizierung Diese Funktion wird im Clientprogramm im Abschnitt *handle_usim* definiert. Es werden externe Informationen (falls vorhanden) für den Aufruf gesammelt und entsprechend an die SIM-Karte gesendet. In jedem Fall wird ein Authentifizierungsvorgang für eine USIM-Karte initiiert.

Abhängig von gegebenen Information ist die SIM-Karte daraufhin in der Lage Ergebnisvektoren zu berechnen und zu liefern. Hier gibt es drei verschiedene Möglichkeiten:

Die Form der Anfrage war inkorrekt. D.h. sowohl der Wert für AUTN, als auch der Wert für RAND entsprechen den standardisierten Vorgaben. Es ist also notwendig die Anfrage mit korrekten Werten erneut durchzuführen.

Die Form der Anfrage war korrekt, jedoch ist eine Resynchronisierung der übermittelten SQN erforderlich. In diesem Fall muss der Aufruf des Authentifizierungsvorgangs nach erfolgreicher Neuberechnung der SQN erneut durchgeführt werden.

Die Form der Anfrage war korrekt, sowie die Beschaffenheit der übermittelten SQN. Ist dies der Fall, kann die SIM-Karte ohne weiteres die Vektoren CK, IK und RES generieren. Diese werden entsprechend ausgegeben.

Der Aufruf des Authentifizierungsvorgangs kann im Abschnitt *main* über *authenticate* eingeleitet werden.

Kommunikationsaufbau Zu Beginn der Laufzeit des Client, wird wie bereits bei der Serverkomponente ein Socket definiert. Über die Angabe des gewünschten Hostnamens und des Ports kann die Verbindung mit dem Socket auf dem Server aufgebaut werden.

Nach dem Aufbau der Verbindung beginnt die Realisierung des Informationsaustauschs zwischen Client und Server. Alle weiteren Operationen werden in der nachfolgenden while-Schleife abgearbeitet.

Verarbeiten der Nachrichten Die vorliegende while-Schleife wird solange ausgeführt, bis entweder der Client oder der Server die Verbindung abbrechen. Dies gewährleistet die Möglichkeit der erneuten Authentifizierung.

In der Schleife werden Nachrichten auf Basis der Informationen, die von der SIM-Karte geliefert werden, an den Server gesendet. Ebenso werden von selbigem Informationen empfangen. Je nach aktuellem Zustand beziehungsweise Position im Authentifizierungsvorgang, werden unterschiedliche Operationen ausgeführt. Welche Operationen ausgeführt werden müssen, wird anhand des Statuscodes erkannt. Dieser kann jeweils aus den Nachrichten des Servers extrahiert werden. Auf entsprechende Werte wird immer über die Länge beziehungsweise Start- und Endstelle zugegriffen. So extrahiert der Client beispielsweise den Statuscode aus den Stellen 0-1 der Antwortnachricht.

Beenden der Verbindung Die Verbindung wird dann beendet, wenn entweder Client oder Server dies initiieren.

3.4.3 Kommunikationsablauf zwischen Client und Server

Dieser Ablauf wird über die durch Sockets initiierte Verbindung realisiert. Aufgrund der Tatsache, dass der Vorgang je nach Ausgangszustand den Aufruf verschiedener Operationen verlangt, wird ein Statuscode in jede Nachricht integriert. Anhand dieses Statuscodes können dann beide Kommunikationsteilnehmer jeweils erforderliche Operationen ausführen.

Der Client bleibt so lange in der Authentisierungsschleife, bis diese erfolgreich abgeschlossen werden. Gleiches gilt für den Server. Beide reagieren je nach Statuscode (extrahiert aus den ersten Stellen) mit nachfolgend erläuterten Aktionen.

Statuscode 0: Bisher wurde nur die Verbindung zwischen beiden Teilnehmern erfolgreich aufgebaut, aber keine Informationen ausgetauscht. Der Client liest die IMSI der SIM-Karte aus und schickt sie mit vorangestelltem *Statuscode 1* an den Server.

Statuscode 1: Der Server kann aufgrund des Statuscodes die Beschaffenheit der Clientnachricht assoziieren und daraufhin die IMSI extrahieren. Der Client führt nach Empfang der Serverantwort, die einen Wert für sowohl AUTN als auch RAND enthält, den ersten Authentifizierungsversuch durch. Glückt dieser, generiert die SIM-Karte bereits

zu diesem Zeitpunkt schon Werte für die Vektoren C_k , I_k und RES und der Authentifizierungsvorgang ist erfolgreich beendet. Sollte dies nicht der Fall sein, antwortet die SIM-Karte mit einem Wert für $AUTS$, da die Resynchronisation der SQN noch aussteht. Der Statuscode des Clients wird auf 2 gesetzt und eine weitere Iteration des Vorgangs aufgerufen.

Statuscode 2: In dieser Iteration sendet der Client neben der $IMSI$ auch die $AUTS$ an den Server. Zusätzlich wird dem Server der Statuscode 3 übermittelt.

Statuscode 3: Erhält der Server diesen Statuscode, führt er die Resynchronisation der SQN aus. Dies erfolgt anhand der vom Client erhaltenen $AUTS$. Hat der Server dies abgeschlossen, übergibt er den neu errechneten Wert an den Client zurück. Dieser nimmt den Wert an und führt die Authentifizierung erneut aus. Glückt dieser Vorgang, antwortet die SIM-Karte schließlich mit den gewünschten Vektoren CK , IK und RES . Die Authentifizierung ist hier erfolgreich abgeschlossen.

4 Ergebnis

Die beiden nachfolgenden Kapitel beschäftigen sich mit der Offenlegung durchgeführter Tests und deren Ergebnisse. Hierbei wird näher darauf eingegangen, für welche Teilspekte welche Prüfmechanismen eingesetzt wurden. Ebenso wird verdeutlicht, welche Funktionen erfolgreich, beziehungsweise mit zwischenzeitlichen Komplikationen implementiert werden konnten.

4.1 Funktionsprüfung

Nachfolgende Tabelle liefert zunächst eine Übersicht zu implementierten und im Anschluss getesteten Funktionen:

Funktion	Komplikationen bei Integration bzw. Implementierung	Abschließender Erfolg
Bereitstellung des Raspberry Pis (UE)	●	✓
Bereitstellung der virtuellen Maschine (Provider)	○	✓
Entwicklung AES	●	✓
Entwicklung Milenage	●	✓
Integration SIM-Karte und SIM-Kartenleser	○	✓
Entwicklung Client/Server-Kommunikation	●	✓
Integration PPPoE (Client und Server)	○	✓
Integration aller Komponenten zur Authentisierung	●	✓

Tabelle 5: Ergebnisse der Funktionsprüfung

●=eingetreten; ○=nicht eingetreten; ✓=Erfolg; ✗=Misserfolg

Wie Tabelle 5 zu entnehmen ist, konnten alle gewünschten Funktionen sichergestellt wer-

den. Welche Komplikationen auftraten und mit welchen Mitteln diese behoben wurden, ist im nächsten Kapitel beschrieben (vgl. 4.2: Fehler auf S. 75).

4.1.1 AES

Wie bereits unter Kapitel 3.1.3: Implementierung AES auf S. 59 erläutert, basierte die Implementierungsphase primär auf der Entwicklung der *AES*-Verschlüsselung.

Während der Implementierung wurde das Tool AES-Vizualization⁸⁹ eingesetzt (vgl. 2.5.7: AES Vizualization Tool auf S. 31). Dieses ermöglicht es den *AES*-Algorithmus schrittweise zu durchlaufen. Um die Korrektheit der Implementierung nachzuvollziehen wurde der *Practice-Mode* des Tools verwendet. Nach dem Start des Tools werden zwei Testwerte für den Eingabetext sowie den CypherKey generiert. Diese wurden jeweils in die eigene Implementierung übertragen und dann alle Folgeschritte geprüft.

Auf diesem Weg konnten alle Funktionen vom *SubBytes*, über *ShiftRows*, *MixColumns*, *AddRoundKey* bis hin zum *KeySchedule* erfolgreich verifiziert werden.

Zusätzlich wurden einige Testvektoren, die von Vladimir Klykov veröffentlicht wurden⁹⁰ testweise übernommen und geprüft. Hier wird immer statisch ein Eingabetext sowie ein Cypherkey gelistet, mit dem die Verschlüsselung getestet werden kann. Der Vergleich der generierten Cyphertexte gibt dann Aufschluss darüber ob die Implementierung wie erwünscht arbeitet.

4.1.2 Milenage

Der Austausch verschiedener, generierter bzw. auch statischer Werte zwischen eigenem Implementierung und der SIM-Karte konnte mit *UMTS Security Algorithms* von Fabricio Ferraz⁹¹ geprüft werden. Das Tool wurde auf Empfehlung von Herrn. Prof. Dr. Müller eingesetzt.

Besonders hilfreich ist die Ausgabe aller Zwischenergebnisse von Teilfunktionen *f1-f5* sowie *AUTN*, *Kc* und *SRES* auf den Seiten beider Teilnehmer. Auf diesem Weg konnten alle festen (geheimen) sowie dynamisch generierten Werte in das Tool eingesetzt und Er-

⁸⁹<http://www.cs.mtu.edu/~shene/NSF-4/AES-Downloads/index.html>

⁹⁰<http://www.inconteam.com/software-development/41-encryption/55-aes-test-vectors#aes-ecb-128>

⁹¹<http://fabricioapps.blogspot.de/2011/05/umts-security-algorithm-milenage.html>

gebnisse Schritt für Schritt verifiziert werden. Bei auftretenden Fehlern im Endergebnis konnten Ursachen genauer eingegrenzt und behoben werden.

4.1.3 SIM-Karte und SIM-Kartenleser

Diese beiden Komponenten konnten direkt nach der Treiberintegration im Betriebssystem geprüft werden.

Die Funktionalität des Treibers konnte durch ~~ein entsprechendes identifizieren~~ des Betriebssystems in Form eines PC/SC-Gerät sichergestellt werden.

Teiberinterne Scan-Mechanismen gaben Auskunft über das Vorhandensein einer SIM-Karte, sowie rudimentäre Abfragen. So ~~kann~~ z.B. die IMSI direkt ausgelesen werden.

4.1.4 Client- und Server-Kommunikation

Die Kommunikation zwischen Client- und Server-Komponenten wurden ~~en~~ über die bereits zuvor fertiggestellte Implementierung der jeweiligen Gegenseite getestet.

Auf der Seite des Providers (AuC) wurden zuerst die erwarteten Werte der SIM-Karte, sowie das Initiieren der Authentifizierung hart kodiert. Hierbei wurde natürlich die spätere Reihenfolge des Authentifizierungsvorgangs eingehalten. Nach Sicherstellung dieses Vorgangs konnten umgekehrt die Antworten beziehungsweise Anfragen des Providers auf der Seite des Teilnehmers simuliert werden. Ebenfalls hart kodiert.

Im Anschluss daran wurden beide Teilkomponenten im realen Umfeld verbunden und der Authentifizierungsablauf erneut geprüft.

4.1.5 PPPoE

Der ordnungsgemäße Aufbau der PPPoE-Verbindung ließ sich durch die diensteigene Log-Ausgabe verfolgen. Nach dem Start des Serverdienstes gibt dieser Auskunft darüber, ob jetzt einkommende Verbindungen erwartet werden.

Aus der Sicht des Clients konnten ebenfalls Logdateien verwendet werden.

Nach erfolgreichem Verbindungsaufbau war es möglich die vorhandene IP-Tables-Route zu prüfen. Mit der Auslieferung von Ressourcen aus dem Internet war auch diese Funktion über PPPoE sicherzustellen.

4.2 Fehler

4.2.1 Einrichtung verwendeter Werkzeuge

Raspbian konnte ohne Probleme installiert und in Betrieb genommen werden (vgl. 3.2.1: Aufsetzen des Raspberry Pis auf S. 65). Der einzige Zwischenfall ist auf eine defekte SD-Speicherkarte zurückzuführen. Nach wenigen Wochen des Betriebes ist diese ausgefallen, was ein **starten** des Raspberry Pis unmöglich machte. Ebenfalls konnten die auf der Speicherkarte enthaltenen Daten nicht wieder hergestellt werden.

Aus diesem Grund war es notwendig **das Betriebssystem** erneut aufzusetzen. Eingeleitete Gegenmaßnahme für zukünftige Ausfälle war ein regelmäßiges Backup nach Änderungen am System. Via *dd* wurde immer blockweise ein komplettes Abbild des aktuellen Systems auf externen Festplattenspeicher übertragen.

Es blieb bei einem einzigen Hardwareausfall der SD-Karte.

Ubuntu konnte ohne Probleme installiert sowie betrieben werden. Schon in der Standardinstallation des Server-Images können Netzwerkschnittstellen, von Virtualbox bereitgestellt und direkt konfiguriert beziehungsweise auch genutzt werden.

4.2.2 Fehleinschätzungen

Die meisten und gravierendsten Fehleinschätzungen beziehungsweise Probleme wurden während der Inbetriebnahme der Karte sowie der Entwicklung der Softwarekomponenten aufgedeckt.

pysim Die erste Fehleinschätzung war der Einsatz des Tools pysim, welches primär zu ersten Test in der Kommunikation mit der vorliegenden SIM-Karte angedacht war. Es stellte sich während der Implementierungsphase als problematisch heraus, da es nicht über alle der benötigten Funktionalitäten verfügt. Zwar konnte mit pysim ein erster Kommunikationsversuch erfolgreich gestartet und abgeschlossen **werden**, nicht jedoch eine Authentifizierung initiiert werden. **Dies lag vor allem, dass die beiden Verfassern** dieser Studienarbeit sich nicht sicher waren **welche** Art der Authentifizierung angewandt werden muss. **Die** für eine GSM- oder UMTS-fähige SIM-Karte.

Bei ersten Tests mittels pysim wurde schnell deutlich, dass es sich um eine UMTS-

Authentifizierung handeln muss. Die vorliegende Python-Bibliothek *pysim* ist jedoch nicht dazu in der Lage diese Art von Authentifizierung zu initiieren. Wie unter 2.10.1: *pysim* auf S. 45 genauer erläutert, konzentriert sich *pysim* von Kevin Prince auch eher auf die Programmierung entsprechender SIM-Karten, statt ~~dem universellen Auslesen dieser~~.

Nachdem sich herausstellte, dass UMTS-Kartensupport ebenfalls notwendig ist, fiel die Alternativwahl auf die Bibliothek *osmo-sim-auth* eingesetzt (vgl. 2.10.2: *osmo-sim-auth* auf S. 46). Über diese Bibliothek konnten alle Funktionen auf der Seite des UE in der Sprache Python implementiert werden.

AUTN Unsicherheiten entstanden auch bei ersten Testdurchläufen mit *osmo-sim-auth*. Da zu diesem Zeitpunkt noch keine vollständige Implementierung zur Generierung aller benötigten Werte existierte, wurde die Zusammensetzung für den Parameter *AUTN* zufällig gesetzt. Dies resultierte jedoch darin, dass die SIM-Karte einen Fehler zurück lieferte. Mit Analyse des Fehlercodes *SW 98 62* wurde schnell klar, dass die Zusammensetzung des Parameters *AUTN* zwingend den Vorgaben entsprechen muss, um korrekt zu funktionieren.

Nach der Fertigstellung der Implementierung konnte der Test erneut durchgeführt werden. Dieses mal mit Erfolg. Die SIM-Karte lieferte einen generierten Wert zurück: *AUTS*.

Resynchronisation der SQN Während die verbleibende Implementierungsphase weitestgehend wie geplant verlief, zeichnete sich kurz vor Abschluss der Phase ein weiteres Defizit im Code ab.

Mit der bereits erläuterten Milenage-Prüfsoftware konnten fast alle berechneten, übertragenen sowie von der SIM-Karte empfangenen Werte korrekt verifiziert werden. Ausgehend von den festgelegten Geheimnissen K , $OP(c)$, dem ausgetauschten *RAND* über die berechneten Variablen $f1-f5$ bis hin zu Kc sowie *SRES* auf beiden teilnehmenden Seiten. Der Vergleich dieser Werte deutete darauf hin, dass die Implementierung korrekt sein muss. Problematisch war jedoch die Integration des Codes in die bereitstehende Umgebung. Während die Übertragung der Werte plangemäß verlief, antwortete die SIM-Karte nicht mit den erwarteten Vektoren CK , IK und Kc , sondern lediglich mit einem einzigen Wert. Dieser stellte sich nach wiederholter Recherche in den 3GPP-Dokumenten als *AUTS* her-

aus, welcher zur Resynchronisation der SN benötigt wird (vgl. 2.3.1: Resynchronisation auf S. 12).

Selbige Resynchronisation wurde im entwickelten Protokoll beziehungsweise Authentifizierungsvorgang bis zu diesem Zeitpunkt nicht berücksichtigt. Nach der Identifikation dieses Defizits konnte bei der Implementierung nachgebessert werden.

Alle erforderlichen Informationen waren der Standardisierung durch die $3GPP$ zu entnehmen. Mit dieser Grundlage konnten notwendige Parameter aus der übermittelten $AUTS$ extrahiert und weiterverarbeitet werden, sodass die SN erfolgreich resynchronisiert wird.


Mit der Integration dieser Funktion ~~konnten~~ auch erfolgreich die Vektoren CK , IK und Kc generiert werden.

5 Diskussion

5.1 Alternative Vorgehensweisen

Die Arbeit ließ einiges an Spielraum besonders die Art der Umsetzung. Deshalb soll in diesem Kapitel aufgezeigt werden, was alternativ möglich gewesen wäre.

Eine Möglichkeit wäre sicherlich gewesen für beide Seiten, also UE und AuC, die selbe Programmiersprache zu verwenden. So hätte beides mit C geschrieben werden können, jedoch wäre der Aufwand ein größerer geworden. Das liegt daran, dass für C keine Wrapper-Klasse in dem Umfang von pycard (vgl. Kapitel 2.10.3: pycard auf S. 46) existiert. Somit müssten einige Dinge, die durch pycard und osmo-sim-auth abgewickelt wurden, selbst programmiert werden. Aber die Funktionalität wäre grundsätzlich durch PCSClite gegeben die USIM über C anzusprechen.

Umgekehrt ~~wäre die Möglichkeit gewesen~~ den Netzprovider ebenfalls in Python zu implementieren. Die Fähigkeiten von Python wären für die Aufgabe ausreichend gewesen, zumal keine Optimierung auf Größe oder Geschwindigkeit des Programms gelegt wurde. Durch seine Hardwarenähe hat C in beiden Bereichen gegenüber Python einen Vorteil, aber diese war nicht gefragt. Jedoch hätte es bei Python das Problem gegeben, dass die vorhandenen Kenntnisse sehr gering waren im Gegensatz zu C. Hier wäre also zu Beginn ein größerer Aufwand entstanden ~~die Sprache zu lernen~~ 

Da das Programm zur Steuerung der SIM-Karte sehr geil ist, ist es jedoch kein Widerspruch hier die Zeit für die Grundkenntnisse in Python zu investieren, da sie vermutlich geringer waren als der Zeitaufwand, der für eine Implementierung in C nötig gewesen wäre.

Abschließend ist zu sagen, dass die Liste der Sprachen aus der gewählt werden konnte ziemlich umfangreich ist, aber mit pycard existiert ein guter Wrapper für PCSClite und es ergab Sinn ein Projekt ~~was~~ mit Hardware kommuniziert auch mit einer hardwarenahen Sprache umzusetzen.

5.2 Ausblick

5.2.1 Verbesserungen

Wie in Kapitel 4.1: Funktionsprüfung auf S. 72 angesprochen wurde, sind alle gesetzten Ziele erreicht worden, aber es gab hier und da einige Probleme, die aufgetreten sind und

~~manche von denen wären vermutlich vermeidbar gewesen.~~ Dazu zählen vor allem Fehler bei der Entwicklung des AES- und Milenage-Algorithmen.

Lange bestand eine unterschiedliche Meinung darüber, wo der Unterschied von UMTS- und GSM-Authentifizierung lag. Und trotz vermehrter kurzer Unterhaltungen in denen dies offensichtlich wurde, hat es lange gedauert bis ein längeres Gespräch stattfand in dem gemeinsam geklärt wurde, wie sich beide unterscheiden. Wäre dieser Diskurs schon früher geführt worden, wäre eine erste Implementierung mit pysim vermutlich nicht geschehen, sondern die Suche wäre weiter gegangen und schon früher auf osmo-sim-auth gefallen.

Der andere große Punkt war, ~~dass nicht bekannt war,~~ wie und wer den Wert von SQN verantwortet. Der Wert tauchte zwar immer wieder auf, aber in den Dokumenten, die gelesen wurden war dieser Wert im Gegensatz zu allen anderen nicht näher spezifiziert. Deshalb wurde zu Beginn angenommen, dass der Netzprovider SQN entscheidet und durch den AUTN die SIM-Karte diesen vermittelt bekommt. Es stellte sich jedoch ~~raus,~~ dass SQN von der USIM verwaltet wird und es sehr wohl eine Methode zur Berechnung der SQN gibt, aber es wurde an den falschen Stellen danach gesucht und ~~versuchte~~ deshalb kurz vor Projektabschluss nochmal Probleme. Ein Lesen der Referenzen der Milenage-Spezifikation 35.205 vom 3GPP hätte direkt zu dem Dokument 33.102 geführt, in ~~der~~ auch die SQN erläutert wurde. Hier gilt es beim nächsten Mal nicht erst Google für die Suche zu bemühen sondern das zu durchsuchen, was schon existiert.

5.2.2 Mögliche zukünftige Projekte

Die Authentifizierung und der Schlüsselaustausch von UMTS noch umfangreicher als ~~in~~ in diesem Projekt umgesetzt ~~wurde.~~ Da es in diesem Projekt allein um die Authentifizierung ging, wurde die Berechnung von CK und IK vernachlässigt. Dies wäre in einem weiteren Schritt eine Möglichkeit. Dadurch könnten dann auch Nachrichten zwischen Nutzergerät und Authentifizierungsstelle verschickt werden. Noch ein Schritt weiter wäre ~~eventuell~~ eine weitere SIM-Karte, die sich beim Netzprovider authentifiziert und dann mit der jetzigen USIM kommuniziert.

Für diesen Schritt müsste jedoch vorher das Programm auf dem AuC angepasst werden und eine Datenbank erhalten, in der die unterschiedlichen Parameter für die einzelnen USIM's abgespeichert werden.

Generell erlaubt der aktuelle Prozess nur USIM-Karten. Hier wäre zu überlegen für

SIM-Karten auf dem GSM-Standard ebenfalls die Authentifizierung zu ermöglichen. Der Milenage-Algorithmus ermöglicht die Abwärtskompatibilität, weshalb kein neuer Standard verstanden werden muss und der zusätzliche Programmieraufwand nicht sehr groß sein wird.

Generell sollte auch der Programmcode optimiert werden. An einigen Stellen können Programmteile in ~~eigene~~ Funktionen ausgegliedert werden, um Redundanzen zu vermeiden und das Programm selbst im Speicherplatzverbrauch zu reduzieren. Des weiteren wird schon jetzt eine ähnliche Aufgabe durch `rotWord` und `shiftRows` ~~gegeben~~, weshalb `shiftRows` einfach `rotWord` implementieren könnte.

Neben der Reduzierung von Redundanzen kann überlegt werden, wo sich die Geschwindigkeit des Programms optimieren lässt. Am Ende ist den Autoren zum Beispiel aufgefallen, dass es für die Umwandlung des Schlüssels und des Textblocks in das Array eine schnellere Variante gibt als die aktuelle. Dabei ist der Code dann doch zeilenweise in dem Array gespeichert und nicht wie aktuell spaltenweise. Damit müssten auch die einzelnen Transformations-Funktionen umgebaut werden, was in Anbetracht der Zeit in diesem Projekt nicht mehr umsetzbar war.

So gibt es aber vermutlich noch einige Stellen an denen eine Optimierung unter dem Aspekt Performanz und wahrscheinlich auch Speicherbedarf möglich ist.

Als nächster Punkt für zukünftige Erweiterungen bestünde die Möglichkeit den Start der Authentifizierung automatisiert zu erkennen. So bietet PCSClite die Möglichkeit eine Meldung zu erhalten, wenn in das Kartenlesegerät ~~die~~ SIM-Karte gesteckt wird, beziehungsweise auch wenn diese wieder entfernt wird. Damit könnte der Server dauerhaft laufen und der Raspberry Pi würde sich ohne den manuellen Start durch einen Nutzer beim Netzprovider registrieren.

Als ~~letzten sinnvollen~~ Punkt wird das Abbilden der realen Struktur gesehen. Das bedeutet, dass eine weitere virtuelle Maschine als MSC benötigt würde. Außerdem sollten dann auch die jeweiligen MAC-Werte und der RES an der richtigen Stelle überprüft werden mit der richtigen Reaktion durch die einzelnen Kommunikationspartner.

6 Appendix sections

6.1 Abbildungen

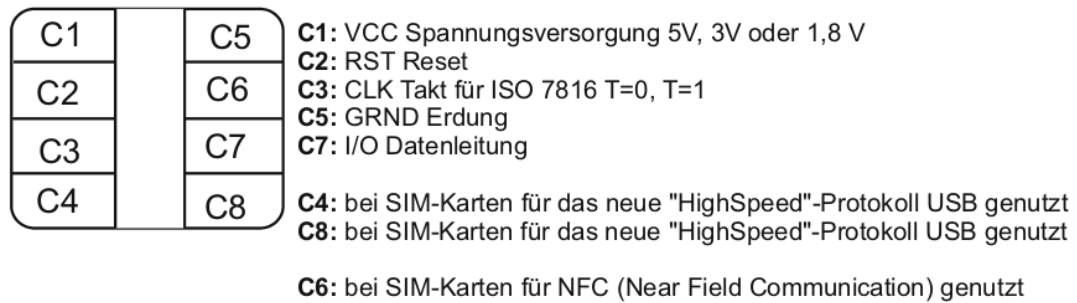


Abbildung 10: Pinbelegung einer Chipkarte nach ISO/IEC 786 [1]

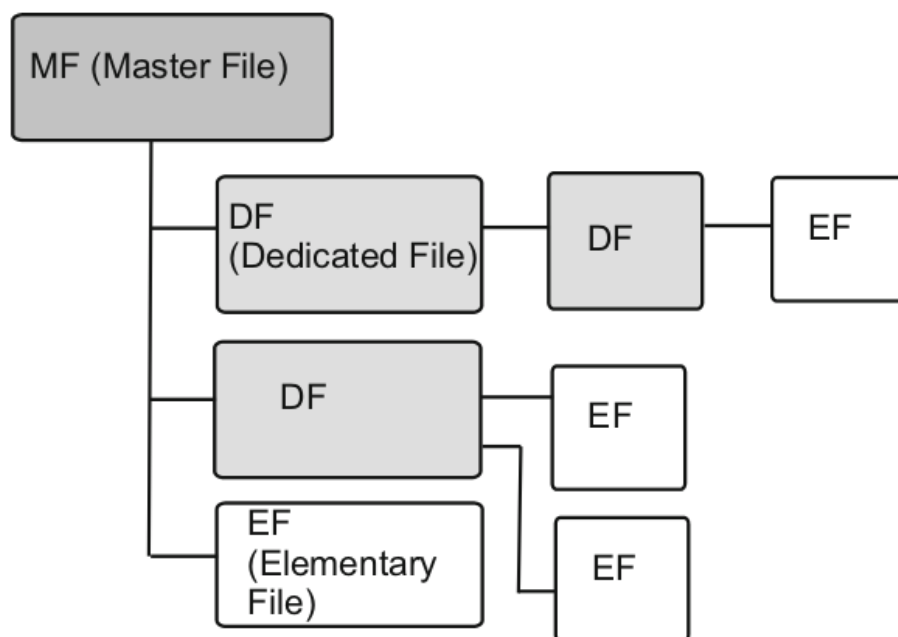


Abbildung 11: Filesystemarchitektur einer Chipkarte nach ISO/IEC 786 [1]

Befehlsklasse	ISO Kommando	Zweck
Dateiverwaltung	SEL, Select File	Auswahl einer Datei
	RDR, Read Record	Lesen eines Records in einer Datei
	WRR, Write Record	Schreiben eines Records in einer Datei
Authentisierung	VER, Verify PIN	Authentisierung über eine PIN
	GCH, Get Challenge	Anforderung einer Zufallszahl für eine Authentisierung
	AIN, Internal Auth.	Authentisierung der Karte beim Terminal
	AEX, External Auth.	Authentisierung des Terminals bei der Karte
Kryptographie	AMU, Muth. Auth.	Gegenseitige Authentisierung von Karte und Terminal
	CRY, Crypto Op.	Verschlüsselung oder MAC-Berechnung
Zählervariablen	INC, Increment	Erhöhen eines Zählers
	DEC, Decrement	Erniedrigen eines Zählers

Abbildung 12: Befehlsklassen einer Chipkarte nach ISO/IEC 786 [1]

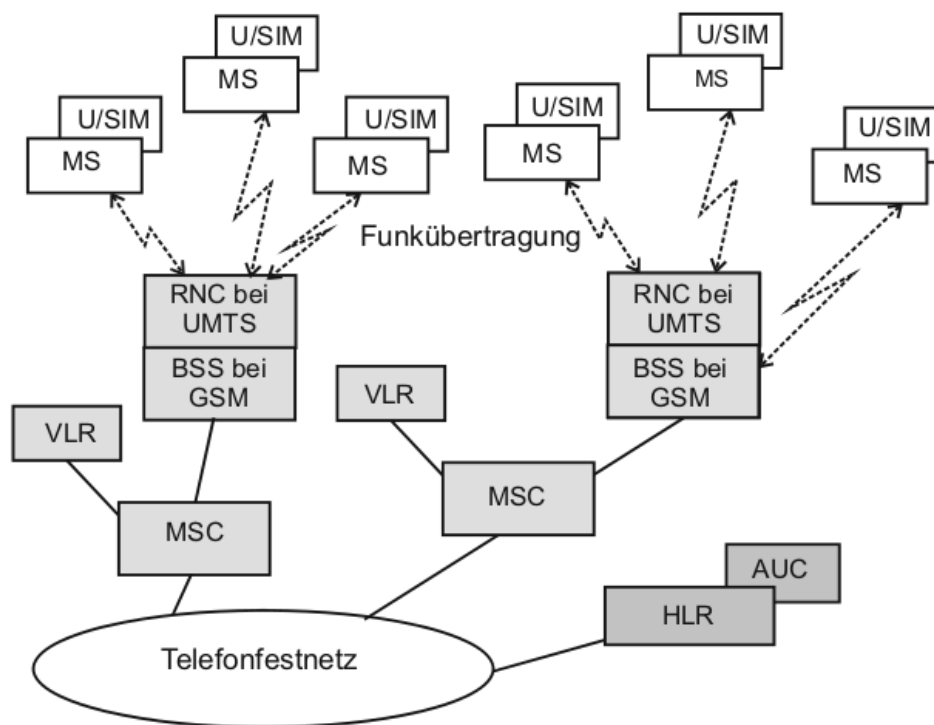


Abbildung 13: Teilnehmer in Mobilfunknetzen (GSM und UMTS) [1]



Abbildung 14: Architektur eines Typ-1-Hypervisors [24]

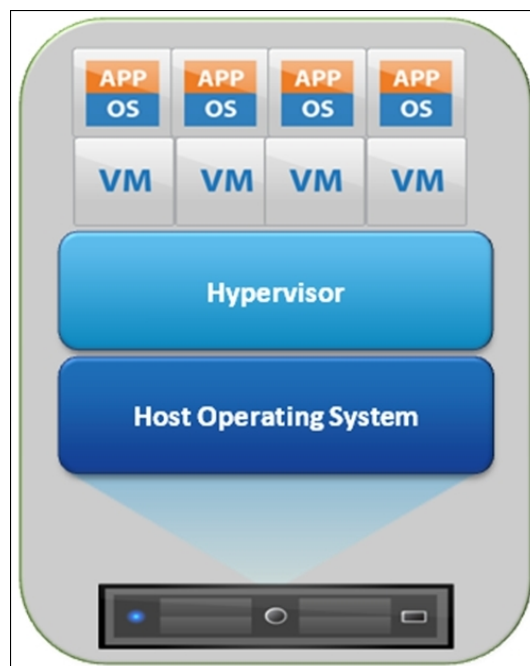


Abbildung 15: Architektur eines Typ-2-Hypervisors [24]

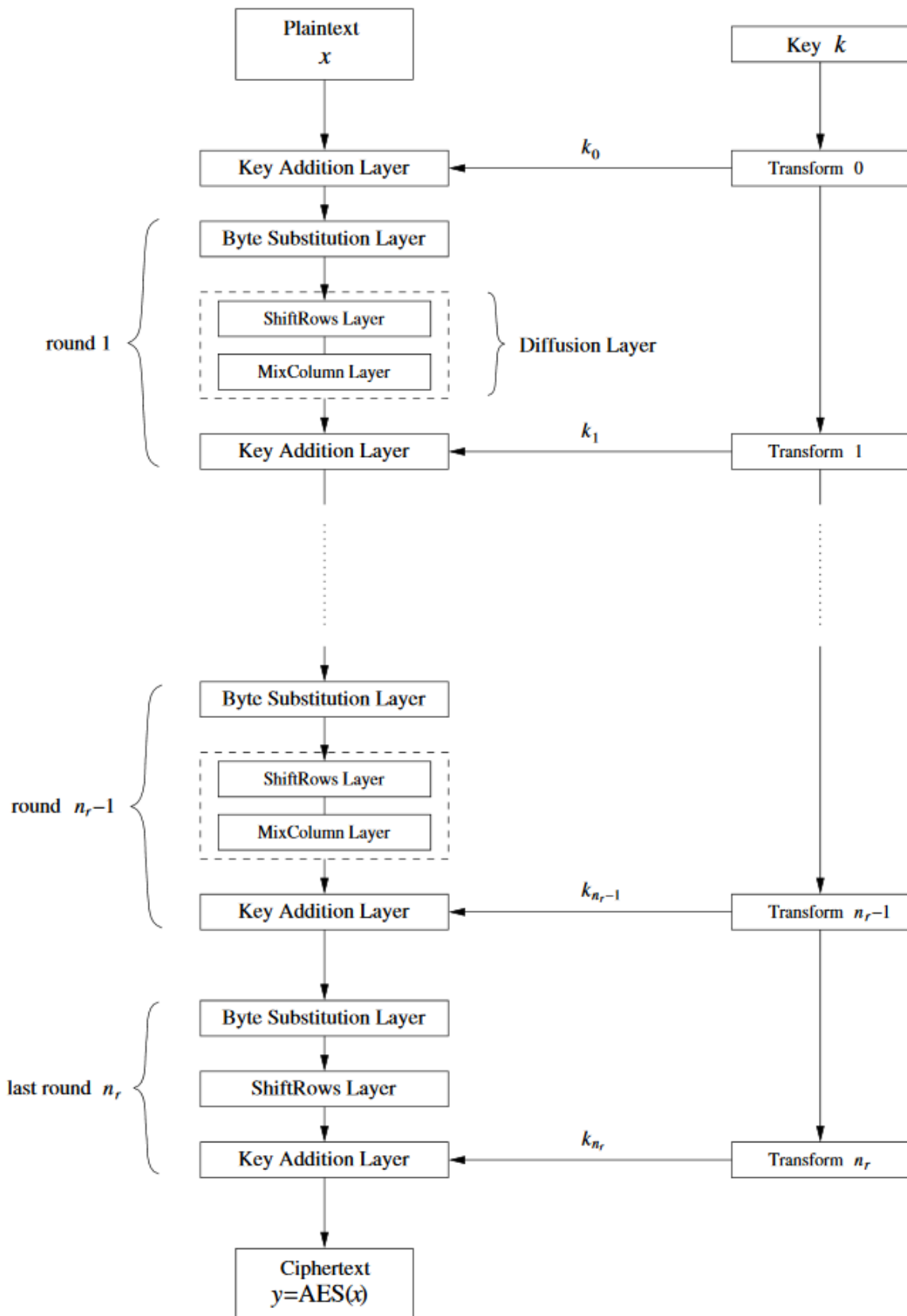


Abbildung 16: Grafische Darstellung der AES-Verschlüsselung [9]

	y															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Abbildung 17: Substitutionstabelle für AES-Verschlüsselung [9]

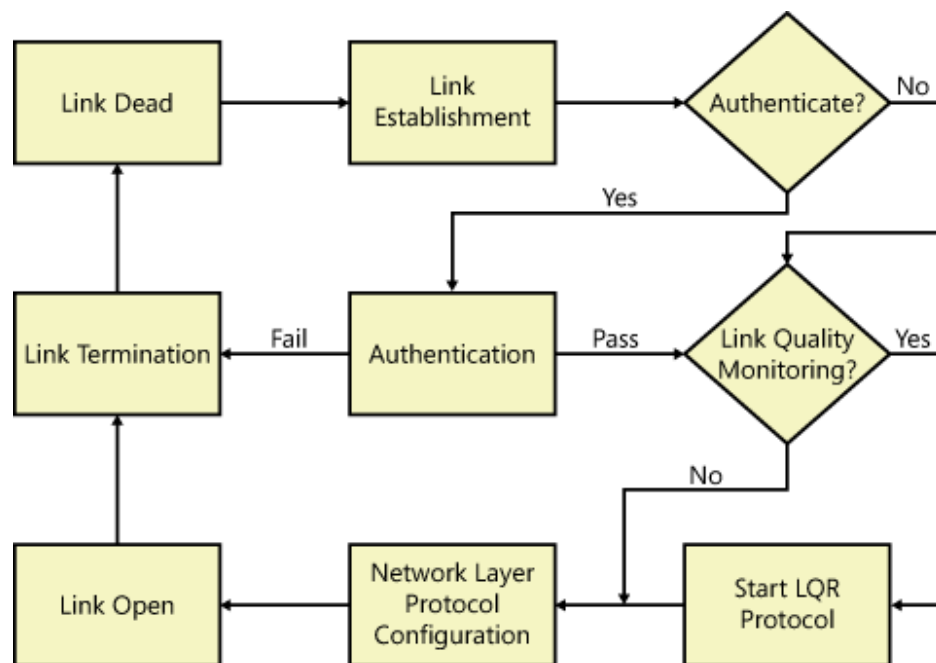


Abbildung 18: Aufbauphasen einer PPP-Verbindung [26]

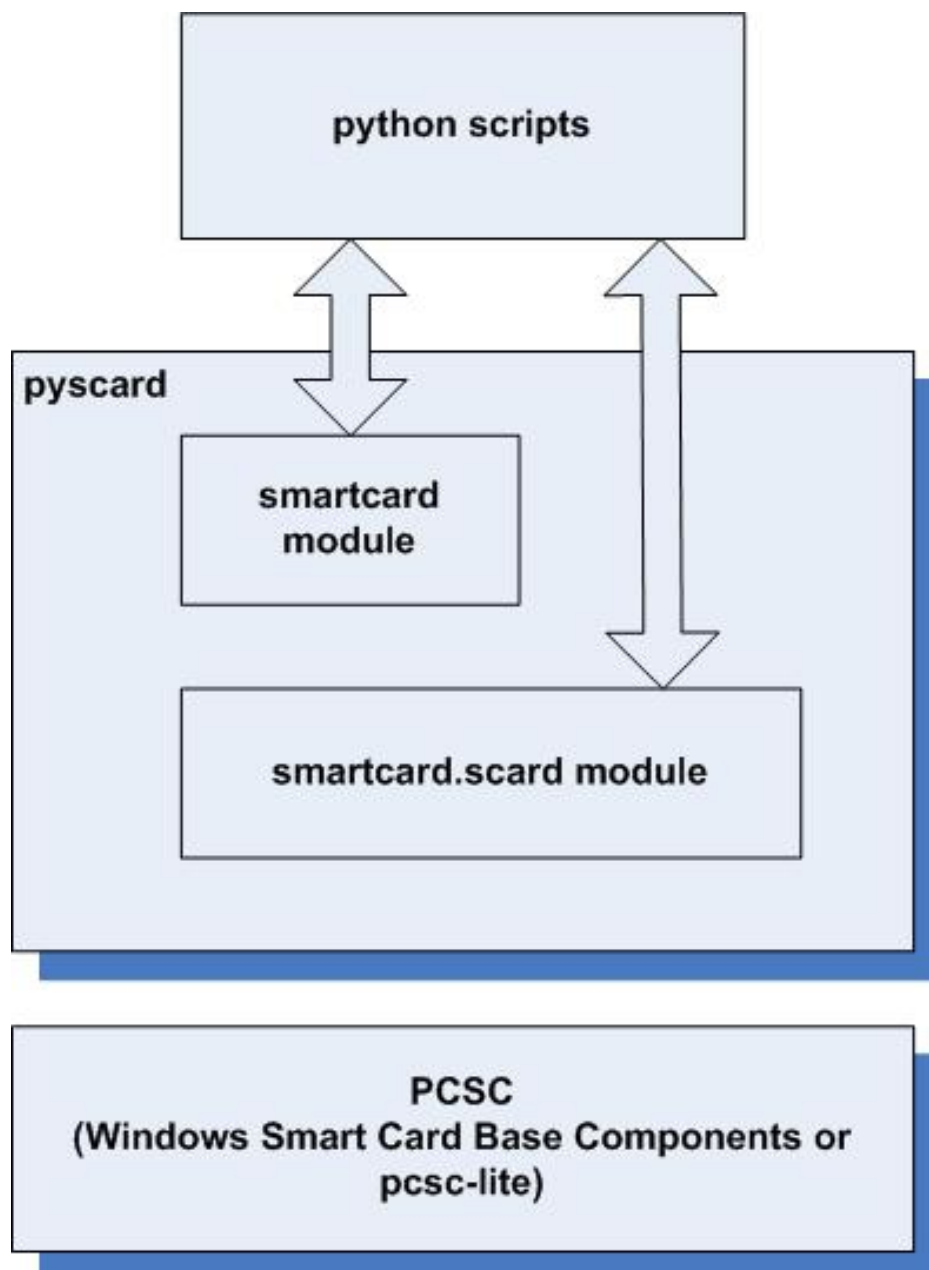


Abbildung 19: Architektur der pycard-Bibliothek [27]

6.2 Listings

```
/*
 * File:    main.c
 * Author:  Marco Heumann
 *
 * Created on 29. Januar 2016
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include "milenage.h"

typedef unsigned char u8;

#define ANSLCOLOR_RED      "\x1b[31m"
#define ANSLCOLOR_GREEN   "\x1b[32m"
#define ANSLCOLOR_YELLOW  "\x1b[33m"
#define ANSLCOLOR_BLUE    "\x1b[34m"
#define ANSLCOLOR_RESET   "\x1b[0m"

u8 inputArr[16], outputArr[16], auts[14];
u8 mrand[16];
u8 keyArr[16] = {0x14, 0x5D, 0xD6, 0x1C, 0xE1, 0xF1, 0x9B, 0x84, 0x96, 0
    x3C, 0x09, 0x8C, 0x84, 0xDF, 0x1B, 0x98};
int i, status;
int socket_desc, client_sock, c, read_size, optval;
struct sockaddr_in server, client;
char client_message[50], imsi[15], response_message[96];

int main(int argc, char** argv) {
    //Create socket
    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_desc == -1) {
        printf(ANSIColor_GREEN "Could not create socket" ANSLCOLOR_RESET
            );
    }
    puts("Socket created");
```

```
//Prepare the sockaddr_in structure
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(12345);

if (bind(socket_desc, (struct sockaddr *) &server, sizeof (server)) <
    0) {
    perror("bind failed. Error");
    return 1;
}
puts("bind done");

listen(socket_desc, 3);

puts("Waiting for incoming connections...");
c = sizeof (struct sockaddr_in);

client_sock = accept(socket_desc, (struct sockaddr *) &client, (
    socklen_t*) &c);
if (client_sock < 0) {
    perror(ANSIColor.RED "accept failed" ANSIColor.RESET);
    return 1;
}
puts(ANSIColor.GREEN "Connection accepted" ANSIColor.RESET);

//Receive a message from client
while ((read_size = recv(client_sock, client_message, 50, 0)) > 0) {
    char* temp;
    temp = malloc(2);

    strncpy(temp, &client_message[0], 1);
    status = strtol(temp, NULL, 10);

    printf(ANSIColor.BLUE "\nReceived message ");
    if (status == 1) {
        printf("(Status[1] || IMSI[15] )");
    } else {
        printf("(Status[1] || IMSI[15] || AUTS[28] )");
    }
    printf(": \n" ANSIColor.RESET);
    puts(client_message);
}
```

```

    strncpy(imsi, &client_message[1], 15);

    response_message[0] = 50; // 50h -> 2d

    printf(ANSIColor.BLUE "\nCalculated values:" ANSLColor.RESET);

    if (status == 3) {
        for (i = 8; i < 22; i++) {
            strncpy(temp, &client_message[i * 2], 2);
            auts[i-8] = strtol(temp, NULL, 16);
        }

        f5star(keyArr, auts);
        response_message[0] = 52; // 52h -> 4d
    }
    free(temp);

    genRand(mrand, &response_message[16]);
    f1(keyArr, mrand);
    f2_5(keyArr, &response_message[72]);
    genAutn(&response_message[48]);

    for(i = 0; i < 15; i++) {
        response_message[i+1] = imsi[i];
    }

    //Send the message back to client
    write(client_sock, response_message, strlen(response_message));

    printf(ANSIColor.BLUE "\n\nSent message (Status[1] || IMSI[15] || RAND
        [32] || AUTN[32]): \n" ANSLColor.RESET);
    puts(response_message);
}

if (read_size == 0) {
    puts(ANSIColor.YELLOW "\nClient disconnected" ANSLColor.RESET);
    fflush(stdout);
} else if (read_size == -1) {
    perror(ANSIColor.RED "\nrecv failed" ANSLColor.RESET);
}

printf("\n");

```

```
    return (EXIT_SUCCESS);  
}
```

Listing 1: main.c

```
/*  
 * File:   milenage.c  
 * Author: Marco Heumann  
 *  
 * Created on 05. Maerz 2016  
 */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include "rijndael.h"  
  
typedef unsigned char u8;  
  
void f1(u8*, u8*);  
void convertToBin(u8*, u8*);  
void convertToHex(u8*, u8*);  
void genRand(u8*, u8*);  
void genAutn(u8*);  
void rotWord(u8*, int, int);  
void convertToBin(u8*, u8*);  
  
u8 opc[16] = {0x25, 0x50, 0x07, 0x6C, 0x5E, 0xF5, 0x9F, 0x77, 0xEE, 0xBE,  
             0xF1, 0xCA, 0x39, 0x91, 0x6E, 0xC8};  
u8 c1[16] = {0x7C, 0x1E, 0xE3, 0x6E, 0x98, 0xC2, 0x74, 0x40, 0xCA, 0x1D, 0  
             x58, 0xF7, 0xE8, 0xD3, 0x7D, 0x2F};  
u8 c2[16] = {0xC1, 0xFC, 0xBC, 0xAB, 0xC7, 0x3E, 0xE4, 0xA2, 0xF3, 0x62, 0  
             x82, 0x11, 0xB8, 0x7A, 0xED, 0x04};  
u8 c3[16] = {0x16, 0x07, 0x87, 0xD0, 0x24, 0xC2, 0xA5, 0x1D, 0xC4, 0x21, 0  
             x4A, 0x3A, 0xE4, 0x3F, 0x5A, 0x34};  
u8 c4[16] = {0x69, 0xA6, 0xF7, 0x01, 0x54, 0x77, 0x69, 0x17, 0x85, 0xD8, 0  
             x0C, 0xA1, 0xBB, 0x7A, 0x0D, 0x93};  
u8 c5[16] = {0xFD, 0x2E, 0xF6, 0x92, 0xE4, 0x14, 0xFB, 0x2E, 0x93, 0xE3, 0  
             x9C, 0x0A, 0x92, 0xD0, 0xE9, 0xAB};  
u8 sqn[6] = {0xff, 0x9b, 0xb4, 0xd0, 0xb6, 0x07};  
u8 amf[2] = {0xb9, 0xb9};  
u8 binArr[128], temp[16], res[8], ak[6], in1[16], toEncrypt[16], out1[16];
```

```
int i;

void f1(u8* keyArr, u8* mrand) {
    printf("\nSQN: ");
    for (i = 0; i < 6; i++) {
        printf("%02x", sqn[i]);
    }

    printf("\nAMF: ");
    for (i = 0; i < 2; i++) {
        printf("%02x", amf[i]);
    }

    // create IN1
    for (i = 0; i < 6; i++) {
        in1[i] = sqn[i];
        in1[i + 8] = sqn[i];
    }
    for (i = 0; i < 2; i++) {
        in1[i + 6] = amf[i];
        in1[i + 14] = amf[i];
    }

    printf("\nRAND: ");
    for (i = 0; i < 16; i++) {
        printf("%02x", mrand[i]);
    }

    for (i = 0; i < 16; i++) {
        toEncrypt[i] = mrand[i] ^ opc[i];
    }
    encrypt(toEncrypt, keyArr, temp);

    for (i = 0; i < 16; i++) {
        out1[i] = in1[i] ^ opc[i];
    }

    convertToBin(out1, binArr);
    rotWord(binArr, 128, 0x1F);
    convertToHex(binArr, out1);

    for (i = 0; i < 16; i++) {
```

```
        out1[i] ^= temp[i]^c1[i];
    }
    encrypt(out1, keyArr, out1);

    for (i = 0; i < 16; i++) {
        out1[i] ^= opc[i];
    }

    printf("\nMAC-A: ");
    for (i = 0; i < 8; i++) {
        printf("%02x", out1[i]);
    }

    printf("\nMAC-S: ");
    for (i = 8; i < 16; i++) {
        printf("%02x", out1[i]);
    }
}

void f2_5(u8* keyArr, u8* response_arr) {
    u8* out2;
    out2 = malloc(16);

    for (i = 0; i < 16; i++) {
        out2[i] = temp[i] ^ opc[i];
    }

    convertToBin(out2, binArr);
    rotWord(binArr, 128, 0x3A);
    convertToHex(binArr, out2);

    for (i = 0; i < 16; i++) {
        out2[i] ^= c2[i];
    }
    encrypt(out2, keyArr, out2);

    for (i = 0; i < 16; i++) {
        out2[i] ^= opc[i];
    }

    printf("\r\nAK: ");
    for (i = 0; i < 6; i++) {
```

```
        ak[i] = out2[i];
        printf("%02x", ak[i]);
    }
    printf("\r\nRES: ");
    for (i = 8; i < 16; i++) {
        sprintf(&response_arr[(i-8)*2], "%02x", out2[i]);
        printf("%02x", out2[i]);
    }
}

void f5star(u8* keyArr, u8* sqn_ak) {
    u8* out5;
    out5 = malloc(16);

    for (i = 0; i < 16; i++) {
        out5[i] = temp[i] ^ opc[i];
    }

    convertToBin(out5, binArr);
    rotWord(binArr, 128, 0x08);
    convertToHex(binArr, out5);

    for (i = 0; i < 16; i++) {
        out5[i] ^= c5[i];
    }
    encrypt(out5, keyArr, out5);

    for (i = 0; i < 16; i++) {
        out5[i] ^= opc[i];
    }

    printf("\r\nAK (f5*): ");
    for (i = 0; i < 6; i++) {
        ak[i] = out5[i];
        printf("%02x", ak[i]);
    }

    for (i = 0; i < 6; i++) {
        sqn[i] = ak[i] ^ sqn_ak[i];
    }

    u8 ind = (sqn[5] & 0b00011111);
```

```
    u8 seq = (sqn[5] & 0b11100000);
    ind = (ind + 1) % 32;
    sqn[5] = 0;
    sqn[5] |= ind;

    seq += 0b00100000;
    if (seq == 0) {
        sqn[4] += 1;
    }
    sqn[5] |= seq;
}

void convertToBin(u8* hexArr, u8* binArr) {
    int i, j;
    unsigned char mask = 1;

    for (j = 0; j < 16; j++) {
        for (i = 0; i < 8; i++) {
            binArr[j * 8 + i] = (hexArr[j] & (mask << (7 - i))) != 0;
        }
    }
}

void convertToHex(u8* binArr, u8* hexArr) {
    int i, arrpos; //i is for calculating each 8 bit to hex

    for (arrpos = 0; arrpos < 16; arrpos++) {
        hexArr[arrpos] = 0;
        for (i = 0; i < 8; i++) {
            hexArr[arrpos] <<= 1;
            hexArr[arrpos] |= binArr[arrpos*8 + i];
        }
    }
}

void genAutn(u8* response_arr) {
    printf("\nAUTN: ");
    for (i = 0; i < 6; i++) {
        sprintf(&response_arr[i*2], "%02x", (sqn[i] ^ ak[i]));
        printf("%02x", (sqn[i] ^ ak[i]));
    }
    for (i = 0; i < 2; i++) {
```

```

        sprintf(&response_arr[(i+6)*2], "%02x", amf[i]);
        printf("%02x", amf[i]);
    }
    for (i = 0; i < 8; i++) {
        sprintf(&response_arr[(i+8)*2], "%02x", out1[i]);
        printf("%02x", out1[i]);
    }
}

void genRand(u8* mrand, u8* response_arr) {
    srand(time(NULL));
    for (i = 0; i < 16; i++) {
        mrand[i] = (unsigned)rand() % 255;
        sprintf(&response_arr[i*2], "%02x", mrand[i]);
    }
}

void reverse(u8* a, int sz) {
    int i, j;
    for (i = 0, j = sz; i < j; i++, j--) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
}

void rotWord(u8* array, int size, int amt) {
    reverse(array, amt - 1);
    reverse(array + amt, size - amt - 1);
    reverse(array, size - 1);
}

```

Listing 2: milenage.c

```

/*
 * File:   rijandael.c
 * Author: Marco Heumann
 *
 * Created on 29. Januar 2016
 */

#include <stdio.h>
#include <stdlib.h>

```

```
typedef unsigned char u8;

int t2(int);
u8 getSboxValue(int);
void addRoundKey(int);
void generateRoundKey();
void shiftRow();
void mixColumn();

static const u8 Sbox[256] = {
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B
        , 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF
        , 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1
        , 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2
        , 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3
        , 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39
        , 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F
        , 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21
        , 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D
        , 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14
        , 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62
        , 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA
        , 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F
        , 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9
        , 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9
        , 0xCE, 0x55, 0x28, 0xDF,
```

```

    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F
    , 0xB0, 0x54, 0xBB, 0x16
};
/* Round constants */
static const u8 Rcon[10] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80
    , 0x1B, 0x36};
u8 mixMatrix[4][4] = {
    {0x02, 0x03, 0x01, 0x01},
    {0x01, 0x02, 0x03, 0x01},
    {0x01, 0x01, 0x02, 0x03},
    {0x03, 0x01, 0x01, 0x02}
};
u8 tmp[4] = {0, 0, 0, 0};

u8 key[4][4], roundKey[11][4][4], state[4][4];
int i, j, k, r, z;

void convert2array(u8 input[16], u8 output[4][4]) {
    for (j = 0; j < 4; j++) {
        for (i = 0; i < 4; i++) {
            output[i][j] = input[(j * 4 + i)];
        }
    }
}

void encrypt(u8 input[16], u8 keyStr[16], u8 output[16]) {
    convert2array(input, state);
    convert2array(keyStr, key);

    // Round 0
    generateRoundKey();
    addRoundKey(0);

    // Round 1-9
    for (r = 1; r < 10; r++) {
        for (i = 0; i < 4; i++) {
            for (j = 0; j < 4; j++) {
                state[i][j] = getSboxValue(state[i][j]);
            }
        }
        shiftRow();
        mixColumn();
    }
}

```

```
        addRoundKey(r);
    }

    // Round 10
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            state[i][j] = getSboxValue(state[i][j]);
        }
    }
    shiftRow();
    addRoundKey(r);

    for (j = 0; j < 4; j++) {
        for (i = 0; i < 4; i++) {
            output[(j * 4 + i)] = state[i][j];
        }
    }
}

u8 getSboxValue(int num) {
    return Sbox[num];
}

u8 getRconValue(int num) {
    return Rcon[num];
}

void shiftRow() {
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            tmp[j] = state[i][j];
        }
        for (j = 0; j < 4; j++) {
            state[i][j] = tmp[(j + i) % 4];
        }
    }
}

void mixColumn() {
    for (j = 0; j < 4; j++) {
        for (i = 0; i < 4; i++) {
            tmp[i] = state[i][j];
```

```
    }
    for (i = 0; i < 4; i++) {
        for (z = 0; z < 4; z++) {
            switch (mixMatrix[i][z]) {
                case 3:
                    if (z == 0) {
                        state[i][j] = t2(tmp[z]) ^ tmp[z];
                    } else {
                        state[i][j] ^= t2(tmp[z]) ^ tmp[z];
                    }
                    break;
                case 2:
                    if (z == 0) {
                        state[i][j] = t2(tmp[z]);
                    } else {
                        state[i][j] ^= t2(tmp[z]);
                    }
                    break;
                case 1:
                default:
                    if (z == 0) {
                        state[i][j] = tmp[z];
                    } else {
                        state[i][j] ^= tmp[z];
                    }
            }
        }
    }
}

void addRoundKey(int round) {
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            state[i][j] ^= roundKey[round][i][j];
        }
    }
}

void generateRoundKey() {
    for (j = 0; j < 4; j++) {
        for (i = 0; i < 4; i++) {
```

```
        roundKey[0][i][j] = key[i][j];
    }
}
for (k = 1; k < 11; k++) {
    tmp[0] = roundKey[k - 1][0][3];

    // RotWord
    for (i = 0; i < 4; i++) {
        if (i < 3) {
            roundKey[k][i][0] = roundKey[k - 1][i + 1][3];
        } else {
            roundKey[k][i][0] = tmp[0];
        }
    }

    for (i = 0; i < 4; i++) {
        // Substitute with Sbox
        roundKey[k][i][0] = getSboxValue(roundKey[k][i][0]);

        // XOR Rcon
        if (i == 0) {
            roundKey[k][i][0] ^= roundKey[k - 1][i][0] ^ getRconValue(
                k - 1);
        } else {
            roundKey[k][i][0] ^= roundKey[k - 1][i][0];
        }
    }

    // XOR Rest
    for (j = 1; j < 4; j++) {
        for (i = 0; i < 4; i++) {
            roundKey[k][i][j] = roundKey[k - 1][i][j] ^ roundKey[k][i
                ][j - 1];
        }
    }
}

int t2(int a) {
    if (a < 128) {
        return 2 * a;
    } else if (a >= 128) {
```

```

        return (2 * a) ^ 283;
    }
}

```

Listing 3: rijndael.c

```

#!/usr/bin/python

import socket
from binascii import *
from card.utils import *
from optparse import OptionParser
from card.USIM import USIM
from card.SIM import SIM

def handle_usim(options, rand_bin, autn_bin):
    u = USIM()
    if not u:
        print "Error opening USIM"
        exit(1)

    if options.debug:
        u.dbg = 2;

    imsi = u.get_imsi()
    ret = u.authenticate(rand_bin, autn_bin, ctx='3G')
    if len(ret) == 1:
        print "AUTS:\t%s" % b2a_hex(byteToString(ret[0]))
    else:
        print "RES:\t%s" % b2a_hex(byteToString(ret[0]))
        print "CK:\t%s" % b2a_hex(byteToString(ret[1]))
        print "IK:\t%s" % b2a_hex(byteToString(ret[2]))
        if len(ret) == 4:
            print "Kc:\t%s" % b2a_hex(byteToString(ret[3]))

    #ret = u.authenticate(rand_bin, autn_bin, ctx='2G')
    #if not len(ret) == 2:
    #    print "Error during 2G authentication"
    #    exit(1)
    #print "SRES:\t%s" % b2a_hex(byteToString(ret[0]))
    #print "Kc:\t%s" % b2a_hex(byteToString(ret[1]))

if __name__ == "__main__":

```



```
u = USIM()
u.debug = 2
imsi = u.get_imsi()

s = socket.socket()
host = socket.gethostname()
#host = '192.168.2.254'
port = 12345
s.connect((host, port))
authenticated = False
status = 0

while True:
    print "\n"
    if authenticated == False:
        if status == 2:
            print "## auth=false; status=2 - trying to send imsi+auts"
            s.send("3"+imsi+auts)
        else:
            print "## auth=false; status!=2 - trying to send imsi only"
            s.send("1"+imsi)
        recvmsg = s.recv(1024)
        status = recvmsg[0:1]
        rand = recvmsg[16:48]
        print "RAND: "+rand
        autn = recvmsg[48:80]
        print "AUTN: "+autn

        rand_bin = stringToByte(a2b_hex(rand))
        autn_bin = stringToByte(a2b_hex(autn))

        ret = u.authenticate(rand_bin, autn_bin, ctx='3G')
        if len(ret) == 1:
            print "##### auth=false; status!=2; len(ret)=1 - trying to
                authenticate -> got auts"
            auts = b2a_hex(byteToString(ret[0]))
            print "AUTS: " + auts
            status = 2
        else:
            print "##### auth=false; status!=2, len(ret)>1 - trying to
                authenticate -> got triple"
            res = b2a_hex(byteToString(ret[0]))
```

```
print "RES: " + res
ck = b2a_hex(byteToString(ret[1]))
print "CK: " + ck
ik = b2a_hex(byteToString(ret[2]))
print "IK : " + ik
authenticated = True

s.close
```

Listing 4: client.py

Literatur

- [1] Spitz, S., Pramateftakis, M. und Swoboda, J., *Kryptographie und IT-Sicherheit: Grundlagen und Anwendungen*, Studium : IT-Sicherheit und Datenschutz, Vieweg+Teubner Verlag, 2011.
- [2] Rankl, W. und Effing, W., *Handbuch der Chipkarten*, 4. Auflage, Carl Hanser Verlag, 2002.
- [3] Antipolis, S., Numbering, addressing and identification, TS 33.003, 3rd Generation Partnership Project (3GPP), 2010.
- [4] Horn, G., 3G security; Security architecture, TS 33.102, 3rd Generation Partnership Project (3GPP), 2015.
- [5] Pütz, S., Schmitz, R. und Martin, T., Security mechanisms in UMTS, *Datenschutz und Datensicherheit* **25** (2001).
- [6] Walker, M., 3G Security; Specification of the MILENAGE algorithm set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 1: General, TS 35.205, 3rd Generation Partnership Project (3GPP), 2015.
- [7] Daemen, J. und Rijmen, V., *The design of Rijndael: AES — the Advanced Encryption Standard*, Springer-Verlag, 2002.
- [8] Specification for the Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197, 2001.
- [9] Paar, C. und Pelzl, J., *Understanding Cryptography: A Textbook for Students and Practitioners*, Kapitel The Advanced Encryption Standard (AES), Seiten 87–121, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [10] Schmeh, K., *Kryptografie - Verfahren, Protokolle, Infrastrukturen*, dpunkt.verlag, 3rd edition, 2007.
- [11] Information technology – Security techniques – Modes of operation for an n-bit block cipher, Standard, International Organization for Standardization, Geneva, CH, 2006.
- [12] Keranan, M., Mayo, J. und Shene, C.-K., Michigan technology university cryptography visualization software, <http://www.cs.mtu.edu/~shene/NSF-4>, 2015.
- [13] Stevens, R. und Fall, K., *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley Professional, 2011.
- [14] Skoll, D. F., Pppoe and linux, <https://www.roaringpenguin.com/files/pppoe-slides.pdf>, 2000.
- [15] Corcoran, D. und Rousseau, L., Pcsclite project website, <https://pcsclite.alioth.debian.org/pcsclite.html>, 2016.

- [16] Ritchie, D. M., The Development of the C Language, SIGPLAN Not. **28** (1993) 201.
- [17] Nørmark, K., Functional programming in scheme - with web programming examples, <http://people.cs.aau.dk/~normark/prog3-03/pdf/all.pdf> [besucht am 19.05.2016], 2003.
- [18] Kernighan, B. W., *The C Programming Language*, Prentice Hall Professional Technical Reference, 2nd edition, 1988.
- [19] Prince, K., pysim github repository, <https://github.com/kevinprince/pysim>, 2016.
- [20] Software, E., osmo-sim-auth website, <http://openbsc.osmocom.org/trac/wiki/osmo-sim-auth>, 2013.
- [21] Software, E., osmocom openbsc website, <http://openbsc.osmocom.org/trac/wiki>, 2016.
- [22] Foundation, R. P., Raspberry pi foundation, <https://www.raspberrypi.org/about/>, 2016.
- [23] Website, R., Raspbian website, <https://www.raspbian.org/>, 2016.
- [24] Dash, P., *Getting Started with Oracle VM VirtualBox*, Packt Publishing, 2013.
- [25] Victor, J., Savit, J., Combs, G., Hayler, S. und Netherton, B., *Oracle® Solaris 10 System Virtualization Essentials*, Prentice Hall, 2010.
- [26] Zacker, C., *CompTIA® Network+® Exam N10-005 Training Kit*, Microsoft Press, 2012.
- [27] Aussel, J.-D. und Rousseau, L., pyscard project website, <http://pyscard.sourceforge.net/>, 2016.

Studienarbeit

Titel: Programmierung einer SIM-Authentifizierung
Subtitel: Irgend ein Untertitel
Autor: Marco Heumann
Hochschule: Duale Hochschule Baden-Württemberg Mannheim
Datum:
Bearbeitungszeitraum:
Studiengang: Angewandte Informatik
Matrikelnummer, Kurs: 4188528, TINF13AI-BI
Betreuer:
Gutachter: Prof. Dr. C. Bürgy

Abstract

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Studienarbeit

Title: Programmierung einer SIM-Authentifizierung
Subtitle: Irgend ein Untertitel
Author: Marco Heumann
University: Duale Hochschule Baden-Württemberg Mannheim
Date:
Time of Project:
Study Course: Angewandte Informatik
Student ID, Course: 4188528, TINF13AI-BI
Supervisor in the Company:
Reviewer: Prof. Dr. C. Bürgy

Abstract

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.