Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Dominique Immanuel Dauch

Implementierung eines Steganographie- und Kryptographie-Verfahrens auf einem FPGA mit Betriebssystemanbindung

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Dominique Immanuel Dauch

Implementierung eines Steganographie- und Kryptographie-Verfahrens auf einem FPGA mit Betriebssystemanbindung

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Studiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Robert Fitz
Zweitgutachter : Prof. Dr.-Ing. Detlef Jensen

Abgegeben am: 17.02.2015

Dominique Immanuel Dauch

Thema der Masterarbeit
Implementierung eines Steganographie- und Kryptographie-Verfahrens auf einem FPGA mit Betriebssystemanbindung

Stichworte
Kryptographie, Steganographie, Advanced Encryption Standard, One-Time-Pad-Verschlüsselung, FPGA, Linux, Zedboard

Kurzzusammenfassung
Diese Arbeit beschreibt die Entwicklung eines Systems, welches kryptographische und steganographische Methoden einsetzt, um einen sicheren Nachrichtenaustausch zwischen mehreren Teilnehmern über ein öffentliches Netz zu ermöglichen. Außerdem wird die Implementierung des Systems als eingebettetes System auf einem Chip (*SoC*) erläutert.

Dominique Immanuel Dauch

Title of the master thesis
Implementation of a steganographic and cryptography method on a FPGA with operating system connection

Keywords
Cryptography, steganography, Advanced Encryption Standard, One-Time-Pad-Encryption, FPGA, Linux, Zedboard

Abstract
This work describes the development of a system which uses cryptographic and steganographic methods to enable a secure exchange of messages between multiple parties over a public network. In addition, the implementation of the system as an embedded system on a chip (*SoC*) is described.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AES	**A**dvanced **E**ncryption **S**tandard.
AXI-Bus	**A**dvanced-**E**xtensible-**I**nterface-Bus
CPU	**C**entral **P**rocessing **U**nit
ECB	**E**lectronic **C**ode **B**ook
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSBL	**F**irst **S**tage **B**oot**l**oader
FSM	**F**inite **S**tate **M**achine
IP-Adresse	**I**nternet-**P**rotokoll-Adresse
IP-Core	**I**ntellectual-**P**roperty-Core
JPEG	**J**oint **P**hotographic **E**xperts **G**roup
LSB	**L**east **S**ignificant **B**it
LUT	**L**ook**u**p-**T**able
MUX	**Mu**ltiple**x**er
NSA	**N**ational **S**ecurity **A**gency
OLED	**O**rganic **L**ight **E**mitting **D**iode
PCBC	**P**ropagating **C**ipher **B**lock **C**haining
PNG	**P**ortable **N**etzwerk**g**rafik
POVs	**P**airs **o**f **V**alues
PS/PL	**P**rocessing **S**ystem/**P**rogrammable **L**ogic
SDK	**S**oftware **D**evelopment **K**it
SOC	**S**ystem **O**n **C**hip
SVZ	**S**chlüssel**v**erteilungs**z**entrale
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
USB	**U**niversal **S**erial **B**us
V4L	**V**ideo for(**4**) **L**inux
VHDL	**V**ery **H**igh Speed Integrated Circuit Hardware **D**escription **L**anguage
XML	E**x**tensible **M**arkup **L**anguage
XOR	E**x**clusive **Or**
XPS	**X**ilinx **P**latform **S**tudio

1 Einleitung

1.1 Motivation

Die weltweite Verbreitung des Internets ermöglicht einen schnellen und einfachen Weg Nachrichten untereinander auszutauschen. Dies liegt vor allem daran, dass das Internet ein öffentliches Netzwerk ist, zu dem ein Großteil der Menschen mit einer geringen technischen Ausstattung Zugriff hat. Diese grundlegende Eigenschaft prädestinierte das Internet zwar erst zu einem vielfach genutzten Medium zum Nachrichtenaustausch, birgt aber auch den Nachteil, dass eine Nachricht nicht nur für die Zielperson empfangbar ist. So kann jede Person, die den Datenstrom zum richtigen Zeitpunkt „mitliest", auf den Inhalt der Nachricht zugreifen. Dies mag für den Großteil der Nachrichten, die über das Internet versendet werden, belanglos sein, da sie keine schützenswerte Information enthalten. Soll das Internet zum Austausch von sensiblen Nachrichten genutzt werden, muss dieser Sachverhalt jedoch berücksichtigt werden.

Zum Schutz von sensiblen Nachrichten gibt es zwei grundlegende Vorgehensweisen. Die eine besteht darin, die Nachricht in den Daten einer anderen Nachricht (dem sogenannten Container) zu verbergen, sodass eine mitlauschende Drittperson (im Folgenden als „Angreifer" oder „Gegner" bezeichnet) nicht erkennt, dass eine Nachricht mit vertraulichen Informationen versendet wurde. Dieses Vorgehen wird als Steganographie bezeichnet. Der Nachteil dieser Methode ist, dass die Nachricht ihren Schutz verliert, sobald der Angreifer erfährt, dass eine versteckte Nachricht gesendet wird und wo sich die Nachricht in den Containerdaten befindet.

Die zweite Möglichkeit Nachrichten zu schützen besteht darin, den Inhalt der Nachricht durch eine Verschlüsselung für jeden unleserlich zu machen, der nicht über zusätzliche, geheime Informationen (beispielsweise einen Schlüssel) verfügt. Dies wird im Allgemeinen als Kryptographie bezeichnet. Diese Methode hat jedoch den Nachteil, dass verschlüsselte Nachrichten leicht als solche zu erkennen sind (siehe Kapitel 4). Erkennt also ein Gegner, dass eine Nachricht verschlüsselt ist, so wächst in der Regel auch das Interesse an dieser Nachricht, da die Vermutung nahe liegt, dass diese Nachricht einen sensiblen Inhalt transportiert. Die Sicherheit einer Verschlüsselung ist jedoch in der Regel davon abhängig, wieviel Aufwand betrieben wird, um diese zu

entschlüsseln (siehe Kapitel 2.1). Somit liefert auch diese Methode nur einen begrenzten Schutz.

Um die Nachteile beider Methoden zu minimieren, bietet es sich an, ein kombiniertes Verfahren aus Steganographie und Kryptographie zu nutzen. So werden die Nachrichten verschlüsselt, um den Inhalt zu schützen und anschließend in anderen Daten versteckt, um das Interesse eines Gegners an den Daten gering zu halten. Ein System, welches ein solch kombiniertes Verfahren nutzt um einen sicheren Datenaustausch von sensiblen Daten über das Internets zu ermöglichen, wird im Rahmen dieser Arbeit entwickelt und beschrieben.

1.2 Zielbeschreibung

Ziel der Arbeit ist es, ein autonomes eingebettetes System zu realisieren, das kryptographische und steganographische Methoden einsetzt, um einen sicheren digitalen Nachrichtenaustausch zwischen mehreren Teilnehmern zu ermöglichen. Als Nachricht können sowohl Text- als auch Binärdaten dienen. Um eine möglichst hohe Sicherheit zu erreichen werden die Nachrichten zunächst verschlüsselt und anschließend in einem Datenstrom von unauffälligen Daten versteckt. Hierfür werden Bilddaten verwendet, die kontinuierlich durch eine USB-Kamera erzeugt werden. Die kryptographischen und steganographische Methoden werden in Kapitel 3 näher erläutert.

Die kontinuierlich erzeugten Bilder werden den anderen Teilnehmern über einen HTTP- bzw. FTP-Server zu Verfügung gestellt. Die Teilnehmer verbinden sich als Client mit dem Server und tasten so die Bilddaten ab. Aus jedem abgetasteten Bild werden die relevanten Informationen extrahiert und mit dem persönlichen Schlüssel entschlüsselt. Erkennt das System in den Daten eine Nachricht, wird diese gespeichert und der Benutzer informiert. Dieser kann die Daten dann auf einen USB-Speicher kopieren oder sich direkt anzeigen lassen. Befindet sich keine Nachricht in den Daten, werden diese verworfen.

Da jedes System sowohl eine Server- als auch eine Client-Funktionalität implementiert hat, lässt sich damit eine komplexe vermaschte Netzwerktopologie aufbauen. Dabei bestimmen die Verteilungen der Serveradressen und die zugehörigen persönlichen Schlüssel unter den Teilnehmern die Verbindungen zwischen den einzelnen Knotenpunkten. Wenn jeder Teilnehmer mit jedem anderen Teilnehmer über einen eigenen Kanal verbunden ist, entsteht ein vollständig vermaschtes Netz.

Um eine unidirektionale Verbindung von Teilnehmer A zu Teilnehmer B zu erstellen, benötigt Teilnehmer A den persönlichen Schlüssel von Teilnehmer B und Teilnehmer B die Server-Adresse von Teilnehmer A. Soll eine bidirektionale Verbindung möglich sein, benötigen beide sowohl den Schlüssel als auch die Server-Adresse des jeweils anderen Teilnehmers.

Die Server- und Client-Funktionalität wird durch eine Linux-Distribution bereitgestellt. Da diese dafür eine Verbindung mit dem Internet benötigt, besteht die Gefahr, dass sich Angreifer beispielsweise durch offene Ports direkten Zugriff auf das System verschaffen. Um den Schaden dadurch gering zu halten, werden die unverschlüsselten Daten von einer separaten CPU verwaltet, die ohne Betriebssystem betrieben wird (*Bare-Metal-Betrieb*) und die keinen Zugriff auf das Internet hat. Die kryptographischen und steganographischen Methoden werden als Hardwarekomponente auf einem FPGA (Field Programmable Gate Array**)** realisiert, der beide CPU-Systeme miteinander verbindet. Somit hat die Linux-CPU nur Zugriff auf die vollständig verschlüsselten und bereits in den Bilddaten eingebetteten Daten des Nutzers. Auch die persönlichen Schlüssel der anderen Teilnehmer werden durch die *Bare-Metal-CPU* verwaltet. Ein Blockschaltbild mit Datenfluss zeigt Abbildung 1.

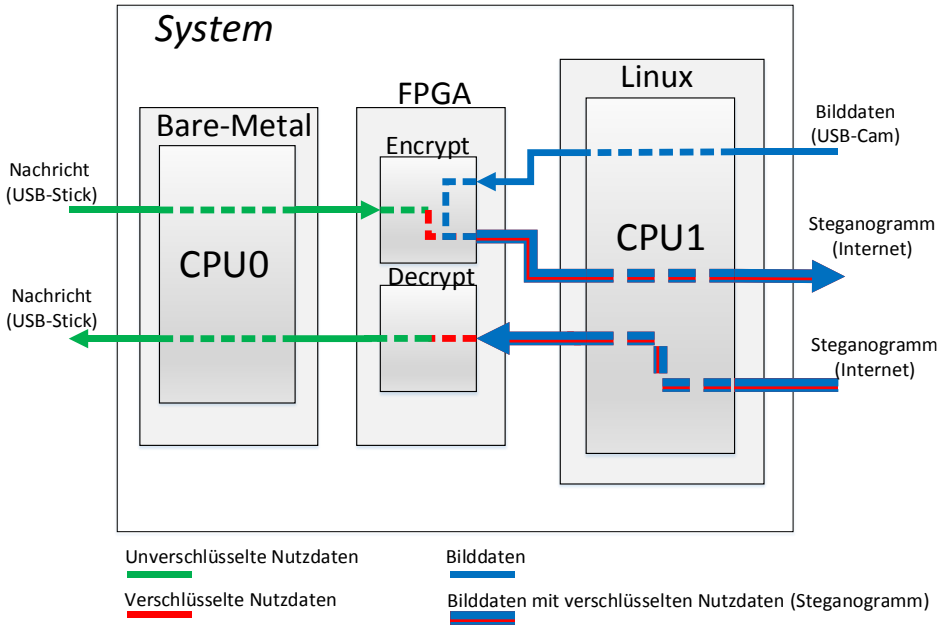


Abbildung 1 - Blockschaltbild mit Datenfluss

Eine detaillierte Systembeschreibung erfolgt in Kapitel 4.

2 Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die zum Verständnis der Arbeit benötigt werden. Dabei wird zwischen den beiden Wissenschaftszweigen Kryptologie (Kapitel 2.1) und Steganographie (Kapitel 2.2) unterschieden.

2.1 Kryptologie

Die Kryptologie befasst sich mit den wissenschaftlichen Aspekten der Kryptographie und der Kryptoanalyse. Somit kann sie als Oberbegriff dieser beiden Teilgebiete betrachtet werden. Die Abgrenzung zwischen den beiden Teilwissenschaften, sowie die Begriffsdefinitionen selbst, stammen von dem russisch-amerikanischen Kryptologen William Friedman und wurden am Ende des ersten Weltkrieges eingeführt [1]. Nach Friedman beschäftigt sich die Kryptographie hauptsächlich mit der Entwicklung und Anwendung von verschiedenen kryptographischen Verfahren, um den Inhalt von vertraulichen Nachrichten vor unberechtigten Personen zu schützen. Die Kryptoanalyse hingegen ist darauf konzentriert, die Schwachpunkte dieser Verfahren zu finden, um so den auferlegten Schutz der Nachrichten wieder zu entfernen. Somit ist die Zielsetzung beider Teilgebiete zwar gegensätzlich, doch zur Entwicklung von kryptographischen Systemen werden Kenntnisse aus beiden Wissenschaften benötigt.

2.1.1 Kryptographie

Die Kryptographie bezeichnet die Lehre von der Verschlüsselung von Nachrichten [2]. Die Anfänge dieser Lehre gehen bis in die Antike zurück. Das älteste bekannte militärische Verschlüsselungsverfahren trägt die Bezeichnung „*Skytale*“ (grich.: „Stock“, „Stab“) und wurde vor mehr als 2500 Jahren von den Spartanern eigesetzt, um geheime Nachrichten zu übermitteln. So wurde beispielweise der spartanische General Lysander während des Peloponnesischen Krieges (431v Chr. – 404 v. Chr.) mit einer mit *Skytale* chiffrierten Botschaft vor dem Angriff der Perser gewarnt, worauf dieser vereitelt werden konnte [3].

Bei dem *Skytale*-Verfahren wird ein Streifen aus Leder (oder ähnliches) um einen Holzstab mit einem festgelegten Durchmesser gewickelt und anschließend beschriftet (Abbildung 2). Der Empfänger der Nachricht benötigt nun einen Stab mit annährend gleichem Durchmesser, um die Nachricht zu entschlüsseln. Stimmt der Durchmesser nicht überein, ergibt sich nur eine scheinbar willkürliche Buchstabenkombination. Der Durchmesser des Stabes entspricht also bei diesem Verfahren dem geheimen Schlüssel.

Abbildung 2 – Skytale

Trotz ihrer langen Geschichte ist die Kryptographie erst in jüngerer Zeit Gegenstand der öffentlichen Forschung. Bis zum Ende des zweiten Weltkriegs befasste sich fast ausschließlich der militärische Bereich mit Verschlüsselungsverfahren, sodass wissenschaftliche Quellen strengster Geheimhaltung unterlagen. Erst danach und besonders durch die Verbreitung der digitalen Kommunikation rückte die Kryptographie in den Fokus des öffentlichen Interesses. Diese Entwicklung wurde jedoch von einigen Stellen kritisch gesehen. So unternahm beispielsweise die *National Security Agency* (NSA) in den 80ger Jahren mehrere Versuche, die Verbreitung kryptographischen Materials zu kontrollieren [4]. Dies gelang zum einen dadurch, da kryptographische Methoden dem Waffenkontrollgesetzt unterliegen, wodurch eine Exportbeschränkung ermöglicht wurde. Erst in den letzten Jahren wurden, unter anderem durch Druck aus der Wirtschaft, die Exportrichtlinien in den USA für Verschlüsselungsverfahren gelockert [5].

Um Nachrichten zu verschlüsseln gibt es zwei grundlegende Herangehensweisen. Die erste wird auch als „*Security through Obscurity*“ („Sicherheit durch Obskurität“) bezeichnet. Die Methode beruht darauf, dass der zur Verschlüsslung genutzte Algorithmus geheim bleibt. Somit ist ein zusätzliches Geheimnis (beispielsweise ein geheimer Schlüssel) nicht mehr zwingend nötig. Dieses Vorgehen verlor jedoch an Bedeutung, als 1949 der amerikanische Mathematiker Claude Shannon den Artikel „*Communication Theory of Secrecy Systems*“ veröffentlichte [6]. Diese Arbeit gilt als eine der ersten mathematischen Auseinandersetzungen mit dem Thema Kryptographie und

legte den Grundstein für eine offene wissenschaftliche Diskussion über verschiedene Verschlüsselungsverfahren. Shannon gilt somit auch als Mitbegründer der modernen Kryptographie [7].

Die öffentlich diskutierten Herangehensweisen bilden das Gegenstück zu „*Security through Obscurity*". Hierbei wird Sicherheit durch Transparenz erlangt, da die benutzen Algorithmen der Öffentlichkeit zur Überprüfung bereitgestellt werden. Das Geheimnis, das für jede Art von Verschlüsslung benötigt wird, besteht bei solchen Verfahren lediglich aus einem oder mehreren Schlüsseln, die zum Ver- und Entschlüsseln der Nachrichten verwendet werden. Dies wird auch als *Kerckhoffs'sches Prinzip* bezeichnet und bildet das zweite der sechs Grundsätze zur Konstruktion eines sicheren Verschlüsselungsverfahren, die 1883 von Kerckhoff in „*La cryptographie militaire*" eingeführt wurden [8][1]:

1) « Le système doit être matériellement, sinon mathématiquement, indéchiffrable »
   (Das System sollte weder physikalisch noch mathematisch entschlüsselbar sein.)
2) « Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi »
   (Es sollte keine Geheimhaltung erfordern und sollte auch vom Feind gestohlen werden dürfen.)
3) « La clef doit pouvoir en être communiquée et retenue sans le secours de notes écrites, et être changée ou modifiée au gré des correspondants »
   (Der Schlüssel sollte leicht zu kommunizieren sein und er sollte nicht notiert werden müssen. Außerdem sollte er vom Anwender änderbar sein.)
4) « Il faut qu'il soit applicable à la correspondance télégraphique »
   (Das System sollte mit einer telegraphischen Kommunikation kompatibel sein.)
5) « Il faut qu'il soit portatif, et que son maniement ou son fonctionnement n'exige pas le concours de plusieurs personnes «
   (Es sollte tragbar sein und seine Verwendung nicht die Unterstützung mehrerer Personen erfordern.)
6) « Enfin, il est nécessaire, vu les circonstances qui en commandent l'application, que le système soit d'un usage facile, ne demandant ni tension d'esprit, ni la connaissance d'une longue série de règles à observer »
   (Das System sollte leicht zu bedienen sein.)

Auch modernere Quellen, wie das *National Institute of Standards and Technology*, raten von einem Verfahren das alleine auf dem Prinzip „*Security through Obscurity*" beruht ab [9].

[1] Die Übersetzung aus dem Französischen erfolgte sinngemäß mit Hilfe des Online-Übersetzers von Google

Die Hauptgründe, die gegen einen geheimen Algorithmus sprechen, sind nach [10]:

- Es ist viel schwieriger einen Algorithmus geheim zu halten, als einen Schlüssel. Dies gilt vor allem für öffentlich zugängliche Systeme, da in Hard- oder Software implementierte Algorithmen durch das sogenannten "*Reverse Engineering*"[2] rekonstruiert werden können.
- Fehler in öffentlichen Algorithmen können im Allgemeinen leichter entdeckt werden, wenn sich möglichst viele Fachleute damit befassen.
- Es wird ein hohes Maß an Vertrauen zu dem Entwickler des Verfahren benötigt, da nicht überprüft werden kann, ob beispielsweise eine „*Hintertür*" implementiert wurde.

Aber auch das Nutzen veröffentlichter Algorithmen birgt Nachteile. So ist es einem Angreifer auch möglich, den verwendeten Verschlüsselungsalgorithmus zu analysieren um gegebenenfalls Schwachstellen zu finden, bevor diese öffentlich bekannt werden. Außerdem lassen sich für standardisierte Verschlüsselungsalgorithmen auch standardisierte Angriffsmethoden entwickeln, um diese zu brechen. Höchste Sicherheit verspricht also auch hier ein kombiniertes Verfahren, so wie es im Rahmen dieser Arbeit entwickelt wird.

Den allgemeinen Ablauf einer verschlüsselten Nachrichtenübertragung zeigt Abbildung 3.

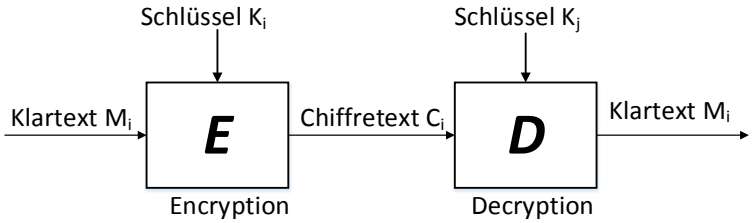


Abbildung 3 - Blockschaltbild einer verschlüsselten Nachrichtenübertragung

Eine Nachricht besteht dabei aus einem Klartext M_i aus dem Nachrichtenraum M (*Message*), der durch eine Verschlüsselungsfunktion E (*Encryption*) in einen Chiffretext C_i aus dem Chiffreraum C überführt wird. Aus diesem Chiffretext C_i generiert die Entschlüsselungsfunktion D (*Decryption*) wieder eine Nachricht mit dem ursprünglichen Klartext M_i.

[2] Reverse Engineering bezeichnet den Vorgang, aus einem System die Konstruktionselemente zu extrahieren, um so beispielsweise die Funktionsweise nachzuempfinden [20].

Es gilt also:

$$E(M_i) = C_i \quad (1)$$

$$D(C_i) = M_i \quad (2)$$

Und somit auch:

$$D\big(E(M_i)\big) = M_i \quad (3)$$

Werden Schlüsselpaare $K_{i,j}$ aus dem Schlüsselraum K bei der Ver- bzw. Entschlüsselung verwendet, sind die entsprechenden Funktionen E und D von K_i beziehungsweise K_j abhängig:

$$E(K_i, M_i) = C_i \quad (4)$$

$$D\big(K_j, C_i\big) = M_i \quad (5)$$

$$D\left(K_j, E(K_i, M_i)\right) = M_i \quad (6)$$

Sind die Schlüsselpaare zur Ent- und Verschlüsselung identisch ($K_i = K_j$), wird dies als symmetrisches Verfahren bezeichnet. In diesem Fall muss der gemeinsame Schlüssel über einen sicheren Kanal ausgetauscht werden, da die Ver- und Entschlüsselung in der Regel von zwei unterschiedlichen Personen durchgeführt wird. Werden zwei verschiedene Schlüssel verwendet ($K_i \neq K_j$) handelt es sich um ein asymmetrisches Verfahren. Dies wird bei Algorithmen mit öffentlichen Schlüsseln (engl.: „*public key algorithms*“) angewandt. Zur Chiffrierung wird dabei ein öffentlich zugänglicher Schlüssel verwendet, der ohne weitere Sicherheitsmaßnahmen verbreitet werden kann. Die Dechiffrierung erfolgt dann über einen geheimen, privaten Schlüssel. Anwendung finden solche Verfahren beispielweise bei Authentifizierungsmaßnahmen im Internet („*Public-Key-Authentifizierung*“). Da in dieser Arbeit keine asymmetrischen Verfahren zum Einsatz kommen, werden diese im Folgenden nicht weiter betrachtet.

Zur weiteren Klassifizierung kryptographischer Verfahren kann in einer ersten Stufe zwischen *Codesystemen* und *Kryptosystemen* unterschieden werden. Codesysteme weisen jeder semantischen Einheit einer Sprache, also Wörtern, Phrasen oder ganzen Sätzen, über eine Zuordnungstabelle (*Code Buch*) einen festen Chiffretext zu. Ein solches Verfahren eignet sich aber nur, wenn der Nachrichtenraum M begrenzt ist, das heißt nur eine vordefinierte Anzahl von Wörtern oder Sätzen verwendet wird. Kryptosysteme hingegen funktionieren unabhängig von dem Inhalt einer Nachricht und können somit für beliebige Arten von Daten verwendet werden.

Die Menge der Kryptosysteme kann wiederum in die zwei Teilmengen, Blockchiffren und Stromchiffren, unterteilt werden.

Eine Blockchiffre verarbeitet nur Nachrichtenblöcke mit fester Länge und bildet diese über die Verschlüsselungsfunktion E in ebenso lange Chiffreblöcke ab. Führen immer gleiche Klartextblöcke zu immer gleichen Chiffreblöcken, wird die zugrunde liegende Blockchiffre als deterministisch bezeichnen. Ist dies nicht der Fall, handelt es sich um eine indeterministische Blockchiffre. Ein prominentes Beispiel aus der Menge der Blockchiffren ist der *Advanced Encryption Standard* (*AES*), ein Verfahren, welches in Kapitel 3 näher erläutert wird.

Im Gegensatz zur Blockchiffre unterteilt eine Stromchiffre die Nachricht in elementare Einheiten des zugrunde liegenden Alphabets A. Somit können Nachrichten beliebiger Länge verarbeitet werden. Bei einer digitalen Nachricht entspricht eine solche Einheit häufig einem Bit $(A = \{0, 1\})$. Die einzelnen Einheiten werden dann mit einer gleichlangen Einheit aus einem Schlüsselstrom verknüpft. Der Schlüsselstrom wird dabei meist aus einer pseudozufälligen Zeichenfolge generiert, die aus einem Initialschlüssel abgeleitet wird [11]. Werden bereits verschlüsselte Teile der Nachricht in die Schlüsselstromgenerierung mit einbezogen, handelt es sich um synchrone Stromchiffren. Werden während der Übertragung Teile des Chiffretextes verfälscht oder entfernt, kann dieser vom Empfänger nicht mehr korrekt entschlüsselt werden, und Sender und Empfänger müssen sich neu synchronisieren. Synchrone Stromchiffren eignen sich auch zur Authentifikation von Nachrichten, da Manipulationen am Chiffretext erkannt werden können.

Eine Blockchiffre mit der Blocklänge B kann in eine Stromchiffre überführt werden, wenn für diese das zugrunde liegende Alphabet $A = \{0, \dots, 2^B - 1\}$ gewählt wird. Somit besteht die elementare Einheit des Alphabets aus einem Block der Länge B. Dies erlaubt auch die Verschlüsselung von längeren Nachrichten mit Hilfe eines Blockchiffre-Algorithmus. Das in dieser Arbeit verwendete Verfahren zur Überführung von Blockchiffren in Stromchiffren (auch Betriebsarten von Blockchiffren genannt) wird in Kapitel 3 vorgestellt.

2.1.2 Kryptoanalyse

Während das Ziel der Kryptographie darin liegt, den Inhalt von Nachrichten für jeden zu verbergen, der nicht über das entsprechende Geheimnis verfügt, sucht der Kryptoanalytiker nach Möglichkeiten den Inhalt von verschlüsselten Nachrichten auch ohne dieses Geheimnis zu offenbaren. Er kann somit als natürlicher Gegenspieler des Kryptographen bezeichnet werden.

Die Durchführung einer Kryptoanalyse wird als Angriff bezeichnet. Im vorherigen Kapitel wurde bereits darauf hingewiesen, dass die Sicherheit eines Verschlüsselungsverfahrens im Wesentlichen nur von dem verwendeten Schlüssel abhängen darf (*Kerckhoffs'sches Prinzip*). Kerckhoff geht in seiner Arbeit davon aus, dass der verwendete Algorithmus und dessen Implementierung dem Kryptoanalytiker bekannt ist, da dieser, besonders bei weit verbreiteten Systemen, nur schwer geheim gehalten werden kann. Somit kann die Ver- und Entschlüsselungsfunktion $E(\mathrm{K}, M_1)$ bzw. $D(\mathrm{K}, C_1)$ bei allen nachfolgend beschriebenen Angriffsarten als gegeben vorausgesetzt werden. Eine Auswahl verschiedener Angriffsarten zeigt Tabelle 1. Die Beschreibungen und Begriffsbezeichnungen beziehen sich dabei auf [4].

Name	**Beschreibung**	**Gegeben**	**Gesucht**
Ciphertext-only-Angriff	Der Angreifer verfügt über Chiffretexte mehrerer Nachrichten. Die Aufgabe besteht darin, den Klartext der Nachrichten herzustellen oder den verwendeten Schlüssel abzuleiten, um auch nachfolgende Nachrichten entschlüsseln zu können.	$C_1 = E(\mathrm{K}, M_1)$, … , $C_i = E(\mathrm{K}, M_i)$	$M_1, \dots, M_i$ Bzw. K oder ein Algorithmus um M_{i+1} aus $C_{i+1} = E(\mathrm{K}, M_{i+1})$ bestimmen zu können.
Known-plaintext-Angriff	Der Angreifer verfügt neben den Chiffretexten noch über die dazu gehörigen Klartexte. Das Ziel besteht darin, den verwendeten Schlüssel K abzuleiten.	$M_1, C_1 = E(\mathrm{K}, M_1)$, … , $M_i, C_i = E(\mathrm{K}, M_i)$	K oder ein Algorithmus um M_{i+1} aus $C_{i+1} = E(\mathrm{K}, M_{i+1})$ bestimmen zu können.
Chosen-plaintext-Angriff	Der Angriff ähnelt dem *Known-plaintext-Angriff*, mit dem Unterschied, dass der Angreifer den Klartext selbst festlegen kann.	$M_1, C_1 = E(\mathrm{K}, M_1)$, … , $M_i, C_i = E(\mathrm{K}, M_i)$, wobei $M_1, \dots, M_i$ frei gewählt werden kann.	K oder ein Algorithmus um M_{i+1} aus $C_{i+1} = E(\mathrm{K}, M_{i+1})$ bestimmen zu können.
Chosen-ciphertext-Angriff	Der Angreifer kann verschiedene Chiffretexte zum Entschlüsseln auswählen und hat Zugriff auf die zugehörigen Klartexte. Das Ziel ist es, den passenden Schlüssel **K** zu bestimmen.	$C_1, M_1 = D(\mathrm{K}, C_1)$, … , $C_i, M_i = D(\mathrm{K}, C_i)$ wobei $C_1, \dots, C_i$ frei gewählt werden kann.	K

Tabelle 1 - Angriffsarten der Kryptoanalyse

Der *Ciphertext-only-Angriff* gehört zu den am weitesten verbreiteten Angriffen, da die benötigten Informationen, die Chiffretexte, in der Regel leicht zu beschaffen sind (beispielsweise durch das „belauschen“ des Übertragungskanals). Allerding ist der Aufwand, mit diesen Informationen die Verschlüsslung zu brechen, relativ hoch. Mehr Erfolg versprechen die *known-plaintext-,* bzw. *chosen-plaintext-Angriffe.* Die benötigten Klartexte sind jedoch meistens nicht so einfach zu beschaffen, da diese in der Regel besonders geschützt sind. Eine Möglichkeit wird in [4] beschrieben. Dabei übergibt der Angreifer eine Nachricht an den Botschafter eines Landes. Sendet dieser die Nachricht verschlüsselt an sein Heimatland, damit sie dort begutachtet werden kann, kommt der Angreifer an den passenden Chiffretext, wenn er den Übertragungskanal zum richtigen Zeitpunkt belauscht. Gut geeignet sind dafür Nachrichten, bei denen bestimme Phrasen oder Schlüsselwörter besonders gehäuft vorkommen, wie es beispielsweise bei Quellcodes oder auch ausführbaren Dateien der Fall ist.

2.1.3 Sicherheit kryptographischer Systeme

Als uneingeschränkt sicher gilt ein Algorithmus, wenn dieser selbst dann nicht zu brechen ist, wenn Chiffretexte in beliebigen Umfang und uneingeschränkte Ressourcen vorhanden sind. Dieses Kriterium erfüllt zurzeit nur eine *One-Time-Pad-Verschlüsselung*, die in Kapitel 3 beschrieben wird. Alle anderen Kryptosysteme sind theoretisch mit einem *Ciphertext-only-Angriff* zu brechen. Die einfachste Art eines solchen Angriffs besteht darin, alle möglichen Schlüssel aus dem Schlüsselraum auszuprobieren und bei jedem Versuch zu überprüfen, ob der resultierende Klartext einen Sinn ergibt. Dies wird auch als *Brute-Force-Angriff* bezeichnet. Die Erfolgsaussicht eines solchen Angriffs ist in der Praxis unter anderem stark von der genutzten Schlüssellänge abhängig. Bei einer Schlüssellänge von 8-Bit besteht der Schlüsselraum aus $2^8 = 256$ möglichen Schlüsseln. Das bedeutet, dass nach maximal 256 Versuchen die Verschlüsselung gebrochen ist. Bei einer Schlüssellänge von 128-Bit liegt die Anzahl der möglichen Schlüssel jedoch schon bei $2^{128} = 3{,}4 \cdot 10^{38}$. Ein Supercomputer, der eine Millionen Schlüssel pro Sekunde ausprobieren kann, könnte demnach bis zu $\sim 10^{25}$ Jahre benötigen, um ein positives Ergebnis zu erzielen. Die Größenordnung dieser Zeitspanne wird deutlich, wenn zum Vergleich das Alter des Universums herangezogen wird, welches auf $\sim 10^{10}$ Jahre geschätzt wird [4]. Allerdings muss dabei berücksichtigt werden, dass durch zukünftige Technologien, wie beispielsweise die des Quantencomputers, die Effizienz solcher Angriffe deutlich gesteigert werden könnte.

Bei der Betrachtung der Sicherheit von kryptographischen Systemen wird zwischen theoretischer und praktischer Sicherheit unterschieden.

Theoretische Sicherheit

Um die theoretische Sicherheit eines kryptographischen Systems beurteilen zu können werden mathematische Modelle benötigt, mit deren Hilfe ein System analytisch betrachtet werden kann.

Die mathematischen Grundlagen, die zur Auseinandersetzung mit der Kryptographie bzw. Kryptoanalyse genutzt werden können, stammen im Wesentlichen aus der modernen Informationstheorie, die 1948 ebenfalls von E. C. Shannon begründet wurde [12]. Die Analogie zwischen der Informationstheorie und der Kryptoanalyse besteht darin, dass in beiden Fällen versucht wird, eine Nachricht, die über einen gestörten Kanal gesendet und somit verfälscht wird, zu rekonstruieren. Bei der Kryptographie sind diese Störungen jedoch gewollt und nicht die Folge von physikalischen Prozessen, wie es bei der herkömmlichen Nachrichtenübertragung der Fall ist. Die Störungen werden gezielt durch die Verschlüsselungsfunktion $E(K_i, M_i)$ erzeugt und müssen möglichst vollständig durch die Entschlüsselungsfunktion $D(K_i, C_i)$ und dem passenden Schlüssel K_i beseitigt werden können. Diese Möglichkeit besitzt der Kryptoanalytiker in der Regel nicht, da ihm der Schlüssel K_i unbekannt ist. Somit bleiben ihm nur die Informationen aus dem gestörten Kanal, also der Chiffretext C_i, um den Klartext aus der Nachricht zu rekonstruieren.

Jeder Klartext M_i aus der Menge aller möglichen Klartexte M besitzt eine gewisse Wahrscheinlichkeit $p_i = P(M_i)$, mit der dieser vom Sender ausgewählt und versendet wird. Diese Wahrscheinlichkeit wird als a priori Wahrscheinlichkeit bezeichnet, wobei gilt:

$$\sum_{i=1}^{|M|} p_i = 1 \qquad (7)$$

Der Empfänger (in diesem Fall der Kryptoanalytiker oder auch Angreifer) empfängt draus ein Chiffretext C_j aus der Menge aller möglichen Chiffretexte C. Die Wahrscheinlichkeit, dass dabei ein bestimmter Chiffretext C_j empfangen wird, wird mit $P(C_j)$ angegeben. Die gemeinsame Wahrscheinlichkeit beider Ereignisse wird durch den Ausdruck $P(M_i, C_j)$ dargestellt. Sind beide Ereignisse stochastisch unabhängig, berechnet sich die gemeinsame Wahrscheinlichkeit aus dem Produkt der Einzelwahrscheinlichkeiten:

$$P(M_i, C_j) = P(M_i) \cdot P(C_j) \qquad (8)$$

Neben der gemeinsamen Wahrscheinlichkeit gibt es noch die bedingte Wahrscheinlichkeit $P(M_i|C_j)$, die angibt, mit welcher Wahrscheinlichkeit M_i gesendet wurde, wenn C_j empfangen wird. Diese Wahrscheinlichkeit wird als a posteriori Wahrscheinlichkeit bezeichnet. Es gilt:

$$P(M_i|C_j) = \frac{P(M_i, C_j)}{P(C_j)} \qquad (9)$$

Der umgekehrte Fall, also die a posteriori Wahrscheinlichkeit, dass C_j empfangen wird, wenn M_i gesendet wurde, wird mit $P(C_j|M_i)$ angegeben. Sie lässt sich wie folgt berechnen:

$$P(C_j|M_i) = \frac{P(M_i, C_j)}{P(M_i)} \qquad (10)$$

Aus (9) und (10) lässt sich zwischen $P(M_i|C_j)$ und $P(C_j|M_i)$ folgender Zusammenhang herstellen:

$$P(M_i|C_j) = P(C_j|M_i) \cdot \frac{\mathrm{P(M_i)}}{\mathrm{P(C_j)}} \qquad (11)$$

Aus der Wahrscheinlichkeit $P(M_i)$, dass also eine bestimmte Nachricht M_i aus M gesendet wird, lässt sich deren Informationsgehalt $I(M_i)$ bestimmen:

$$I(M_i) = log_2\left(\frac{1}{P(M_i)}\right) = -log_2(P(M_i)) \qquad (12)$$

Demnach ist der Informationsgehalt einer Nachricht umso größer, je geringer deren Auftrittswahrscheinlichkeit ist. Durch die Verwendung des Logarithmus wird außerdem die Forderung erfüllt, dass sich der Informationsgehalt zweier unabhängig voneinander ausgewählten Nachrichten M_i und M_j addieren soll, da gilt:

$$log_2\left(\frac{1}{P(M_i) * P(M_j)}\right) = log_2\left(\frac{1}{P(M_i)}\right) + log_2\left(\frac{1}{P(M_j)}\right)$$

Wird der Informationsgehalt jeder möglichen Nachricht aufsummiert und mit der jeweiligen Auftrittswahrscheinlichkeit gewichtet, ergibt sich daraus der mittlere Informationsgehalt der Quelle. Dieser wird auch als Entropie $H(M)$ der Quelle bezeichnet und lässt sich folgendermaßen berechnen:

$$H(M) = -\sum_{i=1}^{|M|} P(M_i) \cdot log_2(P(M_i)) \qquad (13)$$

Die Entropie ist auch ein Maß für die Unsicherheit einer Quelle. Soll eine Nachricht beispielweise Informationen über das Geschlecht einer Person enthalten, so liegt die Unsicherheit bei 1, da dafür nur „männlich“ oder „weiblich“ in Frage kommt.

Berechnen ließe sich dies wie folgt:

$$H(M) = -(0{,}5 \cdot log_2(0{,}5) + 0{,}5 \cdot log_2(0{,}5) = 1$$

Die Entropie einer Nachricht kann sich ändern, wenn ein empfangener Chiffretext Rückschlüsse auf den Inhalt der gesendeten Nachricht zulässt. Dies wird als bedingte Entropie $H(M|C)$ Bezeichnet:

$$H(M|C) = -\sum_{i=1}^{|M|}\sum_{j=1}^{|C|} P(C_j) \cdot P(M_i|C_j) \cdot log_2\left(P(M_i|C_j)\right) \qquad (14)$$
$$= -\sum_{i=1}^{|M|}\sum_{j=1}^{|C|} P(M_i, C_j) \cdot log_2\left(P(M_i|C_j)\right)$$

Die bedingte Entropie misst die Unsicherheit einer Quelle, nach dem eine Nachricht empfangen wurde. Aus der Differenz der Unsicherheiten vor und nach dem Empfang einer Nachricht, kann der Informationsgehalt bestimmt werden, der von dem Kanal übertragen wird. Dieser wird als wechselseitige Information $I(M|C)$ definiert:

$$I(M|C) = H(M) - H(M|C) \qquad (15)$$

Ist die wechselseitige Information $I(M|C) = 0$, so sind nach Shannon die Kriterien für ein absolut sicheres Kryptosystem erfüllt [12]. In diesem Fall entspricht die a posteriori der a priori Wahrscheinlichkeit:

$$P(M_i|C_j) = P(M_i) \qquad (16)$$

Der Gegner kann aus dem Chiffretext keinerlei Rückschlüsse auf den Inhalt der Nachricht ziehen.

Die absolute Sicherheit von Kryptosystemen soll nun anhand zweier Beispiele untersucht werden[3].

Beispiel 1

Als Menge aller Klartexte werden alle Bitfolgen der Länge drei betrachtet, es gilt also:

$$M = \{000,001,010, \dots ,111\}$$

Das gleiche gilt für die Menge aller möglichen Schlüssel:

$$K = \{000,001,010, \dots ,111\}$$

Die Verschlüsselungsfunktion $E(K, M)$ verknüpft den Klartext mit dem Schlüssel durch eine XOR-Operation:

$$E(K_i, M_i) = K_i \oplus M_i$$

Für das Beispiel wird angenommen, dass die Wahrscheinlichkeit, dass ein bestimmter Klartext, bzw. Schlüssel ausgewählt wird, für alle Klartexte, bzw. Schlüssel gleich ist. Es gilt also:

$$P(M_i) = \frac{1}{|M|} = \frac{1}{2^3} = \frac{1}{8}$$

Und

$$P(K_i) = \frac{1}{|K|} = \frac{1}{2^3} = \frac{1}{8}$$

Durch die XOR Operation gibt es für jedes Klartext/Chiffretext-Paar $(M_i|C_j)$ einen Schlüssel K_i der M_i in C_j überführt.

[3] Die beiden nachfolgenden Beispielwerte wurden aus [1] übernommen

Daraus folgt:

$$P(M_i|C_j) = P(K_i) = \frac{1}{8}$$

Außerdem kann ein bestimmter Klartext M_i mit dem passenden Schlüssel K_i in jeden beliebigen Chiffretext (gleicher Länge) überführt werden, sodass sich für die Menge aller möglichen Chiffretexte C ebenfalls eine Gleichverteilung ergibt:

$$P(C_i) = \frac{1}{|C|} = \frac{1}{2^3} = \frac{1}{8}$$

Aus diesen Angaben kann nun mit (15) die wechselseitige Information $I(M|C)$ berechnet werden:

$$I(M|C) = -\sum_{i=1}^{|M|} P(M_i) \cdot log_2(P(M_i)) + \sum_{i=1}^{|M|}\sum_{j=1}^{|C|} P(C_j) \cdot P(M_i|C_j) \cdot log_2\left(P(M_i|C_j)\right) =$$

$$= -\sum_{i=1}^{8} \frac{1}{8} \cdot log_2\left(\frac{1}{8}\right) + \sum_{i=1}^{8}\sum_{j=1}^{8} \frac{1}{8} \cdot \frac{1}{8} \cdot log_2\left(\frac{1}{8}\right)$$

$$= -8 \cdot \left(\frac{1}{8} \cdot (-3)\right) + 64 \cdot \left(\frac{1}{64} \cdot (-3)\right) = 0$$

Somit ist nach Shannon für dieses Kryptosystem absolute Sicherheit gewährleistet.

Beispiel 2

In diesem Beispiel wird die gleiche Klartextmenge wie im vorherigen Beispiel verwendet:

$$M = \{000,001,010, \dots,111\}$$

Die Menge aller Schlüssel wird diesmal jedoch auf drei beschränkt:

$$K = \{010,101,111\}$$

Somit können aus einem bestimmten Klartext M_i durch die Verschlüsselungsfunktion nur drei verschiedene Chiffretexte erzeugt werden, beziehungsweise ein bestimmter Chiffretext C_j kann nur aus drei verschiedenen Klartexten hervorgegangen sein. Daraus folgt für die bedingte Wahrscheinlichkeit $P(M_i|C_j)$:

$$P(M_i|C_j) = P(K_i) = \frac{1}{3}$$

sofern einen Schlüssel K_i existiert, der M_i in C_j überführt ($C_j = E(K_i, M_i)$). Gibt es diesen nicht, so gilt:

$$P(M_i|C_j) = 0$$

Für dieses Beispiel errechnet sich die wechselseitige Information $I(M|C)$ wie folgt:

$$I(M|C) = -\sum_{i=1}^{|M|} P(M_i) \cdot log_2(P(M_i)) + \sum_{i=1}^{|M|}\sum_{j=1}^{|C|} P(C_j) \cdot P(M_i|C_j) \cdot log_2\left(P(M_i|C_j)\right) =$$

$$= -\sum_{i=1}^{8} \frac{1}{8} \cdot log_2\left(\frac{1}{8}\right) + \sum_{i=1}^{8}\sum_{j=1}^{3} \frac{1}{8} \cdot \frac{1}{3} \cdot log_2\left(\frac{1}{3}\right)$$

$$= -8 \cdot \left(\frac{1}{8} \cdot (-3)\right) + 24 \cdot \left(\frac{1}{24} \cdot (-1{,}58)\right) = 3 - 1{,}58 = 1{,}42$$

Da $I(M|C) = 1{,}42 \neq 0$ ist, folgt daraus, dass es sich hierbei um kein absolut sicheres Kryptosystem handelt.

Auf Grund der Ergebnisse dieser beiden Beispiele lässt sich vermuten, dass in einem absolut sicheren System der Schlüsselraum K mindesten gleich groß dem Nachrichtenraum M sein muss:

$$|K| \geq |M| \qquad (17)$$

Ein mathematischer Beweis, der diese Vermutung bestätigt, erfolgt in [2]. Demnach müssen die Schlüssel außerdem stochastisch unabhängig voneinander gewählt werden und dürfen keine feste Wiederholungsfrequenz aufweisen, wie später auch in Beispiel 3 gezeigt wird.

Die Erfüllung dieser Vorrausetzung stellt in der Praxis eine große Herausforderung dar. Zumindest bei symmetrischen Verfahren müssen die Schlüssel über einen sicheren Kanal zwischen den Teilnehmern ausgetauscht werden. Da für ein absolut sicheres System die Schlüssellänge jedoch gleich der Nachrichtenlänge sein muss, und jeder Schlüssel pro Nachricht nur einmal verwendet werden darf, ist dies mit einem relativ hohen Aufwand verbunden und daher nur für kleine Datenmengen praktikabel.

Aus diesem Grund finden absolut sichere Kryptosysteme in der Praxis nur wenig Verwendung. Hier wird lediglich versucht, die Sicherheit des Systems so weit zu erhöhen, wie es für den Inhalt der Nachricht notwendig ist. Dies wird auch als praktische Sicherheit bezeichnet. Die Aspekte der praktischen Sicherheit werden im nachfolgenden Abschnitt erläutert.

Praktische Sicherheit

Wie im vorherigen Abschnitt erläutert, findet der Anspruch auf absolute Sicherheit in der Praxis nur wenig Bedeutung, da er nur mit großem Aufwand erfüllt werden kann. Stattdessen wird nur ein so hohes Maß an Sicherheit gefordert, wie es für den Inhalt der Nachricht angemessen erscheint. Somit wird in der Praxis in der Regel zwar keine absolute, aber dennoch eine wahrscheinliche Sicherheit erreicht.

Als wahrscheinlich sicher gilt ein Verfahren nach [4] wenn folgende Kriterien erfüllt sind:

- Der Geldaufwand, der zum Brechen des Algorithmus nötig ist, übersteigt den Wert der verschlüsselten Nachricht.
- Der Zeitaufwand, der zum Brechen des Algorithmus nötig ist, übersteigt die Zeitspanne, in der die Nachricht geheim bleiben muss.
- Die Datenmenge, die zum Brechen des Algorithmus nötig ist, übersteigt das Datenvolumen, welches mit einem bestimmten Schlüssel geschützt ist.

Die Angriffspunkte für nicht absolut sichere Systeme sind unter anderem statistische Auffälligkeiten in den Klartextblöcken, die sich in den zugehörigen Chiffretextblöcken widerspiegeln. So sind, auf Grund der Redundanz natürlicher Sprachen, die Buchstaben eines Klartextes in der Regel nicht gleichverteilt. Die Verteilung der Buchstaben des deutschen Alphabets ist in Abbildung 4 grafisch dargestellt. Die Werte wurden aus [13] übernommen.

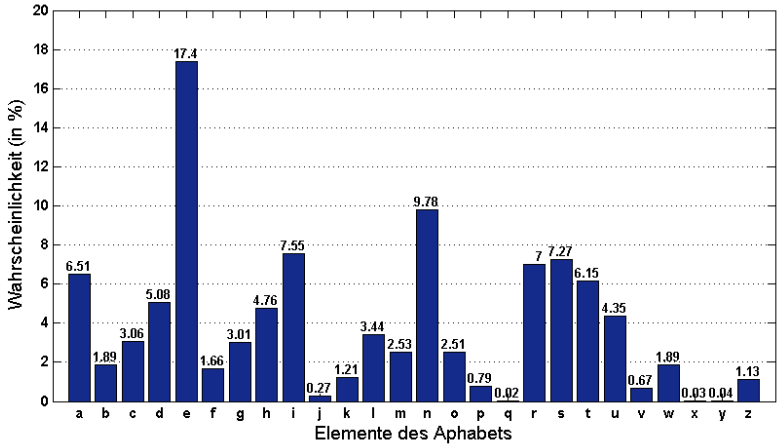


Abbildung 4 - A priori Verteilung des deutschen Alphabets

Entsteht nun in den zugehörigen Chiffretextblöcken ebenfalls eine Ungleichverteilung der möglichen Elemente, kann unter Umständen ein Zusammenhang zwischen Chiffretext und Klartext hergestellt werden. Das Prinzip eines solchen statistischen Angriffs wird in Beispiel 3 gezeigt.

Beispiel 3

In diesem Beispiel werden wieder Nachrichtenblöcke mit einer festen Länge von 3 Bit betrachtet. Somit besteht das zugrunde liegende Alphabet aus $2^3 = 8$ Elementen. Aus diesen Elementen kann nun eine Nachricht mit beliebiger Länge erstellt werden. Die Verteilung der Elemente wird exemplarisch so gewählt, dass sie dem Verlauf nach der Verteilung der Buchstaben von ‚*a*' bis ‚*h*' aus Abbildung 4 entspricht. Um die Forderung

$$\sum_{i=1}^{|M|} P(M_i) = 1$$

zu erfüllen, werden die Werte noch entsprechend skaliert. Daraus ergibt sich eine a priori Verteilung der Quelle wie sie in Abbildung 5 dargestellt ist.

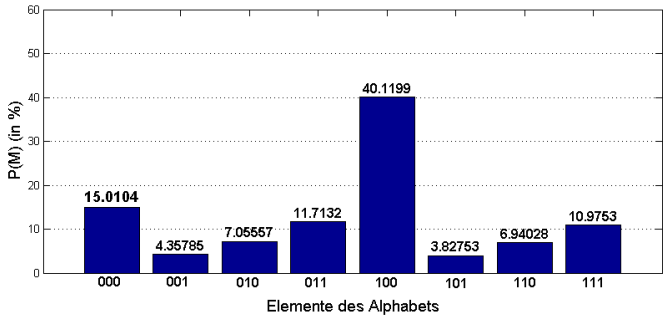


Abbildung 5 - A priori Verteilung zu Beispiel 3

Da in diesem Beispiel kein absolut sicheres Kryptosystem betrachtet werden soll, kann die Schlüssellänge auf 3 Bits begrenzt werden. Um die Komplexität des Beispiels gering zu halten, wird außerdem angenommen, dass jeder Klartextblock mit dem gleichen Schlüssel K_i chiffriert wird. In der Realität werden zwar meistens pseudozufällige Schlüsselfolgen verwendet, wenn diese aber eine feste Periode haben, die dem Angreifer bekannt ist und er genügend Chiffretexte besitzt, ist das Beispiel übertragbar.

In dem Beispiel soll folgende Klartextfolge übertragen werden:

$$M_i = \{m_1|m_2|m_3|m_4|m_5|m_6\} = \{100|111|100|010|100|110\}$$

Dazu wird jeder Block mit einem Schlüssel K_i aus dem Schlüsselraum chiffriert. In dem Beispiel wird der Schlüssel

$$K_i = \{0\ 1\ 0\}$$

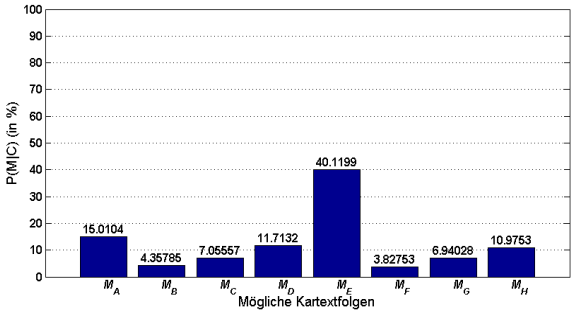
verwendet. Als Ver- und Entschlüsselungsfunktion wird wieder die kontravalente Verknüpfung gewählt:

$$D(K_i, M_i) = E(K_i, M_i) = K_i \oplus M_i$$

Aus den Klartextblöcken werden also durch die Verschlüsselung folgende Chiffretextblöcke erzeugt:

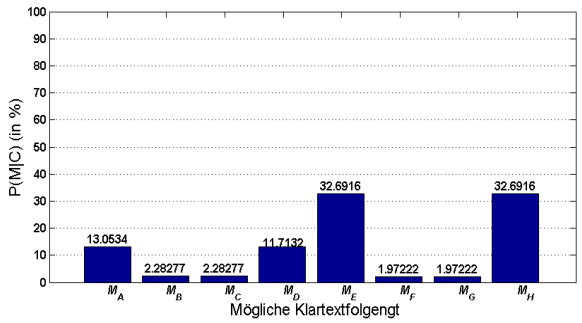
$$C_j = \{c_1|c_2|c_3|c_4|c_5|c_6\} = \{110|101|110|000|110|100\}$$

Nun wird das Beispiel aus Sicht eines Angreifers betrachtet. Ihm ist sowohl die Nachricht M_i als auch der verwendete Schlüssel K_i unbekannt. Er empfängt jedoch die Chiffretextblöcke durch das Belauschen des kryptographischen Kanals. Wie der Tabelle 1 zu entnehmen ist, handelt es sich hier also um einen *Ciphertext-only-Angriff.* Mit Hilfe der Chiffretextblöcke ist es dem Angreifer nun möglich, nach Formel (16) die a posteriori Wahrscheinlichkeit $P(M_i|C_j)$ für jede mögliche Klartextfolge zu berechnen. Der Angriff ist erfolgreich, wenn sich nach einer gewissen Anzahl von empfangenen Chiffretextblöcken eine Klartextfolge herausbildet, dessen a posteriori Wahrscheinlichkeit signifikant höher als die der anderen Klartextfolgen ist. Der Verlauf dieses Angriffs wird im Folgenden Schrittweise dargestellt[4].

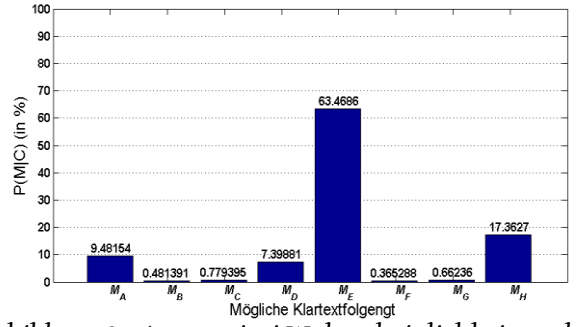
Schritt: 1	Empfangene Chiffretextblöcke:
 *Abbildung 6 - A posteriori Wahrscheinlichkeit nach einem Chiffreblock*	$C \coloneqq \{\mathbf{110}\}$ Mögliche Klartextfolgen: $M_A \coloneqq \{000\}$ $M_B \coloneqq \{001\}$ $M_C \coloneqq \{010\}$ $M_D \coloneqq \{011\}$ $M_E \coloneqq \{\mathbf{100}\}^*$ $M_F \coloneqq \{101\}$ $M_G \coloneqq \{110\}$ $M_H \coloneqq \{111\}$ *gesendete Klartextfolge

[4] Zur Berechnung der Werte und zur graphischen Darstellung wurde ein Matlab-Skript entwickelt, welches sich in Anhang A befindet.

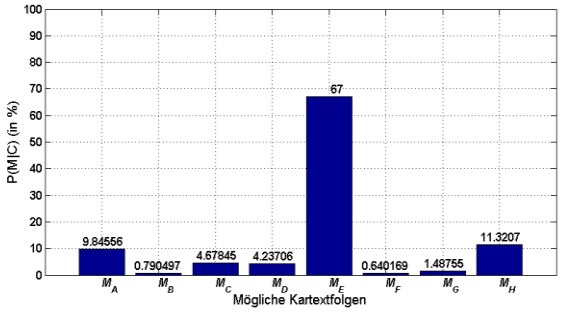
Nach dem ersten Schritt kann der Angreifer noch keine Rückschlüsse auf den gesendeten Klartextblock ziehen, da die a posteriori Wahrscheinlichkeit (Abbildung 6) der a priori Wahrscheinlichkeit (Abbildung 5) gleicht. Nach Shannon ist das System also noch absolut sicher.

Schritt: 2	Empfangene Chiffretextblöcke:
 *Abbildung 7 - A posteriori Wahrscheinlichkeit nach zwei Chiffreblöcken*	$C \coloneqq \{110\|\mathbf{101}\}$ Mögliche Klartextfolgen: $M_A \coloneqq \{000\|011\}$ $M_B \coloneqq \{001\|010\}$ $M_C \coloneqq \{010\|001\}$ $M_D \coloneqq \{011\|000\}$ $M_E \coloneqq \{100\|\mathbf{111}\}^*$ $M_F \coloneqq \{101\|110\}$ $M_G \coloneqq \{110\|101\}$ $M_H \coloneqq \{111\|100\}$ **gesendete Klartextfolge*

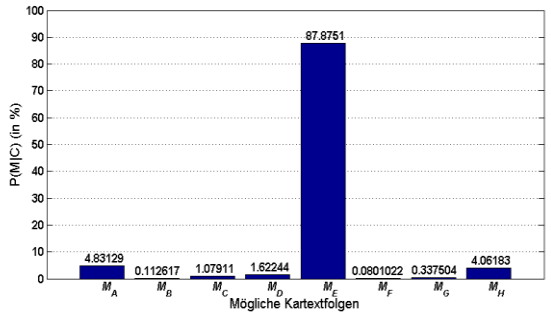
Auch nach dem zweiten empfangenen Chiffretextblock besitzt der Angreifer keine Möglichkeit, auf den Klartext zu schließen, da hier beispielsweise der Klartext M_E und M_H die gleiche a posteriori Wahrscheinlichkeit aufweisen (Abbildung 7).

Schritt: 3	Empfangene Chiffretextblöcke:
 *Abbildung 8 - A posteriori Wahrscheinlichkeit nach drei Chiffreblöcken*	$C \coloneqq \{110\|101\|\mathbf{110}\}$ Mögliche Klartextfolgen: $M_A \coloneqq \{000\|011\|000\}$ $M_B \coloneqq \{001\|010\|001\}$ $M_C \coloneqq \{010\|001\|010\}$ $M_D \coloneqq \{011\|000\|011\}$ $M_E \coloneqq \{100\|111\|\mathbf{100}\}^*$ $M_F \coloneqq \{101\|110\|101\}$ $M_G \coloneqq \{110\|101\|110\}$ $M_H \coloneqq \{111\|100\|111\}$ **gesendete Klartextfolge*

Nach dem dritten Chiffretextblock bildet sich nun mit M_E erstmal eine Klartextfolge heraus, die eine deutlich höhere a posteriori Wahrscheinlichkeit besitzt als alle anderen möglichen Klartextfolgen (Abbildung 8). Bei dieser Klartextfolge handelt es sich tatsächlich um die gesendete Klartextfolge. Der Angreifer kann sich dessen jedoch noch nicht wirklich sicher sein, da auch M_A, M_D und M_H eine nicht verschwindende Wahrscheinlichkeit besitzen.

Schritt: 4	Empfangene Chiffretextblöcke:
 *Abbildung 9 - A posteriori Wahrscheinlichkeit nach vier Chiffreblöcken*	$C \coloneqq \{110\|101\|110\|\mathbf{000}\}$ Mögliche Klartextfolgen: $M_A \coloneqq \{000\|011\|000\|110\}$ $M_B \coloneqq \{001\|010\|001\|111\}$ $M_C \coloneqq \{010\|001\|010\|100\}$ $M_D \coloneqq \{011\|000\|011\|101\}$ $M_E \coloneqq \{100\|111\|100\|\mathbf{010}\}^*$ $M_F \coloneqq \{101\|110\|101\|011\}$ $M_G \coloneqq \{110\|101\|110\|000\}$ $M_H \coloneqq \{111\|100\|111\|001\}$ * *gesendete Klartextfolge*

Der vierte Chiffretextblock $c_4 = \{000\}$ ändert das Ergebnis im Vergleich zu dem 3. Schritt nur geringfügig (Abbildung 9). Dies liegt daran, dass der gesendete Klartextblock $m_4 = \{010\}$ nur eine relativ geringe a priori Wahrscheinlichkeit besitzt (siehe Abbildung 5).

Schritt: 5	Empfangene Chiffretextblöcke:
 *Abbildung 10 - A posteriori Wahrscheinlichkeit nach fünf Chiffreblöcken*	$C \coloneqq \{110\|101\|110\|000\|\mathbf{110}\}$ Mögliche Klartextfolgen: $M_A \coloneqq \{000\|011\|000\|110\|000\}$ $M_B \coloneqq \{001\|010\|001\|111\|001\}$ $M_C \coloneqq \{010\|001\|010\|100\|010\}$ $M_D \coloneqq \{011\|000\|011\|101\|011\}$ $\boldsymbol{M_E} \coloneqq \{100\|111\|100\|010\|\mathbf{100}\}^*$ $M_F \coloneqq \{101\|110\|101\|011\|101\}$ $M_G \coloneqq \{110\|101\|110\|000\|110\}$ $M_H \coloneqq \{111\|100\|111\|001\|111\}$ * *gesendete Klartextfolge*

Im 5. Schritt wird nun der Klartextblock $m_5 = \{100\}$ gesendet. Dieser besitzt eine relativ hohe a priori Wahrscheinlichkeit (siehe Abbildung 5). Dadurch erhöht sich die a posteriori Wahrscheinlichkeit von Klartextfolge M_E im Vergleich zum 4. Schritt deutlich. Nun kann der Angreifer bereits mit einer Sicherheit von 87,9 % auf die richtige Klartextfolge schließen (Abbildung 10). Auf Grund der einfachen Verknüpfung zwischen Klartext, Chiffretext und Schlüssel ($C_j = K_i \oplus M_i$) kann der Angreifer nicht nur den wahrscheinlichsten Klartext, sondern auch den wahrscheinlichsten Schlüssel ermitteln ($K_i = C_j \oplus M_i$), um so auch alle nachfolgenden Klartextfolgen zu entschlüsseln ($M_i = C_j \oplus K_i$).

Um die Erfolgsaussichten solcher statistischen Angriffe zu reduzieren bzw. den nötigen Aufwand zu erhöhen, schlägt Shannon in [6] zwei grundlegende Prinzipien für kryptographische Algorithmen vor:

Diffusion
Das Prinzip der Diffusion schreibt vor, dass jedes Chiffretextelement von möglichst vielen Klartextelementen und von dem gesamten Schlüsselblock abhängen soll. Dadurch werden mögliche statistische Besonderheiten des Klartextes bei der Chiffrierung entfernt. Somit kann die benötigte Menge an Chiffretextblöcken für einen erfolgreichen Angriff deutlich erhöht werden.

Konfusion
Im obigen Beispiel bestand lediglich ein einfacher Zusammenhang zwischen dem Chiffretext C_j, dem Klartextblock M_i und dem Schlüssel K_i. Somit konnte durch den beschriebenen Angriff nicht nur der Klartext, sondern auch der verwendete Schlüssel einfach ermittelt werden. Nach dem Prinzip der Konfusion soll dieser Zusammenhang nun möglichst komplex gestaltet werden, um so den nötigen Aufwand eines Angriffs zu steigern.

Diese beiden Prinzipien dienen also dazu, die praktische Sicherheit von kryptographischen Systemen zu erhöhen und sollten somit in jedem modernen Verschlüsselungsalgorithmus implementiert werden.

2.2 Steganographie

Neben der Kryptologie bildet die Steganographie eine weitere Möglichkeit, den Inhalt von Nachrichten vor Drittpersonen zu schützen. Hierbei wird jedoch nicht der Inhalt der Nachricht unleserlich gemacht, sondern ihre bloße Existenz verschleiert. Dies geschieht in der Regel in dem die zu schützende Nachricht in einer weiteren belanglos wirkenden Nachricht versteckt wird. Wie die Kryptologie hat auch die Steganographie eine weitreichende Geschichte. In der Antike wurden beispielsweise häufig mit Wachs überzogene Holztafeln zur Nachrichtenübertragung herangezogen. Dabei wurde die Nachricht in die Wachsschicht eingeritzt, sodass sie vom Empfänger gelesen werden konnte. Eine steganographische Methode bestand nun darin, eine geheime Nachricht direkt in die Holztafel zu gravieren, diese dann mit einer Wachsschicht zu überdecken, um dort anschließend eine harmlose und geläufige Nachricht hinein zu ritzen. Das Hinzufügen einer harmlosen Nachricht sollte dazu führten, dass die Wachstafel als unverdächtig eingestuft wurde.

Das Prinzip einer steganographischen Nachrichtenübertragung zeigt Abbildung 11.

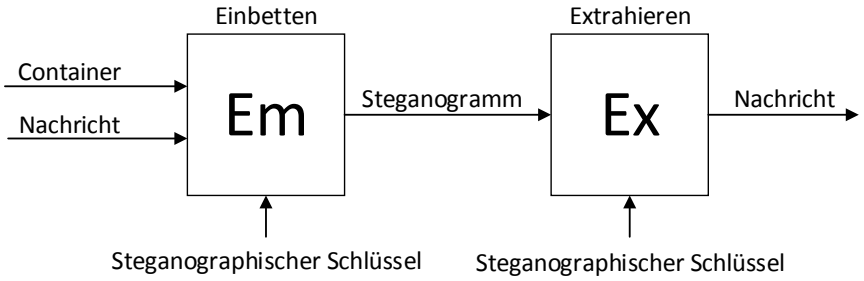


Abbildung 11 - Blockschaltbild einer steganographischen Nachrichtenübertragung

In der digitalen Steganographie werden an Stelle der Wachstafeln häufig Bilder-, Video– oder Audiodateien als sogenannte Container verwendet. Diese Container bieten den Vorteil, dass die zugrunde liegenden Daten häufig einen Digitalisierungsprozess durchlaufen und somit ein gewisses Hintergrundrauschen aufweisen. In dieses Rauschen kann nun eine digitale Nachricht integriert werden, ohne dass es einem menschlichen Betrachter auffallen würde. Um eine computergestützte Analyse zu überstehen, müssen jedoch noch weitere Maßnahmen getroffen werden, da eine hinzugefügte Nachricht das statistische Profil einer Datei verändern kann. Ein Container, der eine eingebettete Nachricht enthält, wird als Steganogramm bezeichnet.

Da in dieser Arbeit Bilddaten als Container für die Nachrichten verwendet werden, werden andere Containerarten nicht weiter behandelt. Bilddaten bieten den Vorteil, dass sie besonders häufig über das Internet versendet und somit im Allgemeinen als unverdächtig eigestuft werden. Ein Algorithmus zum Einbetten von Nachrichten in ein Containerbild kann auf verschiedene Arten umgesetzt werden. Die meisten Vorgehensweisen beruhen darauf, dass die einzelnen Bildpunkte (*Pixel*) eines Containerbildes so manipuliert werden, dass die niederwertigsten Bits (*LSB*) der Pixel den Bits der geheimen Nachricht entsprechen. Der Empfänger des Steganogramms muss dann lediglich die LSBs extrahieren und kann daraus den Inhalt der eingebetteten Nachricht wieder herstellen.

Auch für steganographische Algorithmen sollte das in 2.1.1 vorgestellte Kerckhoffs'sche Prinzip gelten. Dieses Prinzip besagt, dass der Algorithmus keine Geheimhaltung erfordern sollte, um als sicher zu gelten. Die Sicherheit sollte somit alleine von einem geheimen Schlüssel abhängen. Ein steganographischer Schlüssel verzerrt jedoch, im Gegensatz zu kryptographischen Schlüsseln, in der Regel nicht den Inhalt einer Nachricht, sondern gibt lediglich an, an welchen Stellen die Nachricht im Steganogramm eingebettet wird. Bei einer Bilddatei als Container würde der geheime

Schlüssel also beispielsweise festlegen, welche Pixel des Bildes die geheimen Informationen tragen.

2.2.1 Steganalyse

Wie bei der Kryptographie gibt es auch in der Steganographie eine komplementäre Teilwissenschaft, welche sich darauf spezialisiert, die Sicherheit von steganographischen Methoden zu analysieren. Diese Teilwissenschaft im Bezug zur Steganographie wird als Steganalyse bezeichnet.

Die Ziele der Steganalyse sind unter anderem:

1) Das Erkennen, ob Nachrichten in dem Container eingebettet sind.
2) Das Extrahieren der Nachricht aus dem Container, wenn eine Nachricht vorhanden ist.

Wie auch bei der Kryptoanalyse, wird bei der Steganalyse zwischen verschiedenen Angriffsarten unterschieden. Die Beschreibungen und Begriffsbezeichnungen stammen aus [14].

- ***stego-only-Angriff***: Bei diesem Angriff ist nur das Steganogramm für den Angreifer bekannt.
- ***known-cover-Angriff***: Hier hat der Angreifer Zugriff auf das Steganogramm und auch auf den originalen Container (*Cover*).
- ***known-message-Angriff***: Es ist sowohl das Steganogramm, als auch die eingebettete Nachricht bekannt.
- ***chosen-stego-Angriff***: Der Angreifer kennt die Funktionsweise des Algorithmus und hat Zugriff auf das Steganogramm.
- ***chosen-message-Angriff:*** Der Angreifer hat Zugriff auf das Steganographie-System und kann eigene Steganogramme aus ausgewählten Nachrichten erzeugen. Das Ziel besteht darin, den zugrunde liegenden Algorithmus zu erkennen.
- ***known-stego-Angriff***: Das Steganographie-System ist verfügbar und sowohl das Steganogramm, als auch der originale Container sind bekannt.

In dem folgenden Abschnitt werden zwei verschiedene Angriffsmethoden genauer betrachtet[5].

[5] Die Matlab-Skripte, mit denen die Angriffe durchgeführt wurden, befinden sich in Anhang A.

Visuelle Angriffe

Auch wenn sich das Hintergrundrauschen von digitalisierten Bildern oftmals nur schwer von zufälligem Rauschen unterscheiden lässt, ist es nicht stochastisch unabhängig von dem restlichen Bildinhalt. Diese Abhängigkeit lässt sich durch einen speziellen Filter sichtbar machen der von A. Westfeld in seiner Arbeit „*Angriffe auf steganographische Systeme*“ [15] beschrieben wird. Bei diesem Filter werden die niederwertigsten Bits (*LSBs*) eines Bildes extrahiert und anschließend mit dem maximal möglichen Wert eines Pixels multipliziert. Bei einem Bild mit einer 8-Bit-Farbtiefe liegt der Verstärkungsfaktor somit bei 255. Durch die Filterung wird also ein Pixel, dessen LSB gleich 0 ist, als schwarzer Punkt dargestellt und ein Pixel, dessen LSB gleich 1 ist, als weißer Punkt. Ein Beispiel einer solchen Filterung zeigt Tabelle 2. Um den Effekt besonders gut sichtbar zu machen, wurde der Kontrast des Originalbildes nachträglich erhöht.

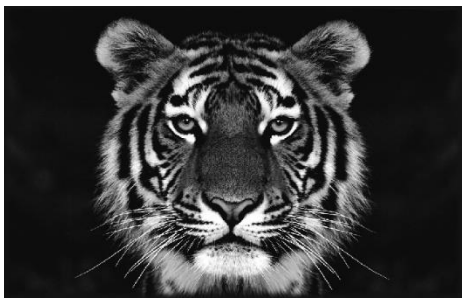
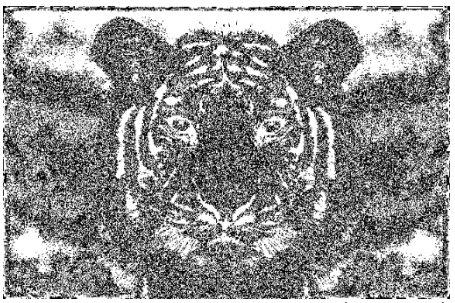
	

Abbildung 12 - Originalbild[6]	*Abbildung 13 – Originalbild (gefiltert)*

Tabelle 2 - Visueller Angriff auf das Originalbild

Auf Abbildung 13 sind die Strukturen des Originalbildes (Abbildung 12) noch deutlich zu erkennen, obwohl die Bildinformationen nur aus den LSBs des Originalbildes gewonnen werden.

Werden die LSBs nun durch eine verschlüsselte Nachricht ersetzt, gehen diese Strukturen verloren. Dies zeigt Tabelle 3. Hier wurde in die linke Hälfte des Originalbildes eine Nachricht eingebettet. Der Unterschied, der im ungefilterten Bild kaum wahrzunehmen ist (Abbildung 14), wird durch die Filterung deutlich sichtbar (Abbildung 15).

[6] Das Originalbild wurde unter der *Creative-Commons-Lizenz* (CC-Lizenz) auf [24] bereitgestellt. Es wurde für das Beispiel in ein Graustufenbild umgewandelt und der Kontrast wurde verändert.

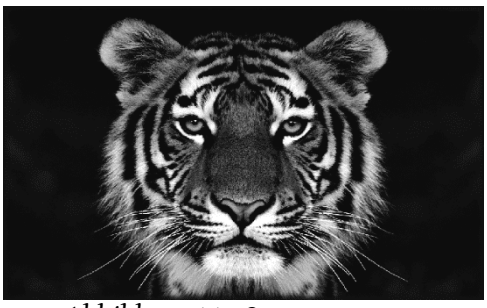
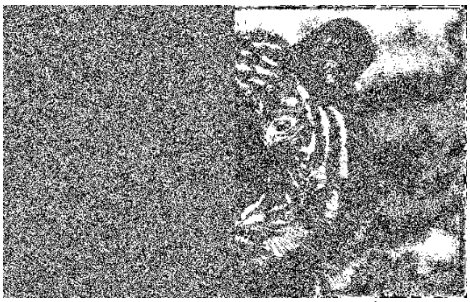
	

Abbildung 14 - Steganogramm	*Abbildung 15 – Steganogramm (gefiltert)*

Tabelle 3 - Visueller Angriff auf ein Steganogramm

Visuelle Angriffe funktionieren jedoch nur, wenn das Containerbild deutliche Strukturen und einen hohen Kontrast aufweist. Tabelle 4 zeigt einen Angriff auf ein Steganogramm, bei dem der Kontrast des Containerbildes reduziert wurde. Auch hier befindet sich in die linke Hälfte des Bildes eine eingebettete Nachricht (Abbildung 16). Der Unterschied zu dem originalen Hintergrundrauschen ist jedoch in dem gefilterten Bild kaum noch zu erkennen (siehe Abbildung 17).

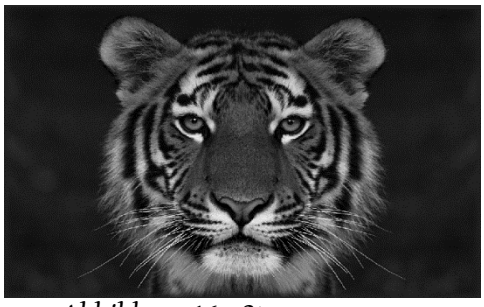
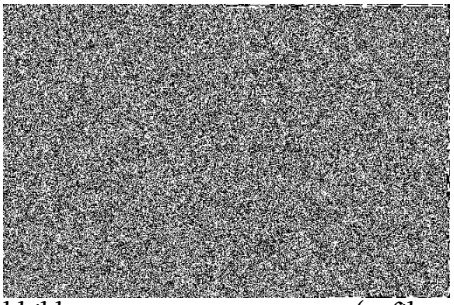
	

Abbildung 16 - Steganogramm	*Abbildung 17 – Steganogramm (gefiltert)*

Tabelle 4 - Visueller Angriff auf ein Steganogramm mit reduziertem Kontrast

Das Ergebnis eines visuellen Angriffs hängen also stark von der Beschaffenheit des Containerbildes ab. Außerdem wird bei dem Angriff die Fähigkeit des menschlichen Sehens ausgenutzt, gewisse Muster erkennen und interpretieren zu können. Dies lässt sich nur äußerst schwierig in einem computergestützten Verfahren umsetzen.

Aus diesem Grund werden statistische Angriffe entwickelt. Die Ergebnisse statistischer Angriffe hängen weniger stark von der Beschaffenheit des Containerbildes ab und sie lassen sich gut durch einen Algorithmus realisieren.

Statistische Angriffe

Ein Angriff, der auf statistische Methoden beruht, wurde von A. Westfeld und A. Pfitzman in ihrer Arbeit „*Attacks on Steganographic Systems*“ vorgestellt [16]. Die Arbeit ist eine Erweiterung von Westfelds Arbeit „*Angriffe auf steganographische Systeme*“.

Bei diesem Angriff wird davon ausgegangen, dass die LSBs des Containerbildes durch die jeweiligen Bits der Nachricht substituiert werden. Dies ist eine einfache und daher weit verbreitete steganographische Methode.

Im Verlauf des Angriffs werden die Pixel eines Bildes auf Wertepaare, sogenannte *POV*s (engl.: „*pairs of values*“) untersucht. Ein Wertepaar besteht dabei aus einem geraden Wert und dem darüber liegenden und somit ungeraden Wert, also beispielsweise $\{0,1\}$ oder $\{2,3\}$. Besteht ein Pixel aus jeweils einem Byte (kann also 256 verschiedene Werte annehmen), können die $256/2 = 128$ möglichen *POVs* folgendermaßen gebildet werden:

$$POV_k = \{2 \cdot k, 2 \cdot k + 1 \mid 0 \leq k \leq 127\} \tag{18}$$

Durch die oben beschriebene Substitution der LSBs werden die Werte der Pixel nur innerhalb ihres *POVs* transferiert, da gerade Werte entweder gleich bleiben oder um eins inkrementiert werden, während ungerade Werte entweder gleich bleiben oder um eins dekrementiert werden.

Die Transformationen, die durch die Substitution der LSBs entstehen können, zeigt Abbildung 18.

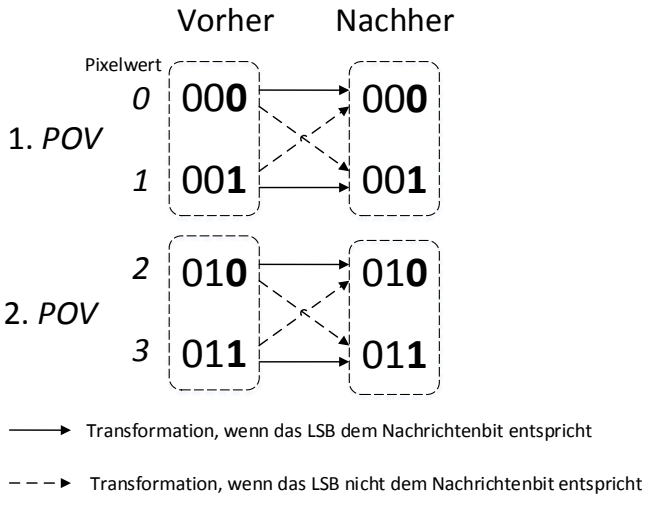


Abbildung 18 - Transformation bei LSB-Substitution

Somit ändert sich durch das Einbetten zwar die Anzahl der einzelnen Elemente eines POVs, die Summe beider Elemente bleibt jedoch konstant. Tabelle 5 zeigt zwei Abbildungen, in denen die Anzahl der Elemente verschiedener POVs ($POV_9 = \{18,19\}$ bis $POV_{20} = \{40,41\}$) als Balkendiagramm dargestellt sind. Bei dem oberen Diagramm wurde ein unverändertes Containerbild analysiert, bei dem unteren Diagramm ein daraus erzeugtes Steganogramm. Als Containerbild wurde wieder der zuvor gezeigte Tigerkopf verwendet.

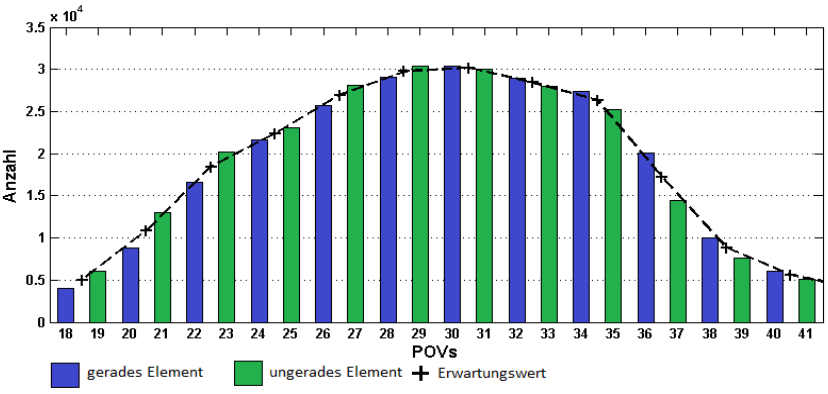


Abbildung 19 - POV-Verteilung im Containerbild

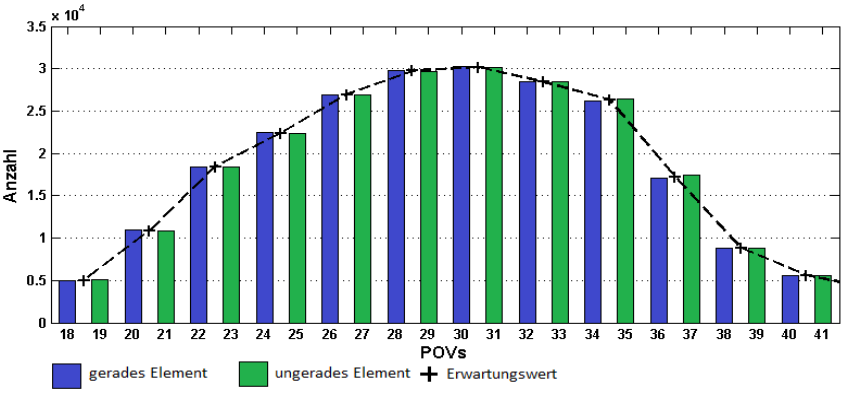


Abbildung 20 - POV-Verteilung im Steganogramm

Tabelle 5 - POVs in Bilddaten

Wird eine verschlüsselte Nachricht in ein Containerbild durch Substitution der LSBs eingebettet, passt sich die Anzahl der beiden Elemente eines POVs (gerade und ungerade) aneinander an (Abbildung 20). Dies liegt daran, dass die Einsen und Nullen in einer verschlüsselten Nachricht nahezu gleichverteilt sind. Das Einbetten einer Null erzeugt einen geraden Wert, während das Einbetten einer Eins immer einen ungeraden Wert erzeugt. Durch diese Anpassung entspricht die Anzahl der Elemente in einem POV

ihrem theoretischen Erwartungswert, der durch den Mittelwert beider Elemente gebildet wird (in den Diagrammen ist dieser mit einem „+“ markiert).

In den Bilddaten des originalen Containerbildes sind die Elemente eines POVs nicht gleichverteilt. Dadurch weicht die Anzahl der einzelnen Elemente von dem theoretischen Erwartungswert ab (siehe Abbildung 19). Diese Abweichung wird im Verlauf des Angriffs durch einen *Chi-Quadrat*-Test[7] geprüft, wodurch der Angriff auch seine Bezeichnung als *chi-square-Angriff* erhalten hat. Das Ergebnis dieser Prüfung kann dann entweder graphisch oder tabellarisch dargestellt werden. Da sich der Erwartungswert durch die Substitution der LSBs nicht verändert (was daran liegt, dass die Summe der Elemente eines POVs konstant bleibt), wird für diesen Angriff das originale Containerbild nicht benötigt, es handelt sich also um einen *stego-only-Angriff*.

In der Arbeit „*Pairs of Values and the chi-squared Attack*” für die Iowa State University [17] wurde von C. A. Stanley der *chi-square-Angriff* in einem Matlab-Skript implementiert[8]. In der Arbeit wurden auch die Stärken und Schwächen des Angriffs genauer untersucht. Zu den Schwächen gehört demnach auch, dass die Ergebnisse des Angriffs unter anderem von der Beschaffenheit des Containerbildes abhängig sind. Somit sinkt die Aussagekraft des Angriffs, je stärker das Containerbild verrauscht ist. Das Matlab-Skript von C. A. Stanley stellt die Ergebnisse des *Chi-Quadrat-Tests* graphisch dar, sodass leicht ermittelt werden kann, ob und in wieviel Prozent der Bildpunkte eine Nachricht eingebettet worden ist. Die Ergebnisse eines solchen Angriffs auf ein Steganogramm, welches durch Substitution der LSBs erzeugt wurde, zeigt Tabelle 6.

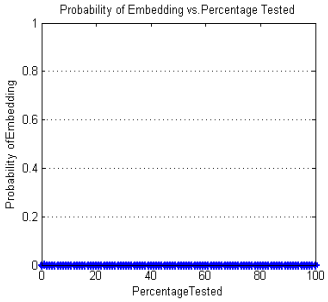
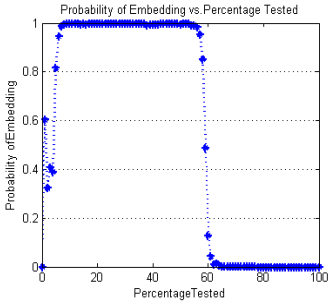
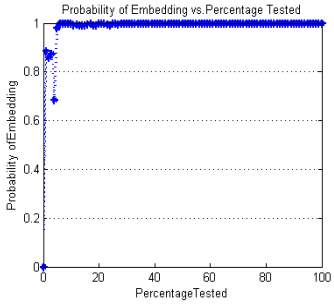
Steganogramm mit 0% eingebetteten Daten	Steganogramm mit 50% eingebetteten Daten	Steganogramm mit 100% eingebetteten Daten
		

Tabelle 6 - Ergebnisse des chi-square-Angriffs auf Steganogramme mit substituieren LSBs

[7] Ein Chi-Quadrat-Test (χ^2-Test) bezeichnet in der mathematischen Statistik eine Gruppe von Hypothesentests mit χ^2-verteilter Testprüfgröße [26].
[8] das leicht modifizierte Matlab-Skript befindet sich in Anhang A.

A. Westfeld und A. Pfitzman beschreiben in [16] jedoch einen Algorithmus, bei dem der *chi-square-Angriff* ein falsches Ergebnis liefert. Dieser Algorithmus schreibt vor, den Wert eines Pixels um Eins zu dekrementieren, wenn das *LSB* ungleich dem entsprechenden Bit der Nachricht ist. Nur wenn der Wert des Pixels gleich 0 ist, wird er stattdessen inkrementiert (eine detaillierte Beschreibung des Algorithmus erfolgt in Kapitel 3.4). Die Elemente eines POVs werden durch die Transformation in andere *POVs* verschoben, es entsteht somit keine Gleichverteilung, so wie sie durch die Substitution der LSBs erzeugt werden würde (siehe Abbildung 21).

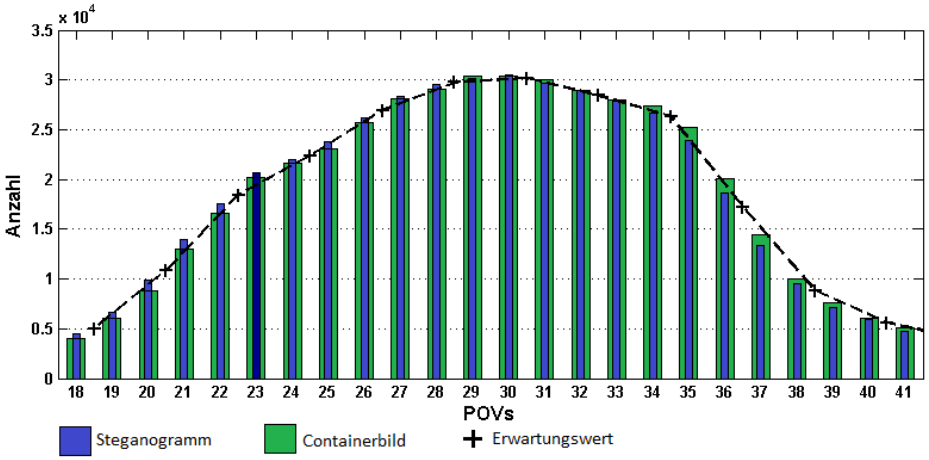


Abbildung 21 - Verteilung der Elemente im Containerbild und im Steganogramm mit dekrementierten Pixelwerten

Die Ergebnisse eines Angriffs auf so erzeugte Steganogramme zeigt Tabelle 7. Der Test fällt nun für alle drei untersuchten Fälle negativ aus.

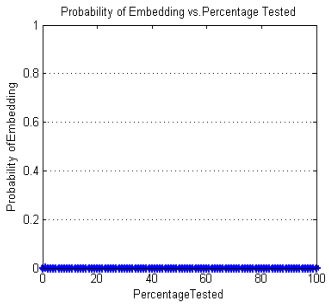
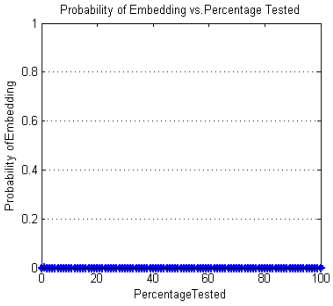
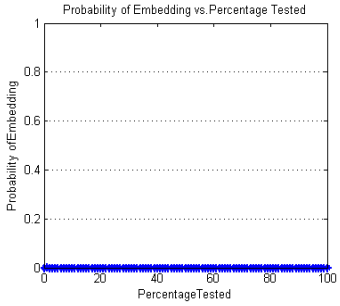
Steganogramm mit 0% eingebetteten Daten	Steganogramm mit 50% eingebetteten Daten	Steganogramm mit 100% eingebetteten Daten
		

Tabelle 7 - Ergebnis des chi-square-Angriffs auf Steganogramme mit dekrementierten Pixelwerten

3 Stegano- und kryptographische Methoden

In diesem Kapitel werden steganographische und kryptographische Methoden vorgestellt, die für das beschriebene System verwendet werden. Dazu gehört der *Advanced Encryption Standard* (Kapitel 3.1), die *One-Time-Pad-Verschlüsselung* (Kapitel 3.2) und der verwendete Betriebsmodus mit dem die *AES*-Blockchiffre in eine Stromchiffre für beliebig lange Nachrichten überführt wird (Kapitel 3.3). Außerdem wird der steganographische Algorithmus beschrieben, mit dem die verschlüsselten Nachrichten in ein Containerbild eingebettet werden (Kapitel 3.4) und ein Verfahren, mit dem pseudozufällige Zahlenfolgen erzeugt werden können (Kapitel 3.5).

3.1 Advanced Encryption Standard (AES)

Der *Advanced Encryption Standard* (*AES*) ist ein weit verbreitetes Verfahren in der modernen Kryptographie. Der Algorithmus wurde im Rahmen einer Ausschreibung entwickelt, mit der das amerikanische Handelsministerium 1997 einen Nachfolger für den *Data Encryption Standard* (DES) gesucht hatte [18]. Diese Ausschreibung gewannen die Kryptographen Vincent Rijmen und Joan Daemen mit ihrem Verfahren, was damals noch als *Rijndael* bezeichnet wurde. Seit 2001 ist es in den USA offiziell standardisiert und ist dort auch für die höchste Geheimhaltungsstufe *Top Secret* freigegeben. Damit ist es der erste öffentlich bekannte Algorithmus, der für diese Geheimhaltungsstufe zugelassen wurde. Das Verfahren ist nicht patentiert und somit frei nutzbar.

Der *Rijndael*-Algorithmus ist eine Blockchiffre mit einer Blocklänge von 128-, 192- oder 256-Bit. Im Rahmen der Standardisierung wurde die Länge jedoch auf 128-Bit festgelegt. Somit kann der *AES* auch als Spezialfall des *Rijndael*-Algorithmus bezeichnet werden. Als Schlüssellänge stehen 128-, 192- oder 256-Bit zur Verfügung. Während des Verschlüsselungsvorgangs durchläuft der Klartextblock den Algorithmus mehrere Runden. Die Anzahl der Runden ist von der Schlüssellänge abhängig:

Schlüssellänge $\|K\|$:	**Rundenanzahl n:**
128-Bit	10
192-Bit	12
256-Bit	14

Tabelle 8 - Rundenanzahl des AES

3.1.1 Verschlüsselungsvorgang

Die einzelnen Funktionsaufrufe des Verschlüsselungsalgorithmus zeigt das Flussdiagramm in Abbildung 22.

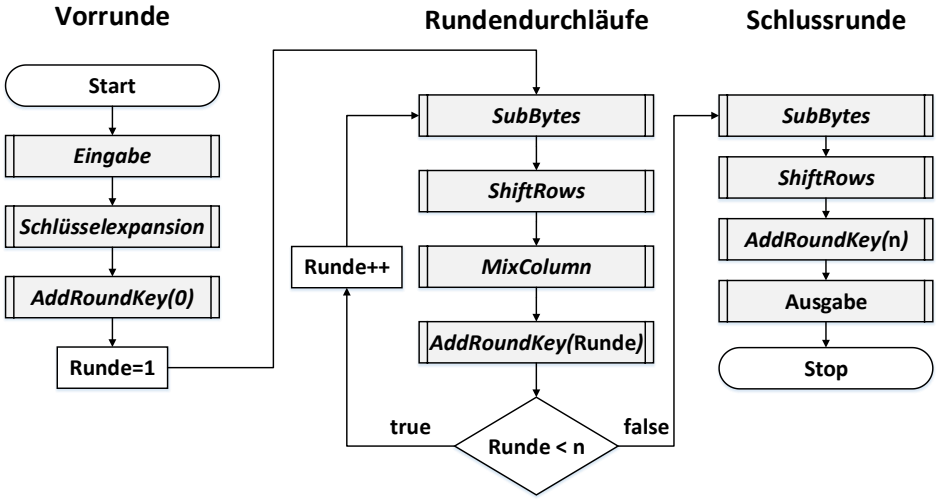


Abbildung 22 - AES-Verschlüsselungsalgorithmus

Im nachfolgenden Abschnitt werden die einzelnen Funktionen des Algorithmus anhand eines Beispiels beschrieben. Die Beispielwerte wurden aus [19] übernommen. Es wird ein 128-Bit-Schlüssel verwendet, woraus eine Rundenanzahl von $n = 10$ resultiert. Die Beispielnachricht lautet:

$$M_i = \{32|43|f6|a8|88|5a|30|8d|31|31|98|a2|e0|37|07|34\}^{9}$$

Und der verwendete Schlüssel:

$$K_i = \{2b|7e|15|16|28|ae|d2|a6|ab|f7|15|88|09|cf|4f|3c\}$$

[9] Die Werte sind byteweise zusammengefasst und im Hexadezimalformat angegeben

1) Eingabe

In der Eingabe-Funktion werden jeweils die 16 Bytes der Nachricht M_i und des Schlüssels K_i spaltenweise in eine 4x4-Matrix gespeichert. Die Matrix für die Nachricht wird als *State-Matrix* bezeichnet und die für den Schlüssel als *Cipher-Key-Matrix.*

$State - Matrix$

32	88	31	$e0$
43	$5a$	31	37
$f6$	30	98	07
$a8$	$8d$	$a2$	34

$Cipher - Key - Matrix$

$2b$	28	ab	09
$7e$	ae	$f7$	cf
15	$d2$	15	$4f$
16	$a6$	88	$3c$

2) Schlüsselexpansion

Mit der Funktion *Schlüsselexpansion* wird aus der *Cipher-Key-Matrix* für jede Runde ein Rundenschlüssel generiert. Jeder Rundenschlüssel besteht ebenfalls aus einer 4x4-Matrix. Somit müssen also aus der ursprünglichen 4x4-Matrix zehn weitere 4x4-Matritzen erzeugt werden. Die Erzeugung der Rundenschlüssel erfolgt rekursiv, der aktuelle Rundenschlüssel wird also alleine aus dem vorangegangenen Rundenschlüssel generiert. Die *Cipher-Key-Matrix* kann somit auch als Initialmatrix betrachtet werden, aus dem der erste Rundenschlüssel erzeugt wird.

Zur Generierung eines Rundenschlüssels werden in der Funktion mehrere Unterfunktionen aufgerufen, die im Nachfolgenden erläutert werden.

2.1) *RotWord*

In diesem Schritt wird die letzte Spalte aus der jeweiligen Vorgängermatrix entnommen (für den 1. Rundenschlüssel also die 4. Spalte der *Cipher-Key-Matrix*) und einmal nach oben rotiert. Somit landet das oberste Element an vierte Stelle, während alle anderen Elemente eine Zeile höher rücken:

09	→	cf
cf		$4f$
$4f$		$3c$
$3c$		09

2.2) *SubBytes*

Nach dem die Spalte durch *RotWord* bearbeitet wurde, folgt die Funktion *SubBytes*. In dieser Funktion wird jedes Element der Spalte durch ein Element einer Substitutionstabelle, der sogenannten *S-BOX* (Tabelle 10), ersetzt. Dabei bestimmt das erste Nibble (also die ersten 4 Bit) des bytegroßen Elements die Spalte und das zweite Nibble die Zeile des Tabelleneintrags, welches zur Substitution ausgewählt wird. Wie

die Tabelleneinträge der *S-BOX* zustande kommen, wird in Kapitel 3.1.3 näher erläutert. Die aktuelle Spalte wird nach dem Prinzip folgendermaßen substituiert:

cf	→	$8a$
$4f$		84
$3c$		eb
09		01

2.3) *Rcon*

Für den nächsten Schritt wird eine weitere Tabelle, die sogenannte *Rcon-Tabelle*, benötigt (Tabelle 9).

01	02	04	08	10	20	40	80	$1b$	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Tabelle 9 - Rcon-Tabelle

Aus dieser Tabelle werden nun die Spalten in aufsteigender Reihenfolge ausgewählt. Für den 1. Rundenschlüssel also Spalte 1 der Tabelle.

Anschließend wird diese Spalte (blau) mit der Spalte die aus *SubBytes* hervorgegangen ist (orange) und der ersten Spalte der Vorgängermatrix (gelb) XOR-verknüpft:

$8a$		$2b$		01	→	$a0$
84	⊕	$7e$	⊕	00		fa
eb		15		00		fe
01		16		00		17

Das Resultat dieser Verknüpfung (grün) bildete die 1. Spalte des aktuellen Rundenschlüssels:

1.Rundenschlüssel

$a0$			
fa			
fe			
17			

Die restlichen Spalten des aktuellen Rundenschlüssels werden durch eine XOR-Verknüpfung der vorrangehenden Spalte mit der entsprechenden Spalte der Vorgängermatrix gebildet. Die 2. Spalte der aktuellen Matrix (gelb) wird also beispielsweise aus der 1. Spalte derselben Matrix (grün) und aus der 2. Spalte der Vorgängermatrix (grau) erzeugt:

$a0$		28		88
fa	$\oplus$	ae	$\rightarrow$	54
fe		$d2$		$2c$
17		$a6$		$b1$

Diesem Muster folgend werden die restlichen Spalten der Matrix aufgefüllt:

1.Rundenschlüssel

$a0$	88	23	$2a$
fa	54	$a3$	$6c$
fe	$2c$	39	76
17	$b1$	39	05

Aus dem 1. Rundenschlüssel werden dann rekursiv alle weiteren Rundenschlüssel erzeugt, bis die Anzahl der Rundenschlüssel der Anzahl der Runden entspricht. Ist dies der Fall, ist die Schlüsselexpansion abgeschlossen.

In den nachfolgenden Funktionen erfolgt die eigentliche Verschlüsselung der Daten. Dabei dient die *State-Matrix* sowohl als Übergabeparameter als auch als Rückgabewert jeder Funktion, wodurch sich deren Inhalt nach jedem Funktionsaufruf ändert. Sie spiegelt also den jeweils aktuellen Status des Verschlüsselungsvorgangs wieder.

3) *AddRoundKey*

In diese Funktion wird der aktuelle Inhalt der *State-Matrix* elementweise mit der jeweiligen Matrix des Rundenschlüssels XOR-verknüpft. Das Ergebnis wird anschließend wieder in der *State-Matrix* gespeichert. Bei dem ersten Aufruf der Funktion (also noch vor dem ersten Rundendurchlauf) beinhaltet die State-Matrix noch die Daten des Klartextblockes und als Schlüssel wird die *Cipher-Key-Matrix* verwendet:

$State - Matrix_1$

19	$a0$	$9a$	$e9$
$3d$	$f4$	$c6$	$f8$
$e3$	$e2$	$8d$	48
be	$2b$	$2a$	08

$=$

$Cipher - Key - Matrix$

$2b$	28	ab	09
$7e$	ae	$f7$	cf
15	$d2$	15	$4f$
15	$a6$	88	$3c$

$\oplus$

$State - Matrix_0$

32	88	31	$e0$
43	$5a$	31	37
$f6$	30	98	07
$a8$	$8d$	$a2$	34

Nach dem ersten Funktionsaufruf von *AddRoundKey* ist die Vorrunde beendet und es startet der erste Rundendurchlauf. Jeder Rundendurchlauf beinhaltet folgende Funktionsaufrufe:

4) *SubBytes*

Diese Funktion wurde schon für die Schlüsselexpansion genutzt. In diesem Fall wird jedoch nicht nur eine einzelne Spalte, sondern der komplette Inhalt der *State-Matrix* durch ein Element der *S-BOX* ersetzt. Auch hier bestimmt wieder der Wert des Inhalts durch welches Element der *S-BOX* es substituiert wird. Durch diese nichtlineare Substitution wird das in Kapitel 2.1.3 vorgestellte Prinzip der Konfusion umgesetzt. Für die erste Runde des Beispiels erfolgt mit der *S-BOX* aus *Tabelle 10* die folgende Transformation:

$State - Matrix_1$

19	$a0$	$9a$	$e9$
$3d$	$f4$	$c6$	$f8$
$e3$	$e2$	$8d$	48
be	$2b$	$2a$	08

→

$State - Matrix_2$

$d4$	$e0$	$b8$	$1e$
27	bf	$b4$	41
11	98	$5d$	52
ae	$f1$	$e5$	30

5) *ShiftRows*

Bei dieser Operation wird jede einzelne Zeile entsprechend ihrer Position nach links rotiert: Die erste Zeile wird um null Positionen verschoben, die zweite Zeile um eine Position, die Dritte um zwei und die Vierte um drei Positionen. Ziel dieser Rotation ist es, die Forderung nach Diffusion (siehe Kapitel 2.1.3) zu erfüllen.

$State - Matrix_2$

$d4$	$e0$	$b8$	$1e$
27	bf	$b4$	41
11	98	$5d$	52
ae	$f1$	$e5$	30

→

$State - Matrix_3$

$d4$	$e0$	$b8$	$1e$
bf	$b4$	41	27
$5d$	52	11	98
30	ae	$f1$	$e5$

6) *MixColumns*

Für die Operation dieser Funktion ist eine spezielle Form der Multiplikation notwendig. Zur Notation dieser Multiplikation wird der Operator „⊙“ eingeführt. Dieser ist wie folgt definiert:

$$2 \odot B := \begin{cases} B \ll 1, & wenn\ B < 128 \\ (B \ll 1) \oplus 0x1b, & wenn\ B \geq 128 \end{cases}$$
$$3 \odot B := (2 \odot B) \oplus B \qquad (19)$$

Wobei B ein Byte darstellt und die Operation $B \ll 1$ einem Linksshift um ein Bit entspricht. Diese Definition schließt zwar nur die Multiplikation mit einer 2 beziehungsweise einer 3 mit ein, dies ist jedoch für nötigen Berechnung in dieser Funktion ausreichend.

Mehr Informationen zum mathematischen Hintergrund dieser Operation werden in Kapitel 3.1.3 beschrieben.

Die *State-Matrix* wird in der Funktion spaltenweise betrachtet. Jede Spalte wird einzeln mit *Rijndael*s sogenannter *Galois-Field-Matrix* multipliziert:

		Galois-Field-Matrix					
$a_0{}'$		02	03	01	01		a_0
$a_1{}'$	=	01	02	03	01	⊙	a_1
$a_2{}'$		01	01	02	03		a_2
$a_3{}'$		03	01	01	02		a_3

Dies kann mit folgendem Gleichungssystem gelöst werden:

$$a_0' = (2 \odot a_0) \oplus (3 \odot a_1) \oplus a_2 \oplus a_3$$
$$a_1' = a_0 \oplus (2 \odot a_1) \oplus (3 \odot a_2) \oplus a_3$$
$$a_2' = a_0 \oplus a_1 \oplus (2 \odot a_2) \oplus (3 \odot a_3)$$
$$a_3' = (3 \odot a_0) \oplus a_1 \oplus a_2 \oplus (2 \odot a_3)$$

Durch diese Operationen werden die einzelnen Spalten der Matrix durchmischt. Auch dies soll die Diffusion des Systems erhöhen. Für das Beispiel folgt aus der Operation folgendes Ergebnis:

$State - Matrix_3$					$State - Matrix_4$			
$d4$	$e0$	$b8$	$1e$		04	$e0$	48	28
bf	$b4$	41	27	→	66	cb	$f8$	06
$5d$	52	11	98		81	19	$d3$	26
30	ae	$f1$	$e5$		$e5$	$9a$	$7a$	$4c$

7) *AddRoundKey*

Nachdem die Spalten und Zeilen der *State-Matrix* durch die Funktionen 5) und 6) durchmischt wurden, erfolgt ein weiterer Aufruf von *AddRoundKey*. Hier wird nun der Inhalt der *State-Matrix* mit dem aktuellen Rundenschlüssel verknüpft. Für die erste Runde des Beispiels ergibt sich also:

$State - Matrix_5$

$a4$	68	$6b$	02
$9c$	$9f$	$5b$	$6a$
$7f$	35	ea	50
$f2$	$2b$	43	49

$=$

1. $Rundenschlüssel$

$a0$	88	23	$2a$
fa	54	$a3$	$6c$
fe	$2c$	39	76
17	$b1$	39	05

$\oplus$

$State - Matrix_4$

04	$e0$	48	28
66	cb	$f8$	06
81	19	$d3$	26
$e5$	$9a$	$7a$	$4c$

Anschließend werden die Schritte 4) bis 7) $n-1$ mal wiederholt, wobei n für die Rundenanzahl steht (für dieses Beispiel gilt: $n = 10$). In der Schlussrunde (Runde$= n$) fällt der Funktionsaufruf von *MixColumns* weg, die *State-Matrix* durchläuft also nur folgende drei Funktionen:

8) *SubBytes*
9) *ShiftRows*
10) *AddRoundKey*

Nach dem letzten Funktionsaufruf von *AddRoundKey* ist der Verschlüsselungsvorgang abgeschlossen. Die *State-Matrix* enthält nun den endgültigen Chiffretext. Für dieses Beispiel hat die Matrix nach vollendeter Verschlüsselung folgenden Inhalt[10]:

$State - Matrix$

39	02	dc	19
25	dc	11	$6a$
84	09	85	$0b$
$1d$	fb	97	32

Somit wurde durch die Verschlüsselung der Klartext M_i

$$M_i = \{32|43|f6|a8|88|5a|30|8d|31|31|98|a2|e4|37|07|34\}$$

in folgenden Chiffretext C_j überführt:

$$C_j = \{39|25|84|1d|02|dc|09|fb|dc|11|85|97|19|6a|0b|32\}$$

[10] Zur Berechnung der Werte wurde ein C-Programm entwickelt, dessen Quellcode sich im Anhang B befindet.

Um die Diffusion des Algorithmus zu zeigen, kann nun die Originalnachricht an einer Stelle leicht verändert werden (rot hervorgehoben):

$$M'_i = \{32|43|f6|a8|88|5a|30|\mathbf{8e}|31|31|98|a2|e4|37|07|34\}$$

Aus dieser geringfügigen Veränderung des Klartextes resultiert nun auf Grund der Diffusion ein völlig veränderter Chiffretext C'_j:

$$C'_j = \{1e|91|01|d1|f0|b5|eb|9e|67|4f|ec|89|30|de|8a|65\}$$

3.1.2 Entschlüsselung

Um eine mit dem *AES*-Verfahren verschlüsselte Nachricht zu entschlüsseln, muss diese Nachricht die gleichen Funktionsaufrufe durchlaufen, wie bei der Verschlüsselung, jedoch in umgekehrter Reihenfolge. Ein Flussdiagramm des Entschlüsselungsvorgangs zeigt Abbildung 23.

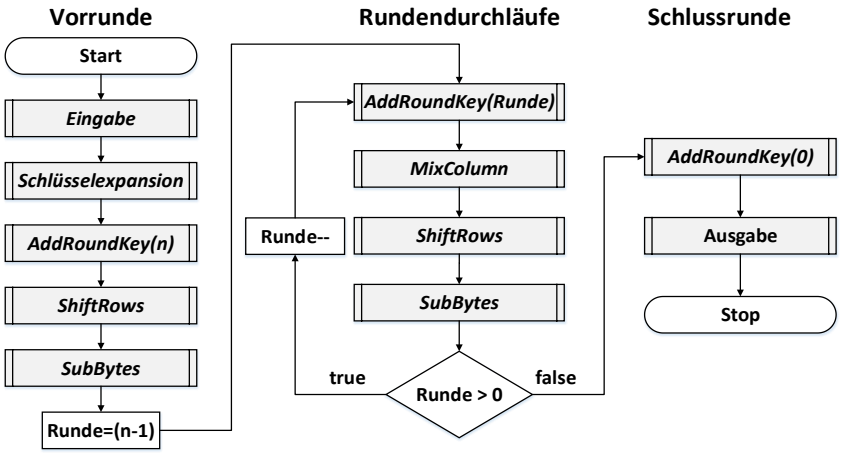


Abbildung 23 - AES-Entschlüsselungsalgorithmus

Außerdem müssen die Funktionen invertiert werden, um die Operationen während der Verschlüsselung wieder rückgängig zu machen.

Die Invertierung der einzelnen Funktionen geschieht folgendermaßen:

- *SubBytes*
  Hier muss die bei der Verschlüsselung durchgeführte Substitution mit den Werten aus der *S-BOX* rückgängig gemacht werden. Das heißt, wenn für die *S-BOX* beispielweise gilt:

$$S(0\text{x}53) = 0\text{xed}$$

  muss für die inverse *S-BOX* gelten:

$$S^{-1}(0\text{xed}) = 0x53$$

- *ShiftRows*
  In dieser Funktion werden die Zeilen um die entsprechenden Elemente nach rechts rotiert, anstatt, wie zuvor, nach links.

- *MixColumns*
  Um die Durchmischung der einzelnen Spalten rückgängig zu machen, muss das zuvor genutzte Gleichungssystem folgendermaßen geändert werden [18]:

$$\begin{aligned}
a'_0 &= (14 \odot a_0) \oplus (11 \odot a_1) \oplus (13 \odot a_2) \oplus (9 \odot a_3) \\
a'_1 &= (9 \odot a_0) \oplus (14 \odot a_1) \oplus (11 \odot a_2) \oplus (13 \odot a_3) \\
a'_2 &= (13 \odot a_0) \oplus (9 \odot a_1) \oplus (14 \odot a_2) \oplus (11 \odot a_3) \\
a'_3 &= (11 \odot a_0) \oplus (13 \odot a_1) \oplus (9 \odot a_2) \oplus (14 \odot a_3)
\end{aligned}$$

  Da der Operator „$\odot$“ bis jetzt jedoch nur für die Operanten *2* und *3* definiert wurde (19), muss die Definition an dieser Stelle erweitert werden:

$$\begin{aligned}
9 \odot B &:= (8 \odot B) \oplus \text{B} \\
11 \odot B &:= (8 \odot B) \oplus (2 \odot B) \oplus \text{B} \\
13 \odot B &:= (8 \odot B) \oplus (4 \odot B) \oplus \text{B} \\
14 \odot B &:= (8 \odot B) \oplus (4 \odot B) \oplus (2 \odot B)
\end{aligned}$$

  Wobei gilt:

$$4 \odot B := 2 \odot (2 \odot B) \text{ und } 8 \odot B := 2 \odot (4 \odot B)$$

Die Funktion *AddRoundKey* muss nicht angepasst werden, da die dort verwendete XOR-Operation eine selbstinverse Operation ist.

Nach dem letzten Funktionsaufruf beinhaltet die State-Matrix den Inhalt des originalen Klartextblockes, der anschließend wieder in seine ursprüngliche Form gebracht werden kann.

3.1.3 Mathematischer Hintergrund

In dem nachfolgenden Abschnitt wird der mathematische Hintergrund der durchgeführten Operationen kurz beschrieben. Tiefergehende Information sind dem Buch zu entnehmen, welches die *Rijndael*-Erfinder veröffentlicht haben [20].

Zur mathematischen Betrachtung wird der endliche Körper (*Galois-Körper*) $GF(2^8)$ eingeführt. Dieser Köper enthält alle $2^8 = 256$ Elemente, die durch ein Byte B dargestellt werden können. Jedes Byte B der Form:

$$B = \{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7\}(LSB0^{11})$$

kann durch ein Polynom $B(x)$ dargestellt werden:

$$B(x) = b_0 \cdot x^7 + b_1 \cdot x^6 + b_2 \cdot x^5 + b_3 \cdot x^4 + b_4 \cdot x^3 + b_5 \cdot x^2 + b_6 \cdot x^1 + b_7$$

Zur Multiplikation zweier Bytes in dem Körper $GF(2^8)$ wird ein irreduzibles Polynom[12] 8. Grades benötigt. Dieses Polynom wurde von den *Rijndael*-Erfinder wie folgt festgelegt:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

Die Multiplikation erfolgt dann durch das Multiplizieren beider Polynome Modulo $m(x)$. Für diese Art der Multiplikation wird der bereits eingeführter Operator „$\odot$“ verwendet. Hier wird er vollständig definiert:

$$B_1 \odot B_2 \coloneqq \big(B_1(x) \cdot B_2(x)\big) \bmod \big(m(x)\big) \tag{20}$$

Das Polynom $m(x)$ wurde von den Entwicklern so gewählt, dass bei einer Multiplikation mit einer 2 oder einer 3 die Berechnung, wie bereits beschrieben, abgekürzt werden kann. So kann bei der *MixColumns*-Funktion auf die aufwändige Berechnung der Polynom-Modulo-Operation verzichtet werden.

[11] LSB0 gibt an, dass das niederwertigste Bit zuerst dargestellt wird.
[12] Ein irreduzibles Polynom ist ein Polynom, welches sich nicht weiter in „einfachere“ Polynome zerlegen lässt.

S-Box

Die Werte der *S-Box* werden ebenfalls durch die für $GF(2^8)$ definierte Multiplikation berechnet, wobei das Byte B als Spaltenvektor betrachtet wird:

$$S(B) = \bar{A} \odot B^{-1} \oplus 01100011 \tag{21}$$

Dabei entspricht B^{-1} dem inversen Element von B in $GF(2^8)$, sodass gilt:

$$B \odot B^{-1} = 1$$

Durch die Invertierung des Eingabeparameters wird die Forderung nach einem nichtlinearen Zusammenhang zwischen $S(B)$ und B erfüllt.

Der Parameter $\bar{A}$ ist eine Matrix mit folgendem Inhalt:

$$\bar{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Da durch die Invertierung die Berechnung von $S(B)$ recht aufwändig ist, wird die S-BOX in der Regel durch eine *Lookup-Tabelle* (LUT) implementiert. Für diese Tabelle werden für alle möglichen Werte von B (0 … 255) der entsprechende Wert von $S(B)$ vorausberechnet.

Eine *S-BOX*-LUT zeigt Tabelle 10. Dabei wird angenommen, dass sich das Byte B aus den zwei 4-Bit-Werten x und y zusammensetzt, welche dann Spalte und Zeile des Tabellenwerts bestimmen.

		x															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
y	**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*	*06*	*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*

Tabelle 10 - S-BOX-LUT (Werte sind im Hexadezimalformat angegeben) [18]

Für $B = 0x53$ würde der entsprechende Tabelleneintrag also $S(0x53) = 0xed$ lauten (5. Zeile, 3. Spalte). Berechnen lässt sich dieser Wert nach Gleichung (21) wie folgt:

$$B = 0x53 = 1100101\ (LSB0) \rightarrow B^{-1} = 0xca = 01010011\ (LSB0)$$

$$S(0x53) = \left(\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \right) \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = 0xed$$

3.1.4 Sicherheit des AES

Der *AES* ist kein absolut sicheres Verfahren nach den Kriterien von Shannon (siehe Kapitel 2.1.3). Es gilt jedoch zurzeit als praktisch sicher. Alle bisher durchgeführten Angriffe konnten nur bei *AES*-Verfahren mit reduzierter Rundenzahl erfolgreich angewandt werden. Somit kann der Algorithmus zum jetzigen Zeitpunkt theoretisch nur mit einem *brute-force-Angriff* gebrochen werden, was jedoch schon bei einer Schlüssellänge von 128-Bit als aussichtslos gilt, da der zeitliche Aufwand deutlich zu hoch wäre (siehe Kapitel 2.1.2).

Dennoch gibt es Kritikpunkte bei diesem Verfahren. Einer dieser Kritikpunkte ist der geringe Sicherheitspuffer. Wie schon erwähnt wurde, konnte das Verfahren mit reduzierter Rundenzahl bereits gebrochen werden. Bei einer Schlüssellänge von 128-Bit liegt diese reduzierte Rundenzahl aktuell bei 7, woraus ein Sicherheitspuffer von lediglich 3 Runden resultiert (bei einem 192-Bit-Schlüsseln liegt der Puffer bei 4 und bei einem 256-Bit-Schlüssel bei 5 Runden). Dies könnte sich zukünftig als zu gering erweisen.

Ein weiterer Kritikpunkt besteht darin, dass sich das Verfahren vergleichsweise gut als algebraische Formel darstellen lässt. Diese Formel ist zwar sehr komplex (bei einer Schlüssellänge von 128-Bit enthält die Formel bereits $\sim 2^{50}$ Bestandteile [18]) doch würde es eines Tages gelingen, diese Formel nach dem Schlüssel aufzulösen, würde bereits ein Klartext-Chiffretext-Paar ausreichen, um den Schlüssel zu ermitteln. Dies ist bisher jedoch noch nicht gelungen.

3.2 One-Time-Pad

Die *One-Time-Pad-Verschlüsselung* ist eine Stromchiffre und wurde 1917 von M. J. Mauborgne und G. Vernan während ihrer Beschäftigung bei AT&T[13] entwickelt [21]. Es gilt zurzeit als einziges Verfahren, welches eine absolute Sicherheit nach der Definition von Shannon gewährleistet (siehe Kapitel 2.1.3). Diese Tatsache impliziert, dass der verwendete Schlüssel mindestens so lang wie die Nachricht sein muss. Außerdem darf jeder Schlüssel nur einmal verwendet werden und muss stochastisch unabhängig von anderen Schlüsseln sein.

[13] Die Firma AT&T Inc. ist ein nordamerikanischer Telekommunikationskonzern.

Die Ver- und Entschlüsselung erfolgt durch eine *Addition-modulo-n* Operation, wobei n die Anzahl der Elemente des Alphabetes angibt:

$$D(K_i, M_i) = E(K_i, M_i) = (K_i + M_i) \bmod(n)$$

Für digitale Nachrichten, bei denen das Alphabet aus den zwei Elementen 0 und 1 besteht, entspricht dies also einer *Addition-modulo-2* Operation, die durch eine einfache XOR-Verknüpfung realisiert werden kann:

$$(0 + 0) \bmod(2) = 0 \rightarrow 0 \oplus 0 = 0$$
$$(0 + 1) \bmod(2) = 1 \rightarrow 0 \oplus 1 = 1$$
$$(1 + 0) \bmod(2) = 1 \rightarrow 1 \oplus 0 = 1$$
$$(1 + 1) \bmod(2) = 0 \rightarrow 0 \oplus 0 = 0$$

Durch diese Verknüpfung kann ein bestimmter Klartext M_i in jeden beliebigen Chiffretext aus dem Chiffreraum C transferiert werden. Im Umkehrschluss bedeutet dies, dass jeder empfangene Chiffretext C_j aus jedem möglichen Klartext aus dem Nachrichtenraum M hervorgegangen sein kann. Ein Angreifer hat keine Möglichkeit festzustellen, welcher Klartext dem Chiffretext zugrunde liegt, sofern er keinen Hinweis auf den verwendeten Schlüssel hat. Somit ist selbst ein *Brute-Force-Angriff* bei diesem Verfahren wirkungslos.

Das Problem bei diesem Verfahren liegt jedoch in der Verteilung der Schlüssel. Eine Möglichkeit besteht darin, zwei identische Schlüsselbücher zu erstellen (heutzutage können auch CD-ROMs oder andere digitale Datenträger verwendet werden) und diese mit zufällig generierten Schlüsselfolgen zu füllen. Diese Schlüsselbücher werden dann unter den Teilnehmern über einen sicheren Kanal verteilt.
Für diesen sicheren Kanal wird häufig ein Bote verwendet, was jedoch wieder einen gewissen Sicherheitsverlust darstellt. Ein Bote kann beispielsweise durch Bestechung, Erpressung oder Folter vom Gegner dazu gebracht werde, die geheimen Schlüsselfolgen zu manipulieren oder herauszugeben. Somit ist die absolute Sicherheit, die das Verfahren theoretisch besitzt, in der Praxis nur schwer aufrechtzuerhalten. Wie das Verfahren in dieser Arbeit umgesetzt wird, beschreibt Kapitel 4.2.2

3.3 Betriebsmodi

In diesem Kapitel werden zwei Betriebsmodi beschrieben, mit denen eine Blockchiffre in eine Stromchiffre überführt werden kann. Die einfachste Art dies zu tun besteht darin, eine Klartextnachricht in feste Blöcke aufzuteilen und diese anschließend nacheinander zu verschlüsseln. Die daraus resultierenden Chiffretextblöcke werden dann vom Empfänger der Nachricht einzeln entschlüsselt und die Klartextnachricht anschließend wieder aus den einzelnen Blöcken zusammengefügt. Sollte die Blocklänge kein ganzzahliger Teiler der Nachrichtenlänge sein, kann der letzte Klartextblock mit Zufallszahlen aufgefüllt werden. Dieses Verfahren wird als *Electronic Codebook* (ECB) bezeichnet [22]. Eine schematische Darstellung des ECB-Verfahrens zeigt Abbildung 24.

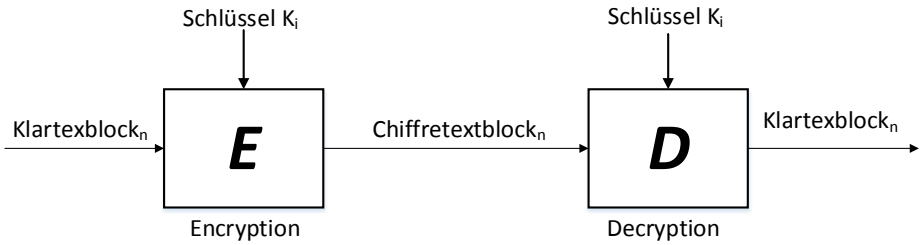


Abbildung 24 - ECB-Betriebsmodus

Der Nachteil dieser einfachen Methode besteht darin, dass bei deterministischen Verschlüsselungsverfahren gleiche Klartextblöcke auf immer gleiche Chiffretextblöcke abgebildet werden. Dadurch übertragen sich Muster im Klartext auch auf den Chiffretext, was das Verfahren anfällig gegen statistische Angriffe macht. Das Prinzip der Diffusion, das solche Angriffe erschweren soll, wird durch eine Blockchiffre wie der *AES* nur innerhalb eines Blockes umgesetzt, nicht aber zwischen den einzelnen Blöcken.

Daher empfiehlt es sich, einen etwas komplexeren Betriebsmodus wie das *Propagating Cipher Block Chaining* (PCBC) zu nutzen. Bei diesem Verfahren wird der aktuelle Klartextblock vor der eigentlichen Verschlüsselung sowohl mit dem vorherigen Klartextblock als auch mit dem vorherigen Chiffretextblock verknüpft. Dafür kann beispielsweise eine XOR-Verknüpfung gewählt werden. Eine schematische Darstellung eines solchen *PCBC*-Verfahrens zeigt Abbildung 25.

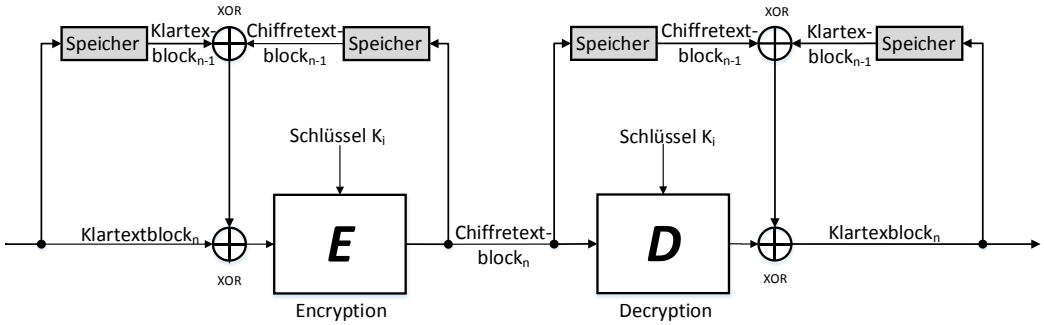


Abbildung 25 - PCBC-Betriebsmodus

Durch die Rückführung des Chiffretextblockes wird erreicht, dass eventuell vorhandene Muster im nachfolgenden Chiffretextblock „verwischt“ werden. Die Miteinbeziehung des vorherigen Klartextblockes führt hingegen dazu, dass sich Fehler bei der Übertragung durch sämtliche Klartextblöcke fortpflanzen. Dadurch entsteht ein vollständig synchrones Verschlüsselungsverfahren, das auch zur Authentifizierung einer Nachricht verwendet werden kann. Dazu muss beispielsweise nur ein festes Bitmuster am Ende einer Nachricht hinzugefügt werden. Tritt nach der Entschlüsselung dieses Bitmuster wieder unverändert auf, so kann der Empfänger der Nachricht davon ausgehen, dass diese während der Übertragung nicht gestört oder manipuliert worden ist. Da der letzte Chiffretextblock von den vorherigen Verschlüsselungsvorgängen abhängig ist, lässt sich ein festes Bitmuster am Ende einer Nachricht nicht im Chiffretext wiedererkennen. Um den ersten Klartextblock zu verschlüsseln, wird statt des Vorgängerblocks (der für den ersten Klartextblock nicht existiert) ein zufällig gewählter Initialvektor verwendet.

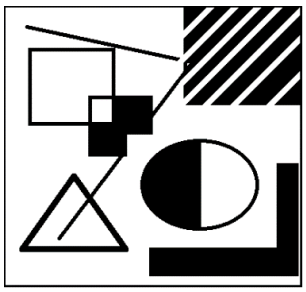
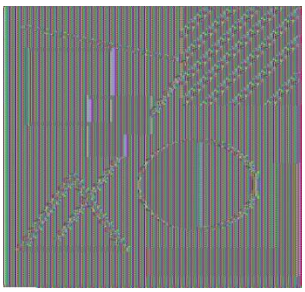
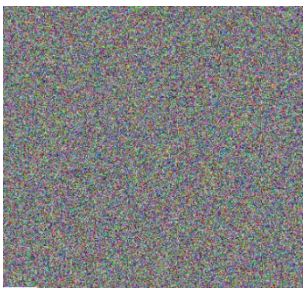
		

Abbildung 26 - Originalbilddaten	*Abbildung 27 - ECB-Verschlüsselung*	*Abbildung 28 - PCBC-Verschlüsselung*

Tabelle 11 - Vergleich zwischen dem ECB- und PCBC-Betriebsmodus[14]

[14] Die Originalbitmap sowie ein C-Programm zur Verschlüsselung der Daten befinden sind im Anhang B.

Den Vorteil von *PCBC* gegenüber *ECB* zeigt Tabelle 11. Hier wurde eine Nachricht mit klaren Strukturen in beiden Betriebsmodi verschlüsselt[15]. Als Nachricht dienten die Bilddaten einer Bitmap-Datei, die in *Abbildung 26* dargestellt sind. Während die Strukturen der originalen Daten noch relativ deutlich in dem Ergebnis der ECB-Verschlüsselung zu erkennen sind (*Abbildung 27*), sind die Bilddaten nach der PCBC-Verschlüsselung vollständig verrauscht (*Abbildung 28*).

3.4 Steganographischer Algorithmus

In Kapitel 2.2 wurden bereits zwei Algorithmen zur Einbettung von Nachrichten in Containerbildern vorgestellt. Es wurde auch erläutert, dass die einfache LSB-Methode, bei der die LSBs der einzelnen Pixel eines Containerbildes durch die entsprechenden Bits der Nachricht ersetzt werden, anfällig für statistische Angriffe wie beispielweise der *chi-square-Angriff* ist. Daher wird für das in dieser Arbeit beschriebene System die andere Variante gewählt. Den Ablauf dieses Verfahrens zeigt das Flussdiagramm in Abbildung 29.

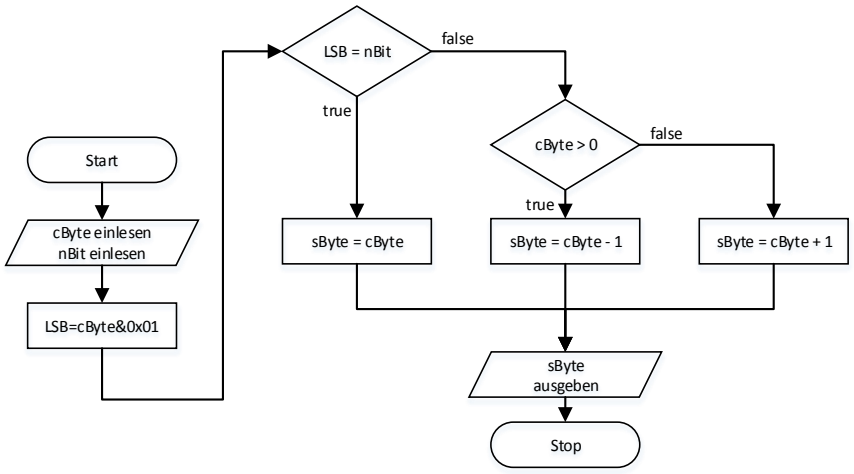


Abbildung 29 - Steganographischer Algorithmus

Hier wird zunächst das aktuelle Byte des Containers (*cByte*) und das aktuelle Nachrichtenbit (*nBit*) eingelesen. Anschließend wird das niederwertigste Bit aus dem Containerbyte extrahiert (*LSB*) und mit dem Nachrichtenbit verglichen. Stimmen beide Bits überein, so wird das Containerbyte direkt als Steganobyte (*sByte*) übernommen. Ist dies nicht der Fall, so wird zunächst geprüft, ob das Containerbyte größer 0 ist. Trifft dies zu, wird das Containerbyte um eins dekrementiert und anschließend dem Steganobyte zugewiesen. Ist diese Bedingung nicht erfüllt, so wird das Containerbyte

[15] Zum Verschlüsseln der einzelnen Blöcke wurde in beiden Fällen das *AES*-Verfahren verwendet.

vor der Zuweisung um eins inkrementiert. Anschließend wird das Steganobyte ausgegeben.

3.5 Pseudozufallsgeneratoren

In Kapitel 2.1 wurde bereits erwähnt, dass bei Verwendungen von Stromchiffren pseudozufällige Schlüsselfolgen verwendet werden, um die einzelnen Elemente des zugrundeliegenden Alphabets zu verschlüsseln. Pseudozufällige Zahlenfolgen zeichnen sich dadurch aus, dass sie zwar alle stochastischen Merkmale echter Zufallsfolgen aufweisen (beispielsweise eine Gleichverteilung von Einsen und Nullen, bei genügend langen Folgen), jedoch deterministisch sind. Das heißt, wird ein Pseudozufallsgenerator mit dem immer gleichen Wert initialisiert, so liefert er auch immer gleiche Zufallsfolgen. Nicht deterministische Zufallsfolgen eignen sich in der Regel nicht für kryptographische Anwendungen, da der Empfänger die Zufallsfolge nicht rekonstruieren kann, um die Nachricht zu entschlüsseln.

Pseudozufallsgeneratoren werden in der Regel durch endliche Automaten realisiert. Jeder Zustand repräsentiert dabei einen Zahlenwert der Zufallsfolge. Da ein endlicher Automat aber nur eine begrenzte Anzahl von Zuständen annehmen kann, bedeutet dies, dass sich die Zufallsfolge nach einem gewissen Zyklus wiederholt und somit die erzeugte Zahlenfolge periodisch ist. Bei der Realisierung von Pseudozufallsgeneratoren muss also darauf geachtet werden, dass die Periode der erzeugten Zahlenfolge möglichst groß ist, um einem Angreifer die Vorhersage der Zufallszahlen zu erschweren.

Eine Möglichkeit einen endlichen Automaten zur Erzeugung von pseudozufälligen Zahlenfolgen zu realisieren, sind linear rückgekoppelte Schieberegister. Diese bestehen aus einer Reihe seriell angeordneter Speicherzellen (bei binären Zufallsfolgen können dafür Flipflops[16] verwendet werden). Bei jedem Takt übernimmt jede Speicherstelle den Wert der vorherigen Speicherstelle. Der Wert für die erste Speicherstelle wird aus einer Rückkopplungsfunktion gewonnen. Das Blockschaltbild eines solchen rückgekoppelten Schieberegisters zeigt Abbildung 30.

[16] Flipflops sind elektronische Speicherstellen, mit denen ein Bit über einen längeren Zeitraum gespeichert werden kann.

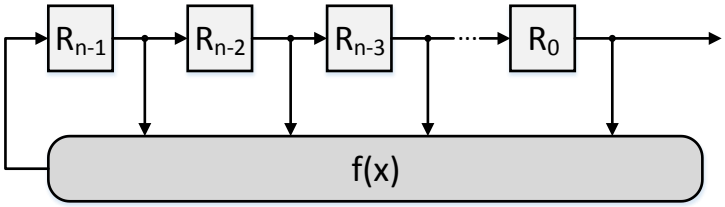


Abbildung 30 - Rückgekoppeltes Schieberegister

Wird für $f(x)$ eine lineare Rückkopplungsfunktion verwendet, wird dies als linearer Pseudozufallsgenerator bezeichnet, ansonsten als nichtlinearer Pseudozufallsgenerator. Eine lineare Rückkopplungsfunktion kann als Polynom der Form

$$f(X) = X^n + \sum_{i=0}^{n-1} c_i \cdot X^i$$

dargestellt werden, wobei n der Länge des Schieberegisters entspricht. Dabei gibt der Koeffizient c_i an, ob der Wert einer Speicherstelle verwendet wird ($c_i = 1$) oder nicht ($c_i = 0$). Soll eine binäre Zahlenfolge erzeugt werden, wird statt der Addition die XOR-Verknüpfung verwendet. Ein Schieberegister mit dem Rückkopplungspolynom $f(X) = X^4 + X^2 + 1$ ist beispielsweise auf Abbildung 31 dargestellt.

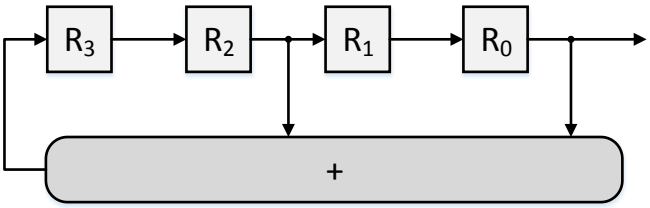


Abbildung 31 - Linear rückgekoppeltes Schieberegister mit f(X)=X^4+X^2+1

Die Periodenlänge einer so produzierten Zahlenfolge hängt von der Wahl des Rückkopplungspolynoms ab. Die maximale Periodenlänge von $2^n - 1$ wird dann erreicht, wenn für $f(X)$ ein primitives Polynom verwendet wird [2]. Ein primitives Polynom ist ein irreduzibles (also nicht mehr zerlegbares) Polynom, dessen Exponent den maximal möglichen Wert annimmt. Eine Auswahl primitiver Polynome unterschiedlichen Grades ist in [4] aufgeführt.

Ein Schieberegister mit $n = 128$ hätte demnach eine Periodendauer von $2^{128} - 1 = 3{,}4 \cdot 10^{38}$, bei einer Taktfrequenz von 100 MHz entspricht dies einer Länge von $\sim 10 \cdot 10^{22}$ Jahren. Auf Grund der Linearität vom $f(X)$ muss ein Angreifer jedoch nicht eine komplette Periodenlänge abwarten, um die einzelnen Glieder der Zufallsfolge vorherbestimmen zu können. Mit dem sogenannten *Berlekamp-Massy-Algorithmus*, der

in [2] näher beschrieben wird, ist es beispielsweise möglich, die Koeffizienten des Rückkopplungspolynoms bereits nach $2 \cdot n$ Gliedern zu bestimmen. Sind die Koeffizienten bekannt, können auch sämtliche anderen Glieder der Zufallsfolge vorhergesagt werden. Dies stellt eine große Schwäche von linearen Zufallsgeneratoren dar.

Eine einfache Lösung dieses Problems wäre es, nichtlineare Funktionen für die Rückkopplung zu verwenden. Solche nichtlinearen Zufallsgeneratoren sind jedoch mathematisch nur sehr schwer zu analysieren und werden daher in der Praxis nur selten verwendet, da ihre Sicherheit nicht ausreichend gut überprüft werden kann. Stattdessen werden häufig lineare Schieberegister verwendet, dessen Ausgangswerte durch eine nichtlineare Operation verknüpft werden. Ein Beispiel eines solchen nichtlinearen Zufallsgenerators ist der *Multiplexer-Generator*, der auf Abbildung 32 dargestellt ist.

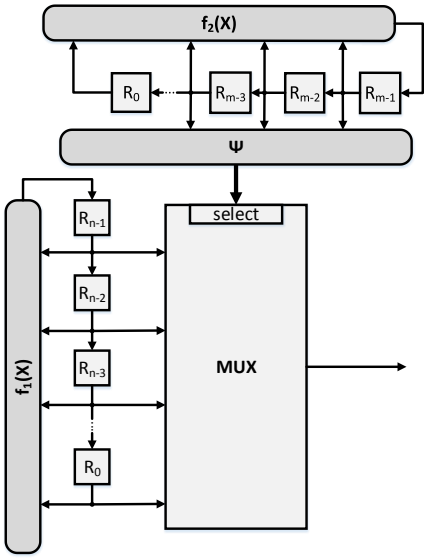


Abbildung 32 - Blockschaltbild eines Multiplexer-Generators

Bei dem *Multiplexer-Generator* wird nicht, wie zuvor, der Inhalt einer festen Speicherstelle ausgegeben. Stattdessen wird die Speicherstelle, deren Wert das nächste Glied in der Zahlenfolge bestimmt, durch einen Multiplexer (*MUX*) ausgewählt. Die Adressierung des Multiplexers erfolgt durch ein weiteres linear rückgekoppeltes Schieberegister und einer Auswahlfunktion Ψ. Durch dieses Verfahren wird zum einen die Periodenlänge auf $(2^n - 1) \cdot (2^m - 1)$ erhöht. Zum anderen steigt die lineare Komplexität des Generators und somit auch der Aufwand zur Bestimmung der genutzten Koeffizienten deutlich [2]. Ein solcher Zufallsgenerator wird auch in dem hier beschriebenen System verwendet (siehe Kapitel 4.1)

4 Systembeschreibung

In diesem Kapitel erfolgt eine detaillierte Beschreibung des Systems, welches im Rahmen dieser Arbeit entwickelt wurde. Das System soll mit Hilfe von steganographischen und kryptographischen Methoden einen sicheren Nachrichtenaustausch mittels HTTP- oder FTP-Protokoll zwischen mehreren Teilnehmern ermöglichen. Als Nachricht kann eine Datei mit beliebigem Dateiformat verwendet werden.

Sowohl die steganographischen als auch die kryptographischen Methoden werden jeweils in einem zweistufigen Verfahren umgesetzt. Zunächst wird ein Container aus Zufallszahlen generiert. In diesen Container wird in unregelmäßigen Abständen die Nachricht als Klartext eingebettet (Das Ergebnis wird im Folgenden als Zufallssteganogramm bezeichnet). Die Abstände werden durch den steganographischen Schlüssel bestimmt (siehe Kapitel 4.2.3). Anschließend durchläuft das so erzeugte Zufallssteganogramm einen zweistufigen Verschlüsselungsvorgang. Dieser beinhaltet eine *AES*- und eine *One-Time-Pad-Verschlüsselung* (siehe Kapitel 3.1 und 3.2). Im letzten Schritt werden die verschlüsselten Daten in einen weiteren Container, den Bilddaten einer USB Kamera, mit dem im Kapitel 3.4 beschriebenen Algorithmus eingebettet. Die einzelnen Stufen, die eine Nachricht durchläuft, zeigt Abbildung 33.

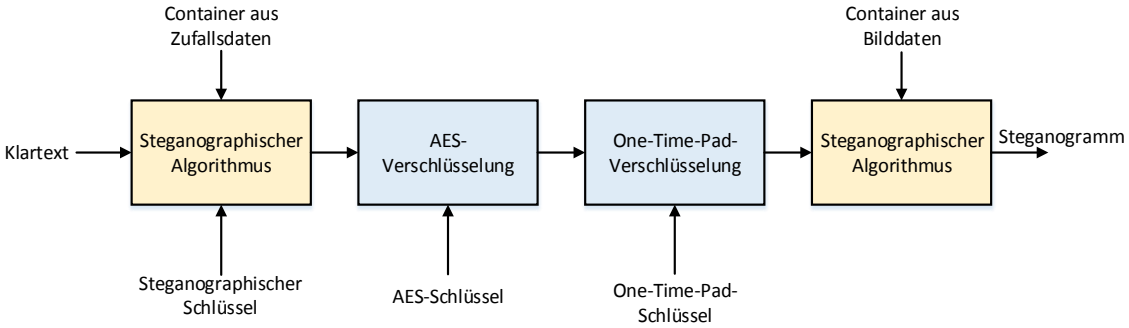


Abbildung 33 - Stufen zur Sicherung der Klartextnachrichten

Die erste Stufe, in der die Nachricht in einen Datenstrom aus Zufallszahlen eingebettet wird, ist in Software auf einer CPU umgesetzt. Sie dient dazu, einen *Brute-Force-Angriff* auf den *AES*-Algorithmus zu erschweren. Die restlichen Stufen werden als Hardwarekomponente auf einem FPGA realisiert.

Das System soll außerdem die in Kapitel 2.1 beschriebenen und von Kerckhoff formulierten Anforderungen an ein kryptographisches System erfüllen:

1) ***Das System sollte weder physikalisch noch mathematisch entschlüsselbar sein.***
   Diese Anforderung kann in der Praxis nur schwer vollständig erfüllt werden (siehe Kapitel 2.1.3). Um sich der Anforderung zumindest anzunähern, wird ein standardisiertes Verschlüsselungsverfahren (*AES*) mit einem Verfahren kombiniert, welches auf dem Prinzip der „*Security through Obscurity*“ beruht. Dies soll unter anderem das Anwenden von standardisierten Angriffen erschweren.
2) ***Es sollte keine Geheimhaltung erfordern und sollte auch vom Feind gestohlen werden dürfen.***
   Diese Anforderung wird erfüllt, in dem mit dem *AES*-Algorithmus unter anderem ein standardisiertes Verschlüsselungsverfahren verwendet wird. Dessen Sicherheit ist allein von dem verwendeten Schlüssel abhängig.
3) ***Der Schlüssel sollte leicht zu kommunizieren sein und er sollte nicht notiert werden müssen. Außerdem sollte er vom Anwender änderbar sein.***
   Die Schlüssel werden durch eine zentrale Schlüsselverwaltung vergeben. Somit können sie zwar nicht direkt vom Anwender verändert werden, müssen aber auch nicht notiert werden, da sie das System nicht verlassen. Mehr Details zur Schlüsselvergabe werden in Kapitel 4.2 beschrieben.
4) ***Das System sollte mit einer telegraphischen Kommunikation kompatibel sein.***
   Dies ist durch die Verwendung des HTTP-, bzw. FTP-Protokolls gewährleistet.
5) ***Es sollte tragbar sein und seine Verwendung nicht die Unterstützung mehrerer Personen erfordern.***
   Dies wird erfüllt, in dem das System im Wesentlichen als ein eingebettetes System auf einem Chip (*SoC*)[17] realisiert wird. Eine genauere Beschreibung der Hardwareplattform erfolgt in Kapitel 5.1.
6) ***Das System sollte leicht zu bedienen sein.***
   Diese Anforderung wird durch eine einfach gehaltene Menü-Struktur realisiert, die in Kapitel 4.3 beschrieben wird.

Jedes System beinhaltet eine Server- und eine Clientfunktionalität. Als Client verbindet sich das System zyklisch mit den Servern der anderen Systeme im Netz und tastet so die Steganogramme der anderen Teilnehmer ab. Erkennt es in diesen Steganogrammen eine Nachricht, die mit dem eigenen persönlichen *AES*-Schlüssel verschlüsselt wurde, speichert das System die Nachricht, sodass sie von dem Benutzer gelesen werden kann. Möchte der Benutzer selbst eine Nachricht an einen anderen Teilnehmer verschicken, so wird diese mit dessen persönlichen *AES*-Schlüssel verschlüsselt und als

[17] SoC (*system on a chip*) bezeichnet ein System, dessen Hauptfunktionalitäten auf nur einem Chip integriert sind.

Steganogramm auf dem eigenen systeminternen Server gespeichert. Von dort kann das Steganogramm vom Zielsystem abgetastet und mit dem persönlichen *AES*-Schlüssel entschlüsselt werden. Somit bestimmt die Verteilung der Serveradressen und der persönlichen *AES*-Schlüssel die Verbindungsmöglichkeiten der Teilnehmer innerhalb eines Netzes. Besitzt jeder Teilnehmer sämtliche *AES*-Schlüssel und Server-Adressen der anderen Teilnehmer, entsteht ein vollständig vermaschtes Netz, wie es beispielsweise auf *Abbildung 34* dargestellt ist.

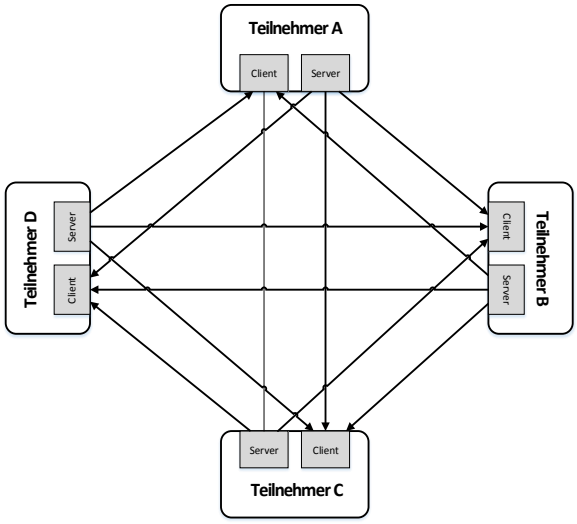


Abbildung 34 - Vollständig vermaschtes Netz mit 4 Teilnehmern

Auf ein komplexeres Übertragungsprotokoll, welches beispielsweise eine automatische Rückmeldung an den Sender einer Nachricht beinhaltet, wurde bewusst verzichtet. Dies könnte eventuell Muster in den übertragenen Daten erzeugen, die von einem Angreifer erkannt und ausgenutzt werden könnten. Eine solche Rückmeldung kann jedoch auch manuell erfolgen, in dem der Empfänger beispielsweise die empfangene Nachricht an den Sender zurück schickt. So kann von diesem überprüft werden, ob die Nachricht korrekt übermittelt wurde.

4.1 Hardware

4.1.1 Verschlüsselung

Die Hardwarekomponenten, mit denen die steganographischen- und kryptographischen Methoden zur Verschlüsselung der Daten in dem System umgesetzt werden, sind auf Abbildung 35 dargestellt. Alle internen Komponenten, die auf der Abbildung gezeigt werden, sind als Hardwarekomponenten auf einem Chip implementiert. Dies erhöht den Schutz vor Manipulation durch Schadenssoftware und erschwert einem Angreifer die genutzten Algorithmen mittels *reverse engineering* zu rekonstruieren [4]. Die beiden CPUs (*CPU0* und *CPU1*) werden zum Ansteuern der externen Komponenten, zur Benutzerinteraktion und zum Einbetten der Nachricht in Zufallsdaten verwendet (*CPU0*), sowie zur Anbindung an ein öffentliches Netz (*CPU1*).

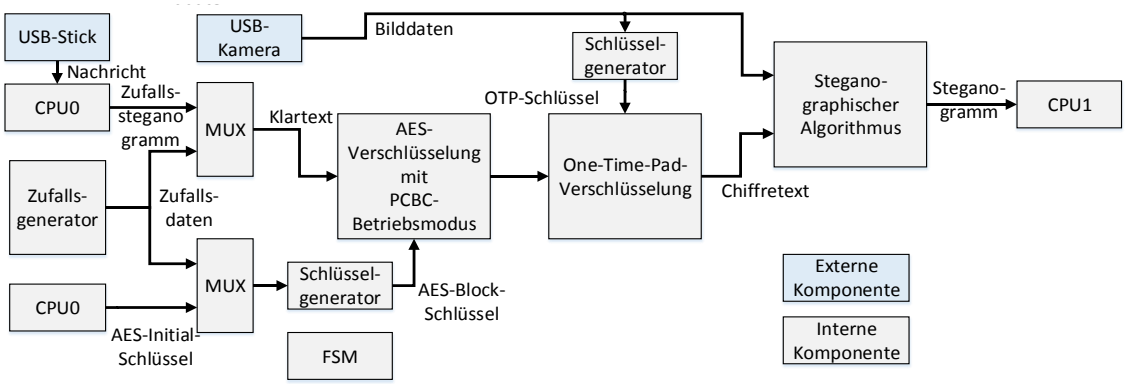


Abbildung 35 – Hardwarekomponente zur Verschlüsslung der Daten

AES-Verschlüsselung

Zur Verschlüsselung der Nachrichten wird das in Kapitel 3.1 beschriebene *AES*-Verfahren genutzt. Dieses Verfahren schreibt eine Blocklänge von 128-Bit vor. Als Schlüssellänge wurde ebenfalls 128-Bit gewählt, sodass die Nachricht und der Schlüssel über den gleichen 128-Bit breiten Datenbus übertragen werden können. Sollte dies den Sicherheitsansprüchen nicht genügen, kann die Schlüssellänge in einer nachfolgenden Version gegebenenfalls noch erweitert werden. Aus dem 128-Bit-Initial-Schlüssel, der von einer zentralen Schlüsselverwaltung vergeben wird, wird von einem Schlüsselgenerator für jeden Klartextblock ein neuer *AES*-Block-Schlüssel generiert. Die *AES*-Blockchiffre wird mit dem im Kapitel 3.3 beschriebenen *PCBC*-Betriebsmodus

in eine Stromchiffre überführt. Mit dem *AES*-Schlüssel kann auch der Empfänger der Nachricht adressiert werden. Theoretisch empfängt zwar jedes System in dem Netzwerk die Nachricht, doch nur das System des Teilnehmers, dessen persönlicher *AES*-Schlüssel zur Verschlüsselung verwendet wurde, kann die Nachricht korrekt entschlüsseln und gegebenenfalls an den Benutzer weiterleiten. Wie die *AES*-Schlüssel unter den Teilnehmer verteilt werden, wird in Kapitel 4.2.1 beschrieben.

One-Time-Pad-Verschlüsselung

Um die Anforderung 1) von Kerckhoff möglichst gut zu erfüllen, werden die Daten nach der *AES*-Verschlüsselung noch mit einer *One-Time-Pad-Verschlüsselung* geschützt. Dies verspricht theoretisch absolute Sicherheit (siehe Kapitel 3.2). Um diese in der Praxis aufrecht zu erhalten, müsste jedoch eine große Anzahl von Schlüsseln, die zudem noch relativ lang sein müssten, unter den Teilnehmer ausgetauscht werden. Dies wiederspricht jedoch der Anforderung 3) von Kerckhoff und ist daher für dieses System, welches einen schnellen und einfachen Nachrichtenaustausch ermöglichen soll, nicht geeignet. Stattdessen wird die Tatsache ausgenutzt, dass die Nachrichten in Form von Steganogrammen versendet werden. Hier wird lediglich ein Bit pro Byte zum übertragen der Nachricht ausgenutzt. Aus den restlichen 7 Bits des Bytes wird ein Bit für die *One-Time-Pad-Verschlüsselung* ausgewählt. Das Auswahlverfahren, mit dem die Schlüsselbits aus dem Steganogramm bestimmt werden, beschreibt Kapitel 4.2.2.

Steganographischer Algorithmus

Nachdem der Inhalt der Nachricht mit kryptographischen Methoden geschützt wurde, wird anschließend ihre Existenz mit Hilfe der in Kapitel 3.4 beschriebenen steganographischen Methode verschleiert. Als Container dienen dabei Bilder von einer USB-Kamera, die direkt mit dem System verbunden ist. Da die Containerbilder also vom System selbst erzeugt werden, wird ein *known-cover-Angriff* (siehe Kapitel 2.2.1) auf das steganographische System deutlich erschwert. Hinzu kommt, dass kontinuierlich Steganogramme erzeugt und unter den Teilnehmern ausgetauscht werden, unabhängig davon, ob eine Nachricht verschickt werden soll oder nicht. Ist einem Angreifer trotz der steganographischen Methoden bekannt, dass verschlüsselte Nachrichten verschickt werden, soll durch den andauernden Datenaustausch verschleiert werden, zu welchen Zeitpunkt dies geschieht. Damit sich das stochastische Profil der einzelnen Steganogramme nicht unterscheidet, werden in jedem Steganogramm verschlüsselte Daten eingebettet. Dabei handelt es sich entweder um eine Nachricht oder um Zufallsdaten, die mit einem zufälligen Schlüssel verschlüsselt werden. Das Verfahren wird außerdem so implementiert, dass die Frequenz, mit der

neue Steganogramme erzeugt werden, unabhängig davon ist, ob Zufallsdaten oder eine Nachricht in dem Steganogramm verschickt wird.

Zufallsgeneratoren

In dem System sind zwei Generatoren für pseudozufällige Zahlenfolgen implementiert. Dafür werden die in Kapitel 3.5 beschriebene Multiplexer-Generatoren verwendet. Der erste Zufallsgenerator wird als Schlüsselgenerator verwendet. Er erzeugt für jeden Block, der mit dem *AES*-Verfahren verschlüsselt wird, einen eigenen Schlüssel. Initialisiert wird der Schlüsselgenerator mit dem persönlichen Kommunikationsschlüssel des Empfängers.

Der zweite Pseudozufallsgenerator erzeugt mit jedem Takt eine 128-Bit große Pseudozufallszahl. Dies geschieht parallel und unabhängig von den anderen Prozessen auf dem System. Der Generator dient unter anderem zur Erzeugung der Zufallsdaten, die anstelle einer Nachricht verschlüsselt und in ein Steganogramm eingebettet werden. Außerdem wird er auch für andere Zufallsereignisse genutzt, wie die zufällige Auswahl einer ID (siehe Kapitel 4.4) oder zur Erzeugung des Initialvektors für den PCBC-Betriebsmodus (siehe Kapitel 3.3). 8 Bit der aktuellen Zufallszahl werden über einen eigenen Port mit *CPU0* verbunden. Dort werden die Ausgangswerte des Generators zur Erzeugung des Zufallscontainers und der Kommunikationsschlüssel verwendet. Letzteres wird jedoch nur von der Schlüsselverteilungszentrale genutzt (siehe Kapitel 4.2.1).

Zustandsautomat (FSM)

Das zeitliche Verhalten der oben beschriebenen Komponenten wird durch einen endlicher Zustandsautomat (engl.*: finite state machine, kurz: FSM*) in dem Modul gesteuert. Die Beschreibung der einzelnen Zustände dieses Automaten erfolgt in Kapitel 5.3.

4.1.2 Entschlüsselung

Zur Entschlüsselung der Daten werden separate Hardwarekomponenten verwendet. Somit können die Ver- und Entschlüsselungsvorgänge unabhängig voneinander und parallel ablaufen. Ein Blockschaltbild der Komponenten zeigt Abbildung 36.

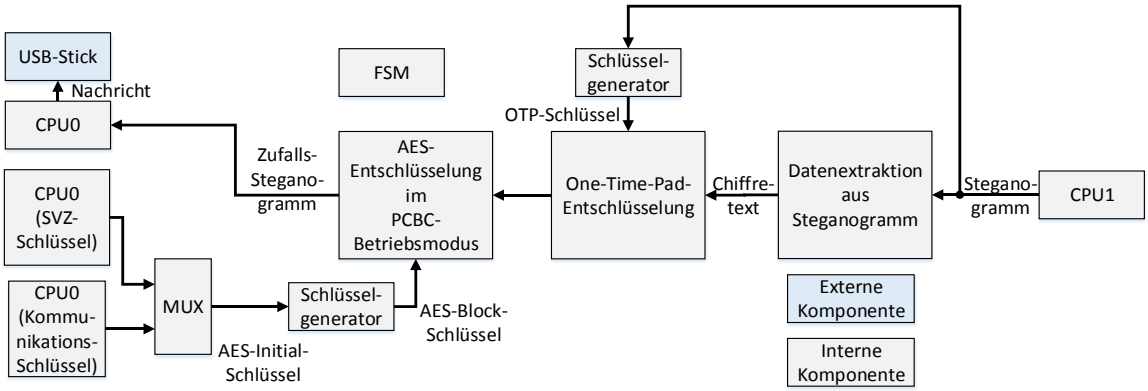


Abbildung 36 - Hardwarekomponente zur Entschlüsselung der Daten

Da von vornherein nicht bekannt ist, in welchem Steganogramm eine Nachricht eingebettet ist, werden aus jedem abgetasteten Steganogramm die relevanten Daten extrahiert und entschlüsselt. Anschließend wird der Inhalt jeder Nachricht geprüft und diese dann gegebenenfalls weitergeleitet. Da ebenfalls unbekannt ist, ob eine Nachricht von einem anderen Teilnehmer oder von der Schlüsselverteilungszentrale (*SVZ*) gesendet wurde (die sich im verwendeten *AES*-Schlüssel unterscheiden, mehr dazu in Kapitel 4.2.1), wird der jeweils erste Block einer Nachricht mit dem Schlüssel des *SVZ* entschlüsselt. Erkennt das System darin eine Nachricht vom *SVZ*, wird die restliche Nachricht ebenfalls mit diesem Schlüssel dechiffriert. Ist dies nicht der Fall, werden die restlichen Blöcke mit dem persönlichen Kommunikationsschlüssel entschlüsselt. Wie eine Nachricht aufgebaut ist und woran das System erkennt, ob eine Nachricht vom *SVZ* oder von einem anderen Teilnehmer gesendet wurde, wird in Kapitel 4.4 näher erläutert.

4.2 Schlüsselverwaltung

In diesem Abschnitt wird beschrieben, wie die Schlüssel für die einzelnen Sicherheitsmechanismen generiert, beziehungsweise unter den Teilnehmern verteilt werden. Dazu gehören der *AES*-Schlüssel, der Schlüssel für die *One-Time-Pad-Verschlüsselung*, sowie der steganographische Schlüssel.

4.2.1 Schlüsselverteilungszentrale (AES-Schlüssel)

Bei einem standardisierten Verschlüsselungsverfahren wie dem *AES* hängt die Sicherheit alleine von der Geheimhaltung des verwendeten Schlüssels ab. Da es sich um ein symmetrisches Verfahren handelt, muss der Schlüssel jedoch zuvor unter den Teilnehmern ausgetauscht werden, bevor eine kryptographische Verbindung aufgebaut werden kann. Dies setzt die Existenz eines sicheren Kanals voraus. Ein solcher Kanal könnte beispielsweise aus einem vertrauenswürdigen Boten bestehen. Abgesehen von der Tatsache, dass auch ein Bote nur begrenzten Schutz bietet, besteht das Problem, dass bei einem Netz mit n Teilnehmern $\frac{n*(n-1)}{2}$ Kommunikationsbeziehungen entstehen können und somit gegebenenfalls auch $\frac{n*(n-1)}{2}$ verschiedene Schlüssel verteilt werden müssten. Dies kann für eine größere Zahl von Teilnehmern einen hohen Arbeitsaufwand für einen Boten darstellen, insbesondere, wenn die Teilnehmer geographisch weit voneinander entfernt sind. Erschwerend kommt hinzu, dass die Schlüssel nach einer gewissen Zeit erneuert werden müssen, um die Sicherheit des Verfahrens nicht zu gefährden. Daher eignet sich die Lösung mit einem Boten in dieser Form nur für ein Netz mit wenigen Teilnehmern, die nur eine geringe Anzahl von Nachrichten verschicken wollen.

Eine Alternative besteht darin, die notwendigen Kommunikationsschlüssel über das Netzt selber zu verteilen. Dafür muss die Schlüsselübertragung wiederum mit einem anderen Schlüssel geschützt werden, da das Netz sonst keinen sicheren Kanal darstellt. Ein Nutzen entsteht dann daraus, wenn die Kommunikationsschlüssel durch eine Schlüsselverteilungszentrale (*SVZ*) verteilt werden. In diesem Fall benötigt jeder Teilnehmer zunächst nur den Schlüssel, mit dem die *SVZ* ihre Nachricht verschlüsselt. Somit reduziert sich die Anzahl der zu verteilenden Schlüssel auf maximal n anstelle von $\frac{n*(n-1)}{2}$. Hinzu kommt, dass der Schlüssel wesentlich seltener erneuert werden muss, als die Kommunikationsschlüssel, da dieser ausschließlich für die Schlüsselübertragung verwendet wird.

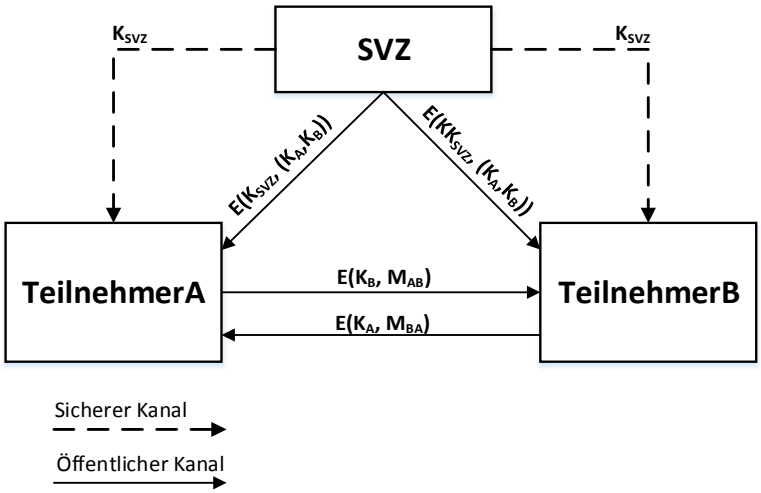


Abbildung 37 - Schlüsselverteilung mit SVZ

Da die Schlüsselverteilung keinen Schwerpunkt dieser Arbeit darstellt, wird hier eine vereinfachte Form der Schlüsselverteilung verwendet: Die *SVZ* generiert in zufällig gewählten Zeitabschnitten ein Steganogramm, in dem sich die verschlüsselten Kommunikationsdaten der Teilnehmer befinden. Dazu gehören die persönlichen Kommunikationsschlüssel, die für das *AES*-Verfahren verwendet werden, die Server-Adressen, mit denen die Steganogramme der Teilnehmer abgetastet werden können, sowie eine ID-Nummer, mit dem jeder Teilnehmer identifiziert werden kann.

Jedes System, das mit der *SVZ* verbunden ist, speichert die Kommunikationsdaten anschließend in einem flüchtigen Speicher. Da flüchtige Speicher ihren Inhalt verlieren, sobald das System ausgeschaltet wird, muss das System beim Wiedereinschalten warten, bis die *SVZ* ein Steganogramm mit den Kommunikationsschlüssel generiert. Alternativ könnten die Schlüssel auch auf dem USB-Datenträger gespeichert werden, damit sie direkt nach dem Start des Systems verfügbar sind. Dies birgt jedoch die Gefahr, dass ein Gegner den Datenträger eines Teilnehmers entwendet und somit an die Kommunikationsschlüssel gelangt. Außerdem könnte ein Teilnehmer seinen eigenen Schlüssel durch den Schlüssel eines anderen Teilnehmers austauschen, um so Nachrichten zu empfangen, die nicht für ihn bestimmt sind.

Allgemein muss angemerkt werden, dass diese vereinfachte Realisierung der *SVZ* eine Schwachstelle des Systems darstellen könnte. Gelingt es einem Angreifer, die Sicherheitsmechanismen der *SVZ* zu brechen, so hat er Zugang zu allen Informationen die er benötigt, um den Nachrichtenaustausch zwischen den Teilnehmern mitzulesen.

Daher sollte bei einer tatsächlichen Nutzung des Systems ein deutlich komplexeres Verfahren zur Schlüsselverteilung genutzt werden.

Verschiedene Protokolle für eine sichere Schlüsselverteilung sind in [2], [18] und [4] beschrieben. Ein einfaches Beispiel aus [2] besteht aus den Teilnehmern A und B. Beide Teilnehmer besitzen jeweils einen gemeinsamen Schlüssel mit der *SVZ* ($K_{A,SVZ}$ bzw. $K_{B,SVZ}$). Wenn Teilnehmer A eine Nachricht an Teilnehmer B übermitteln möchte, so schickt Teilnehmer A zunächst eine Anfrage an die *SVZ*. Diese generiert eine Nachricht mit einem Kommunikationsschlüssel K und den mit $K_{B,SVZ}$ chiffrierten Kommunikationsschlüssel K. Diese Nachricht wird mit dem Schlüssel $K_{A,SVZ}$ verschlüsselt und an Teilnehmer A geschickt. Dieser entschlüsselt die Nachricht und sendet den mit $K_{B,SVZ}$ chiffrierten Schlüssel K an Teilnehmer B. Nun besitzen beide Teilnehmer den Kommunikationsschlüssel K und die eigentliche Nachrichtenübertragung kann beginnen.

Damit das System leicht durch ein komplexeres Protokoll erweitert werden kann, ist das Schlüsselverteilungsprotokoll als Softwaremodul realisiert (siehe Kapitel 4.3).

4.2.2 One-Time-Pad-Schlüssel

Wie bereits beschrieben, wird der Schlüssel für die *One-Time-Pad-Verschlüsselung* aus den Bildinformationen gewonnen, die nicht durch die eingebettete Nachricht überschrieben werden. Somit kann der Schlüssel von dem Empfänger wieder aus den Bilddaten extrahiert werden und der Schlüssel muss nicht über einen gesonderten Kanal verschickt werden.

Bei dem Verfahren werden die Bilddaten byteweise verarbeitet. Jedes Byte entspricht dabei einem Farbkanal eines Pixels (siehe Kapitel 4.4). In dem jeweils niederwertigsten Bit eines solchen Bytes wird später die verschlüsselte Nachricht eingebettet. Aus den restlichen 7 Bits des Bytes wird ein Bit ausgewählt und mit dem entsprechenden Bit der Nachricht XOR-Verknüpft. Das Resultat dieser Verknüpfung wird anschließend als LSB in das Byte eingefügt. Hierbei muss jedoch beachtet werden, dass durch den steganographischen Algorithmus nicht nur das LSB sondern der Wert des gesamten Bytes beeinflusst wird (siehe Kapitel 3.4). Das heißt, die Auswahl des Schlüsselbits darf erst dann getroffen werden, nach dem das Byte durch den Algorithmus dekrementiert, beziehungsweise inkrementiert, wurde.

Die Auswahl des Schlüsselbits sollte möglichst zufällig getroffen werden. Dies würde garantieren, dass für jede Nachricht ein anderer Schlüssel verwendet und die Schlüssel stochastisch unabhängig voneinander gewählt werden würden. Dies würde nach

Shannon absolute Sicherheit gewährleisten (siehe Kapitel 2.1.3). Rein zufällig gewählte Schlüsselbits könnten von dem Empfänger ohne zusätzliche Informationen jedoch nicht rekonstruiert werden, um die Nachricht wieder zu entschlüssen. Daher wird für die Auswahl der Schlüsselbits eine Zufallsgröße verwendet, die sich bereits in den Bilddaten befindet, nämlich das Hintergrundrauschen, das durch die Digitalisierung der Bilder entsteht. Dafür werden die Bits b_1, b_2 und b_3 des Bytes betrachtet. Der Wert dieser drei Bits bestimmt, welches Bit des Farbkanals als Schlüsselbit ausgewählt wird. Das Schlüsselbit wird anschließend mit dem Nachrichtenbit XOR-Verknüpft und das Ergebnis als LSB in dem Byte des Farbkanals gespeichert. Das Prinzip zeigt das Blockschaltbild auf Abbildung 38.

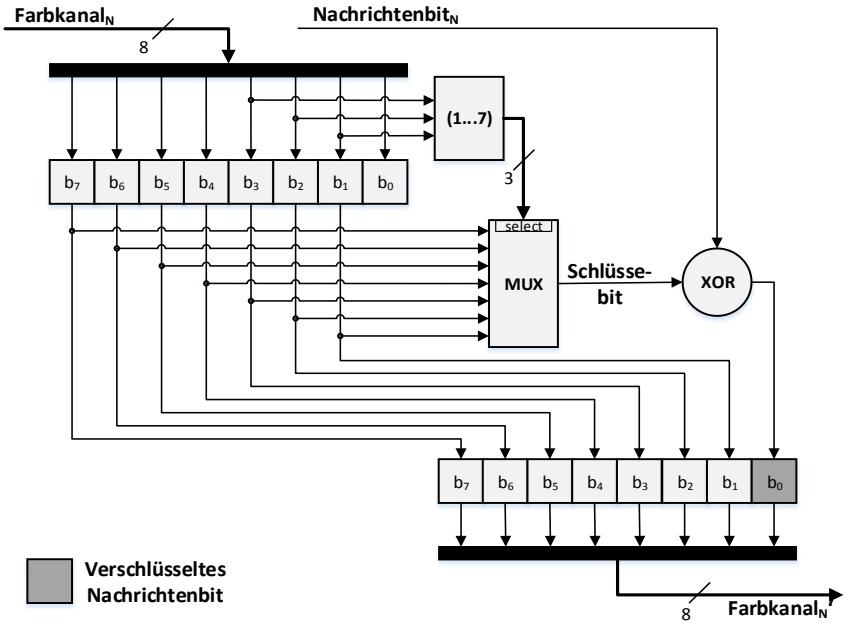


Abbildung 38 - One-Time-Pad-Verschlüsselung eines Nachrichtenbits

Durch das Hintergrundrauschen unterscheidet sich der Wert der drei Bits in jedem Bild, selbst wenn von der Kamera immer das gleiche Motiv aufgenommen wird. Besonders geeignet sind hierbei jedoch dynamische Motive, wie beispielsweise ein Baum, dessen Blätter durch den Wind bewegt werden, oder Motive, bei denen sich die Lichtverhältnisse häufig ändern. Dadurch entstehen nicht deterministische Zufallsgrößen ohne stochastischen Zusammenhang. Diese Zufallsgröße bestimmt, welches Bit aus dem Byte ausgewählt und mit dem aktuellen Nachrichtenbit verknüpft wird. Da das Bit b_0 nicht als Schlüsselbit infrage kommt, wird der Wertebereich zuvor noch auf den Bereich von {1, … ,7} beschränkt.

Somit befinden sich alle Informationen, die der Empfänger zur Entschlüsselung der *One-Time-Pad-Verschlüsselung* benötigt in den übertragenen Daten des Steganogramms. Es wird kein zusätzlicher externer Schlüssel benötigt, das Verfahren beruht somit auf dem in Kapitel 2.1.1 beschriebenem Prinzip „*Security through Obscurity*“. Es sollte daher auch nicht als alleiniger Schutz verwendet werden, sondern lediglich als Erweiterung eines standardisierten Verfahrens, wie in diesem Fall die *AES*-Verschlüsselung. Außerdem sollte das Verfahren möglichst in Hardware realisiert werden, um die Analyse der Funktionsweise zu erschweren, falls ein System vom Gegner entwenden werden sollte.

Um den Nutzen des Verfahrens beurteilen zu können, muss zwischen zwei Fällen unterschieden werden:

1) **Dem Angreifer ist das Verfahren unbekannt.**
   In diesem Fall bietet das Verfahren absoluten Schutz, da alle Kriterien für ein absolut sicheres Verfahren erfüllt sind:
   - Der Schlüssel wird nur einmal pro Nachricht verwendet.
   - Die Schlüssel werden stochastisch unabhängig ausgewählt.
   - Die Schlüssellänge entspricht der Nachrichtenlänge.

2) **Dem Angreifer ist der Verfahren bekannt.**
   Gelingt es einem Angreifer, die Funktionsweise des Verfahrens zu rekonstruieren, so verliert die *One-Time-Pad-Verschlüsselung* jeden Schutz, da der Schlüssel von dem Angreifer aus dem Steganogramm extrahiert werden kann. Jedoch erhöht es den Aufwand, die darunter liegende *AES*-Verschlüsselung zu brechen, da zur Analyse der einzelnen Chiffretexte zunächst die *One-Time-Pad-Verschlüsslung* rückgängig gemacht werden muss.

Ein allgemeiner Nutzen dieses Verfahrens besteht außerdem darin, dass sich die eingebettete Nachricht in jedem Steganogramm unterscheidet, selbst wenn der verwendete *AES*-Schlüssel und der Inhalt der Nachricht immer gleich bleiben. Dies erschwert es mittels Steganalyse zu erkennen, in welchen Steganogrammen sich eine Nachricht befindet.

Um die Sicherheit der Methode zu erhöhen, könnte das Verfahren, mit dem das Schlüsselbit ausgewählt wird, nach gewissen Zeitabständen variiert werden. Beispielsweise könnte der Schlüssel aus vorherigen Steganogrammen gebildet werden, oder es wird ein Schlüssel zur Auswahl verwendet, der dann jedoch wieder über einen sicheren Kanal ausgetauscht werden müsste.

4.2.3 Steganographischer Schlüssel

Der steganographische Schlüssel legt fest, an welcher Stelle eines Containers eine Nachricht eingebettet wird. Dafür könnte ein gesonderter Schlüssel verwendet werden, wobei jedoch wieder die Problematik der Schlüsselverteilung entsteht. Um den Aufwand der Schlüsselverteilung gering zu halten, wird stattdessen der steganographischer Schlüssel aus dem *AES*-Schlüssel generiert.

In dem hier beschriebenen System wird ein Schlüssel nur für die erste steganographische Stufe verwendet (siehe Abbildung 33). Dort wird der Klartext in einen Datenstrom aus Pseudozufallszahlen eingebettet. Dieser Datenstrom wird in 16 Byte große Blöcke aufgeteilt. In jedem dieser Blöcke wird ein Byte des Klartextes eingebettet. Welches Byte der 16 Byte großen Blöcke für das Einbetten des Nachrichtenbytes verwendet wird, bestimmt der steganographische Schlüssel (in diesem Fall also der *AES*-Schlüssel). Dazu wird der Schlüssel in 4 Bit große Nibble aufgeteilt. Der Wert eines Nibbles (0 … 15) bestimmt, in welches Byte des aktuellen Blockes sich das Nachrichtenbyte befindet.

Um möglichst viele verschiedene Nibble aus dem 128-Bit großen Schlüssel generieren zu können, werden die einzelnen Bits des Schlüssels mehrfach verwendet. Das Prinzip, mit dem die Nibble erzeugt werden, zeigt Abbildung 39.

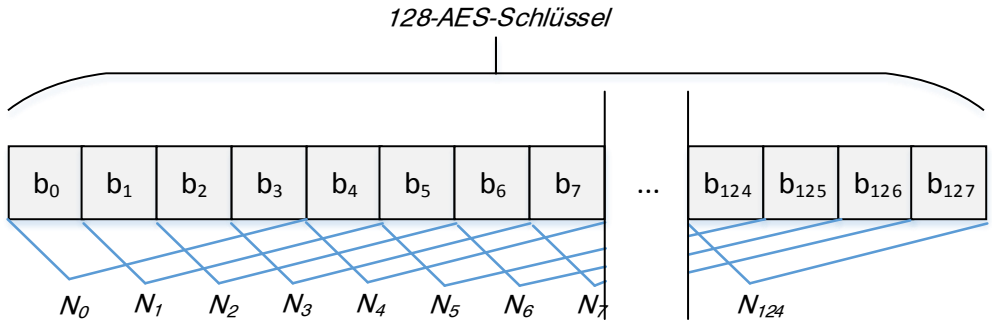


Abbildung 39 - Zerlegung des AES-Keys in 4-Bit große Nibble

Das erste Nibble N_0 ergibt sich somit aus den Bits $\{b_0, \dots, b_3\}$, Nibble N_1 aus den Bits $\{b_1, \dots, b_4\}$, Nibble N_2 aus den Bits $\{b_2, \dots, b_5\}$ und das Nibble N_{124} letztendlich aus den Bits $\{b_{124}, \dots, b_{127}\}$. So können aus den 128 Bits des Schlüssels 125 verschiedene Nibble erzeugt werden ($\{N_0, \dots, N_{124}\}$). Da jedoch 7198 verschiedene 16-Byte-Blöcke existieren (siehe Kapitel 4.4), muss jeder Nibble mehrfach verwendet werden. Dies geschieht in diesem Fall periodisch, könnte aber beispielsweise auch einem bestimmten Muster folgen.

Der Nachteil bei diesem Verfahren besteht darin, dass die maximale Nachrichtenkapazität auf 1/16 reduziert wird. Daher ist das Verfahren als Softwaremodul in dem System implementiert und kann so leicht deaktiviert werden, falls die maximale Nachrichtenlänge, die pro Bild übertragen werden kann, erhöht werden soll.

4.3 Software

Die Daten Ein- und Ausgabe des Systems, sowie die Benutzerinteraktion wird mittels Softwarekomponenten realisiert. Diese werden auf zwei separat laufenden CPU-Kernen implementiert. Die Verbindung der Soft- und Hardwarekomponente zeigt das Blockschaltbild in Abbildung 40.

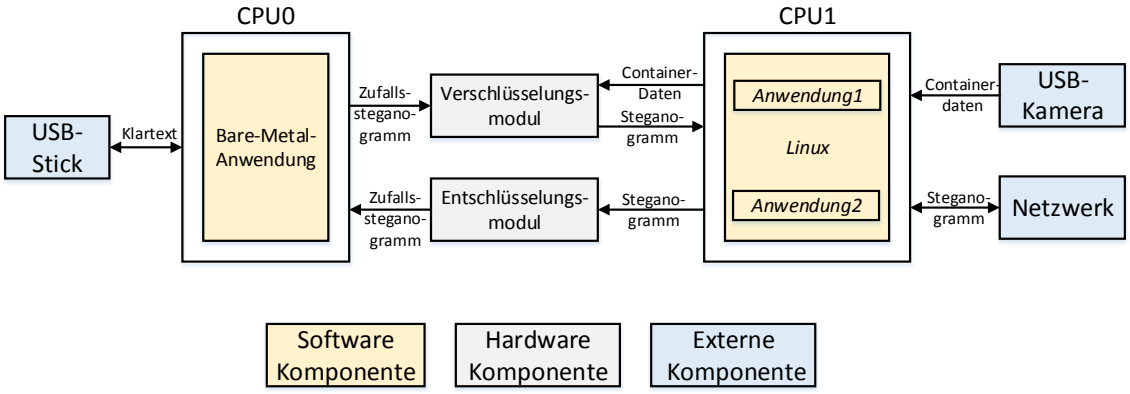


Abbildung 40 - Blockschaltbild mit Soft- und Hardwarekomponenten

CPU0

CPU0 wird im *Bare-Metal-Modus* betrieben, das heißt, die Software läuft ohne Betriebssystem auf dem Prozessor. So kann die Gefahr von eingeschleuster Schadenssoftware reduziert werden. Die Software, die auf *CPU0* implementiert ist, muss besonders vor Manipulationen geschützt werden, da sie sicherheitsrelevante Funktionen des Systems realisiert. Dazu gehört die Verwaltung der *AES*-Schlüssel, das Einlesen, beziehungsweise Abspeichern der Klartexte auf dem USB-Datenträger und das Einbetten des Klartextes in einen Zufallsdatenstrom. Außerdem wird die Interaktion mit dem Benutzer des Systems auf der CPU realisiert. Über eine einfache Menüstruktur kann der Benutzer auf sämtliche Funktionen des Systems zugreifen. Den Programmablauf zeigt das Flussdiagramm in Abbildung 41 (Um die Übersichtlichkeit zu erhalten, werden nur die wichtigsten Funktionen dargestellt).

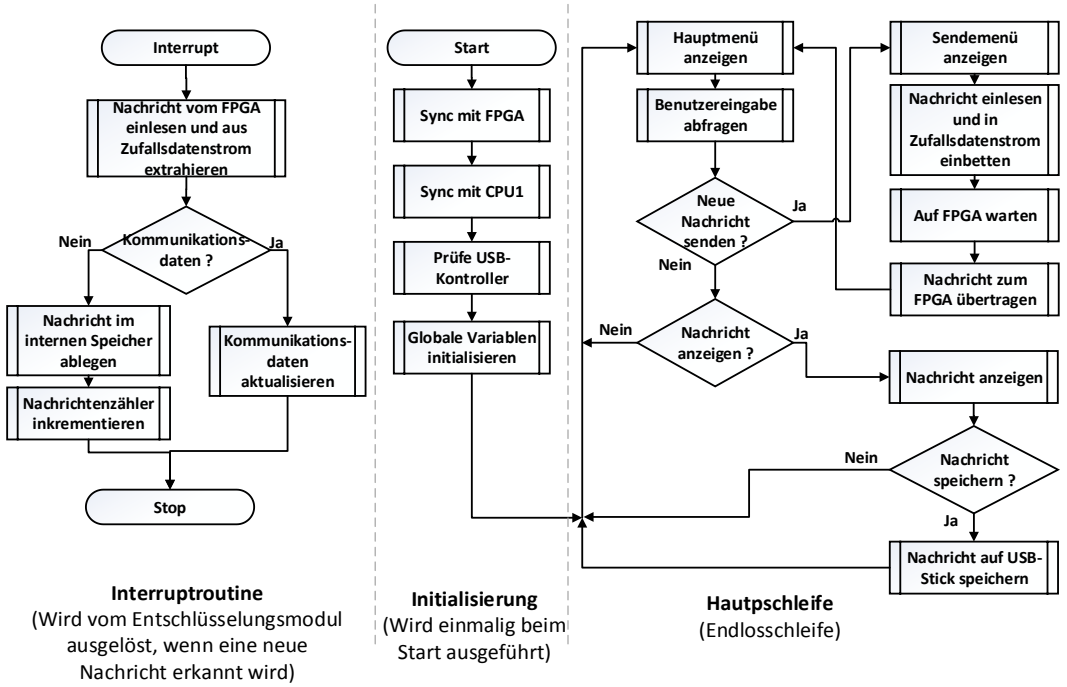


Abbildung 41 - Ablauf der CPU0-Anwendung

Nach dem Start der Anwendung erfolgt zunächst eine Synchronisation mit dem FPGA. Anschließend wartet die Anwendung darauf, dass *CPU1* die Linux-Anwendungen startet. Bedingt durch das Betriebssystem dauert der Bootvorgang für *CPU1* deutlich länger als für *CPU0*. Daraufhin wird die Existenz des externen USB-Kontrollers geprüft und die globalen Variablen initialisiert. Ist der Initialisierungsvorgang abgeschlossen geht die Anwendung in eine Endlosschleife über. In dieser Schleife wird das Hauptmenü angezeigt und die Benutzereingabe abgefragt.

Signalisiert die Eingabe des Benutzers, dass eine neue Nachricht verschickt werden soll, wird ein weiteres Menü angezeigt. Dort kann der Benutzer eine Quelle für die Nachricht auswählen. In der aktuellen Version der Software hat der Benutzer die Wahl zwischen dem USB-Datenträger und einer intern generierten Testnachricht. Diese Auswahl könnte jedoch leicht durch zusätzliche Quellen, wie beispielsweise ein UART-Terminal oder eine direkte Tastatureingabe, erweitert werden. Hat der Benutzer den USB-Datenträger als Quelle ausgewählt, werden ihm sämtliche Dateien auf dem Datenträger angezeigt. Hier kann er eine beliebige Datei als Nachricht auswählen. Wurde eine Datei ausgewählt, wird diese von *CPU0* eingelesen und in einen Datenstrom aus Zufallsdaten eingebettet, falls diese Funktion aktiviert worden ist. Die Zufallsdaten werden durch den Pseudozufallsgenerator im Verschlüsselungsmodul erzeugt. Anschließend wartet die Anwendung, bis der FPGA in einen Zustand gerät, in dem Daten von der *CPU0*

übertragen werden können (mehr Details zu dem Zustandsautomaten, die auf dem FPGA realisiert sind, werden in Kapitel 5.3 beschrieben). Sind die Daten auf den FPGA übertragen worden, springt die Anwendung zurück in die Endlosschleife, und der beschriebene Vorgang beginnt von vorne.

Erkennt das Entschlüsselungsmodul den Empfang einer Nachricht, setzt es ein Interrupt-Signal. Dies führt dazu, dass die Anwendung in die Interrupt-Routine springt. Dort wird geprüft, ob es sich bei der Nachricht um neue Kommunikationsdaten von der *SVZ* handelt oder um eine Nachricht von einem anderen Teilnehmer. Ist ersteres der Fall, werden die Kommunikationsdaten aktualisiert. Anderenfalls wird die Nachricht in dem internen Speicher der CPU abgelegt und der Nachrichtenzähler wird inkrementiert. Die Anzahl der im Speicher hinterlegten Nachrichten wird im Hauptmenü angezeigt.

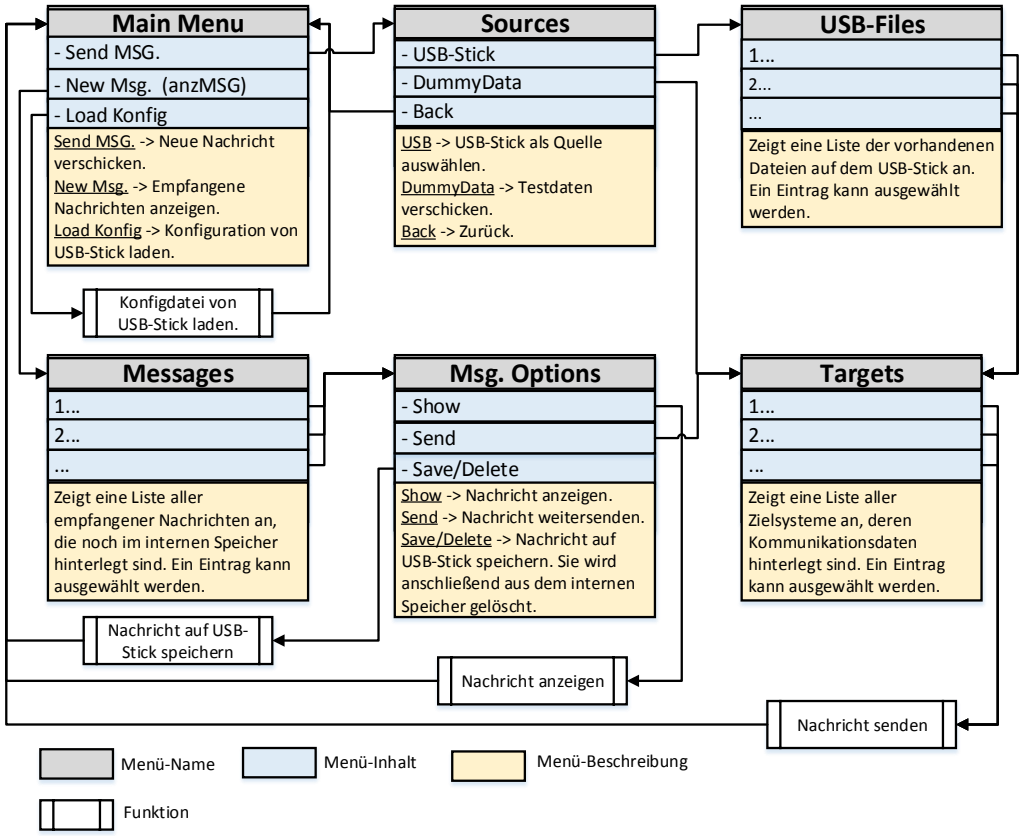


Abbildung 42 - Menüstruktur auf CPU0

Die Menüstruktur, mit der der Benutzer mit dem System interagieren kann, zeigt Abbildung 42. Die einzelnen Menüs werden über ein Display ausgegeben, die Benutzereingabe erfolgt über fünf Taster (siehe Kapitel 5.1).

CPU1

CPU1 dient zum Einlesen der Bilddaten von der USB-Kamera sowie zur Anbindung des Systems an ein öffentliches Netz, wie beispielweise das Internet. Dafür wird ein Betriebssystem auf der CPU verwendet. Die Nutzung eines Betriebssystems hat den Vorteil, dass in der Regel schon fertige Softwarekomponenten für die genannten Aufgaben existieren, was die Implementierung vereinfacht. Der Nachteil besteht darin, dass durch das Betriebssystem und die Anbindung an ein öffentliches Netz die CPU anfällig für einen Hacker-Angriff ist. Somit sollten keine sicherheitskritischen Daten von *CPU1* verwaltet werden.

Als Betriebssystem wird eine Embedded-Linux-Distribution verwendet, die von der Firma *Xilinx* bereitgestellt wird (siehe Kapitel 5.2). Embedded-Linux-Distributionen sind speziell für eingebettete Systeme optimierte Linux-Distributionen mit reduziertem Funktionsumfang, um den Speicherbedarf gering zu halten. Die hier verwendete Distribution stellt bereits eine Server- und Clientfunktionalitäten für das HTTP- und FTP-Protokoll bereit. Lediglich der Treiber für *V4L*-Kompatible[18] USB-Kameras muss nachträglich im Linux-Kernel[19] integriert werden. Das benötigte Dateisystem für Linux wird bei jedem Bootvorgang aus einer Imagedatei gelesen und in den Arbeitsspeicher geladen. Dies ermöglicht einen schnellen Lese- und Schreibzugriff auf die Dateien. Es bedeutet auch, dass Änderungen im Dateisystem nach einem Neustart des Systems verloren gehen und nicht dauerhaft gespeichert werden. Somit können Manipulationen des Systems, die beispielweise durch einen Hacker-Angriff verursacht wurden, durch einen einfachen Neustart rückgängig gemacht werden.

[18] Video4Linux (*V4L*) ist eine weit verbreitete Programmierschnittstelle (*API*) zum Ansprechen von Video-Aufnahme-Geräten unter Linux

[19] Ein Kernel bildet die unterste Softwareschicht in einem Betriebssystem. Er verwaltet unter anderem den Zugriff auf die Hardware und auf das Dateisystem.

Auf dem Linux-Betriebssystem werden zum Senden und Empfangen der Steganogramme zwei zyklische Prozesse gestartet (*Anwendung1* und *Anwendung2*). Die Abläufe der Anwendungen zeigen die Flussdiagramme in Abbildung 43.

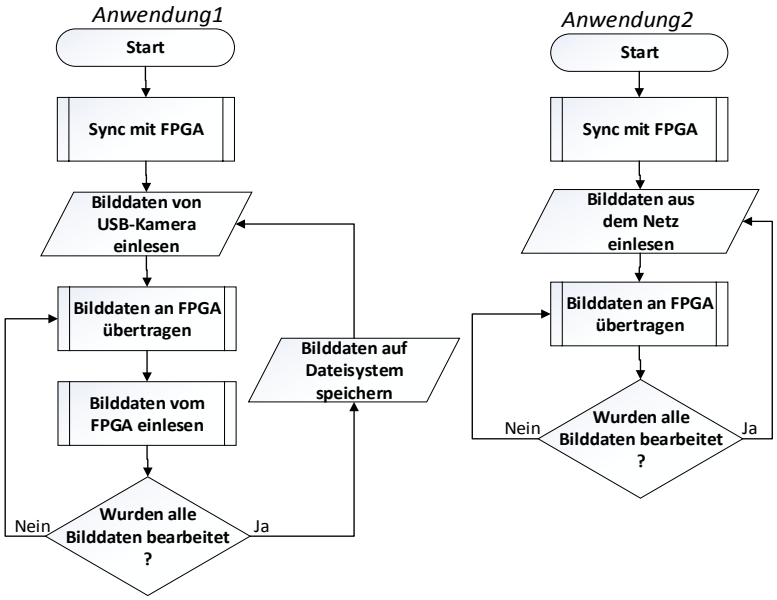


Abbildung 43 - Flussdiagramme der Linux-Anwendungen

Die erste Linux-Anwendung (*Anwendung1*) liest die Bilddaten von der USB-Kamera ein und übermittelt diese an das Verschlüsselungsmodul (siehe Abbildung 40), wo die verschlüsselten Daten eingebettet werden. Anschließend werden die Daten wieder zur *CPU1* übertragen und von der Anwendung auf dem Dateisystem als Steganogramm gespeichert. Von dort werden die so erzeugten Steganogramme durch einen Serverprozess dem öffentlichen Netz zugänglich gemacht, sodass sie von den anderen Teilnehmern abgetastet werden können.

Die zweite Linux-Anwendung (*Anwendung2*) liest die Steganogramme der anderen Teilnehmer über das Netz ein und übermittelt die Bilddaten an das Entschlüsselungsmodul. Dort werden die relevanten Daten extrahiert und entschlüsselt. Erkennt das Modul in den entschlüsselten Daten eine Nachricht, werden diese zu *CPU0* übertragen.

Um sicherzustellen, dass ein erzeugtes Steganogramm von allen Teilnehmern im Netz rechtzeitig empfangen werden kann, sollte die Zykluszeit von *Anwendung1*, mit der die Steganogramme erzeugt werden, deutlich größer sein als die Zykluszeit von

Anwendung2, mit der die Steganogramme abgetastet werden. Wie groß dieser Unterschied sein muss, hängt unter anderem von der Anzahl der Teilnehmer ab.

4.4 Nachrichtenaufbau

Die Steganogramme werden im PNG-Format (*Portable Network Graphics*) gespeichert. Dieses Format hat den Vorteil, dass es eine verlustfreie Kompression verwendet, sodass keine Informationen durch den Kompressionsvorgang verloren gehen. Bei Bildformaten mit einer verlustbehafteten Kompression, wie beispielsweise das JPEG-Format (*Joint Photographic Experts Group*), müsste die Nachricht nach der Kompression in die Bilddaten eingefügt werden, was eine Implementierung deutlich erschweren würde. Vor der Kompression eingebettete Nachrichten würden mit hoher Wahrscheinlichkeit durch das Kompressionsverfahren entfernt oder zumindest verfälscht werden. Bildformate ohne Komprimierung eignen sich aufgrund ihrer Dateigröße nicht für den Transfer über das Internet.

Das System verarbeitet Bilder in einer Auflösung von $640 \cdot 480$ Pixel. Jedes Pixel besteht aus drei Farbkanälen (Rot, Grün und Blau), die als 8-Bit-Werte gespeichert werden. Somit ergibt sich eine Farbtiefe von 24-Bit pro Pixel. In jedes LSB eines Farbkanals kann ein Bit einer Nachricht eingebettet werden, woraus eine maximale Nachrichtenlänge von $640 \cdot 480 \cdot 3 = 921\,600$ Bit pro Bild resultiert. Dies entspricht 115200 Byte, beziehungsweise 7200 128-Bit-Blöcken. Wie diese aufgeteilt sind, zeigt Abbildung 44.

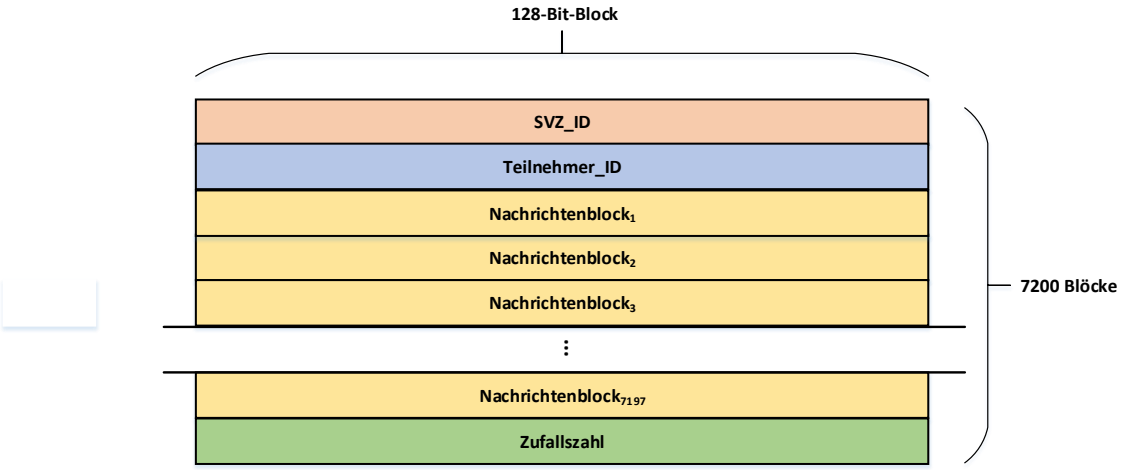


Abbildung 44 - Aufbau einer Nachricht in einem Steganogramm

In dem ersten 128-Bit-Block der Nachricht (*SVZ_ID*) trägt die Schlüsselverteilungszentrale eine von 16 vordefinierten ID-Nummern ein. Somit wird gekennzeichnet, dass sich in dem Steganogramm eine Nachricht von der *SVZ* befindet. Die Auswahl der ID-Nummer erfolgt zufällig. Der erste Block wird von dem System immer mit dem *AES*-Schlüssel der *SVZ* entschlüsselt. Erkennt das System eine von den 16 vordefinierten ID-Nummern, werden die restlichen Blöcke ebenfalls mit dem *SVZ*-Schlüssel entschlüsselt. Dort befinden sich dann die Server-Adressen, ID-Nummern und Kommunikationsschlüssel sämtlicher Teilnehmer des Netzes. Erkennt das System keine ID-Nummer in dem Block, wird zum Entschlüsseln der restlichen Blöcke der persönliche Kommunikationsschlüssel verwendet.

Anschließend wird der zweite Block auf eine ID-Nummer geprüft. Befindet sich dort eine der 16 möglichen ID-Nummern, geht es davon aus, dass sich in den restlichen Blöcken eine Nachricht eines Teilnehmers befindet. In diesem Fall werden die Daten vom Entschlüsselungsmodul zur *CPU0* übertragen, ansonsten verworfen.

Die ersten 16 Byte der übertragenen Nachricht werden für zusätzliche Informationen verwendet. Dazu gehören die ID-Nummer des Senders (2 Byte), der Dateiname (12 Byte) und die Dateilänge (2 Byte). In den nachfolgenden 7183 Bytes befinden sich die Nutzdaten, welche die eigentliche Nachricht beinhalten. Daraus ergibt sich eine maximale Nachrichtenlänge von ~7,00 Kilobyte pro Steganogramm. Soll eine längere Nachricht verschickt werden, kann diese auf mehrere Steganogramme aufgeteilt werden.

In dem letzten Block wird beim Senden ein 128-Bit großer Zufallswert eingefügt. Dies ermöglicht es dem Entschlüsselungsmodul, zu erkennen, ob eine Nachricht bereits abgetastet wurde oder nicht.

5 Implementierung

In diesem Kapitel wird beschrieben, wie das in Kapitel 4 dargestellte System auf einer Hardwareplattform implementiert wird. Dazu wird im Kapitel 5.1 zunächst die verwendete Hardwareplattform vorgestellt. Anschließend werden in Kapitel 5.2 die Softwaretools beschrieben, die zur Entwicklung des Systems genutzt werden. In den danach folgenden Unterkapiteln wird dann die Implementierung der einzelnen Hardware-Komponenten näher erläutert.

5.1 Hardwareplattform

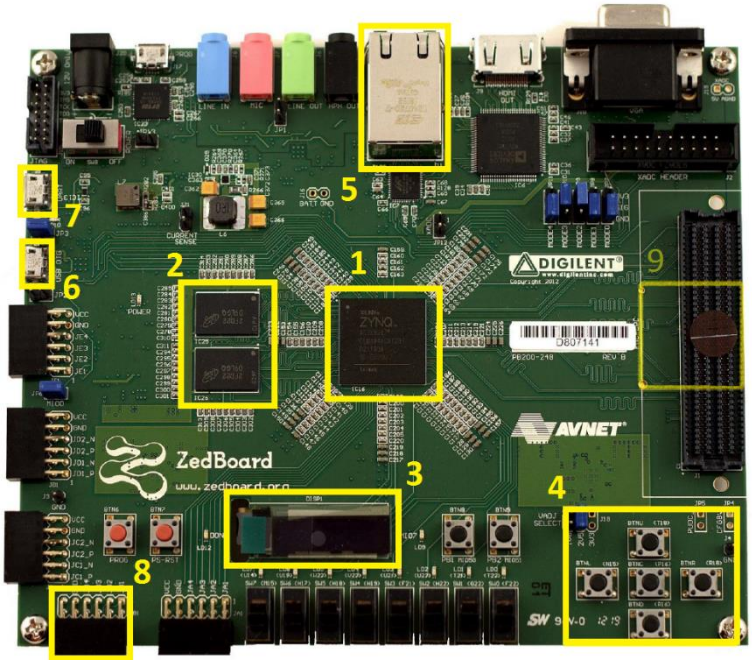


Abbildung 45 - Draufsicht des Zedboards[20]

Als Hardware-Plattform für das System dient das *Zedboard*, welches von der Firma *DIGILENT* [23] entwickelt wurde. Die Draufsicht des *Zedboards* zeigt Abbildung 45.

[20] Die Abbildung wurde aus [36] entnommen.

Die für das System relevanten Komponenten sind auf der Abbildung gelb markiert und werden im Folgenden kurz beschrieben:

1) ***Zynq-Z-7020:***
   Der *Z-7020* ist ein Vertreter der *Zynq-7000-All-Programmable-SoC*-Serie von *Xilinx* und das zentrale Element des *Zedboads*. Er kombiniert ein Prozessor-System (*PS*), bestehend aus einer *Dual-ARM-Cortex-CPU,* mit einer programmierbaren Logik (*PL*), einem *Artix-7 FPGA*. Mehr Details zu den Geräten der *Zynq-7000-Serie* sind in [24] beschrieben.
2) ***512-MB-DDR3-RAM-Speicher:***
   In diesen Speicher wird beim Bootvorgang das Linux-Dateisystem geladen. Außerdem dient er als Arbeitsspeicher für die Linux-Anwendung und als ein Übertragungskanal zwischen den CPUs (*shared memory*).
3) ***128x32-OLED-Display:***
   Das Display wird als Ausgabeelement des Systems verwendet und kann über die programmierbare Logik mit einem SPI-Interface angesteuert werden.
4) ***5-Bit-Taster:***
   Über die fünf integrierten Taster wird die Benutzereingabe realisiert.
5) ***10/100/1G-Ethernet-Anschluss:***
   Mit dem Ethernet-Anschluss wird eine Verbindung zu einem lokalen Netz oder dem Internet hergestellt.
6) ***USB-OTG-Schnittstelle:***
   Dieser Anschluss wird von dem Linux-System zur Ansteuerung der USB-Kamera verwendet.
7) ***UART- Schnittstelle:***
   Diese Schnittstelle dient der Standard-Ein- und -Ausgabe des Linux-Systems.
8) ***PMOD-Anschluss:***
   Über diesen frei belegbaren Anschluss wird das System mit einem externen *USB-OTG*-Kontroller verbunden. Dieser dient zur Anbindung des USB-Datenträgers.
9) ***SD-Karten-Slot:***
   Über den SD-Karten-Slot werden beim Bootvorgang die Systemdateien von einer SD-Karte geladen. Der SD-Karten-Slot befindet sich auf der Rückseite des *Zedboards*.

Wie in Kapitel 4 beschrieben wurde, werden zur Implementierung des Systems mindestens zwei CPUs benötigt. In einem ersten Ansatz wurden dafür die beiden ARM-Prozessoren verwendet, die bereits als *Hardcore-CPUs*[21] auf dem *Zedboard* vorhanden sind. Es zeigte sich jedoch, dass diese, zumindest mit der verwendeten Entwicklungsumgebung, nicht soweit separiert werden konnten, wie es für die

[21] Eine *Hardcore-CPU* ist fest auf einem Chip implementiert. Dies führt in der Regel zu einer geringen Konfigurierbarkeit, jedoch auch zu einer höheren Arbeitsgeschwindigkeit.

Anwendung erforderlich ist. Zwar ist es möglich, für beide CPUs einen eigenen Speicherbereich einzurichten, jedoch können sie nicht mit unterschiedlichen Hardwarekomponente verbunden werden. Somit wäre es beispielsweise möglich, dass von der Linux-CPU auf den USB-Datenträger mit den Nachrichten zugegriffen wird, was die Sicherheit des Systems stark reduzieren würde.

Um diese Gefahr auszuschließen wird eine weitere CPU, ein *MicroBlaze*-Prozessor [25], als *Softcore*[22] auf dem FPGA implementiert. Dieser wird im *Bare-Metal-Modus* betrieben und übernimmt die Aufgaben von *CPU0* (siehe Kapitel 4.3). Die beiden ARM-Prozessoren werden für die Linux-Anwendungen verwendet (*CPU1*).

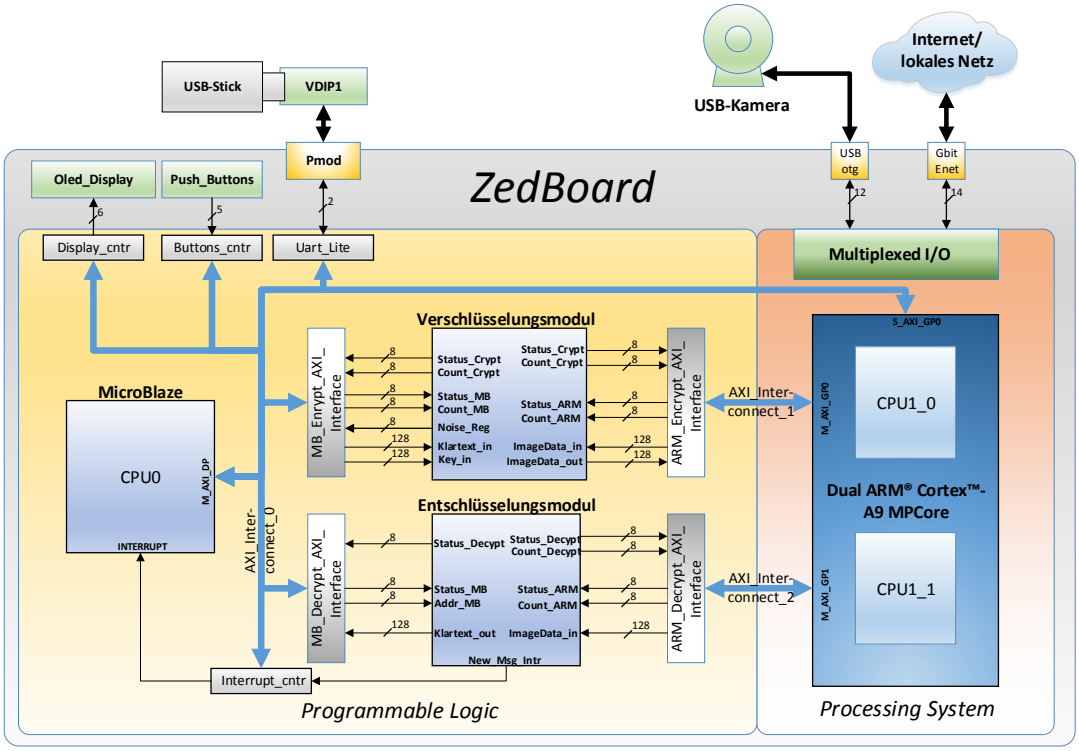


Abbildung 46 - Plattformspezifisches Blockschaltbild

Ein plattformspezifisches Blockschaltbild des Systems zeigt Abbildung 46. Neben dem *Zedboard* werden noch drei externe Komponenten benötigt, eine *V4L*-kompatible USB-Kamera (eine Liste mit kompatiblen USB-Kameras ist in [26] aufgeführt), ein externer USB-Kontroller und ein USB-Datenträger. Als USB-Kontroller wird das *VDIP1*-Modul von *FTDI* verwendet [27]. Dieser wird über ein UART-Interface von dem *MicroBlaze*

[22] Eine *Softcore-CPU* wird durch die Anwenderlogik auf einem FPGA realisiert. Dies bietet einen höheren Freiheitsgrad bei der Konfiguration der CPU.

angesprochen. Anstelle des *USB*-Moduls können auch andere Kommunikationsmodule eingesetzt werden, wie beispielsweise *Bluetooth*- oder *XBee*-Module[23]. Bei der Nutzung drahtloser Übertragungstechnik ist jedoch zu beachten, dass dies die Sicherheit des Systems gefährden könnte.

Die Anbindung der Hardwarekomponenten an die jeweiligen CPUs erfolgt über das AXI-Bussystem (*Advanced extensible Interface Bus*). Das Ver- und Entschlüsselungsmodul wurde im Rahmen dieser Arbeit entwickelt. Der UART- und Interrupt-Kontroller sowie der *MicroBlaze* wurden als fertige *IP-Cores*[24] in das Projekt integriert. Die AXI-Interfaces wurden mit Hilfe von konfigurierbaren Templates erzeugt. Der Displaykontroller basiert auf einem Beispielprojekt von *DIGILENT*, das von Farhad Abdolian für das *Zedboard* angepasst wurde [28]. Es wurde im Rahmen dieser Arbeit so erweitert, dass das OLED-Display über den AXI-Bus angesteuert und eine beliebige Textausgabe erfolgen kann. Zum Einlesen der Taster wurde ebenfalls eine Komponente entwickelt. Diese führt eine Entprellung durch und gibt den Zustand der Taster in einer *One-Hot-Kodierung*[25] auf dem AXI-Bus aus.

[23] *Bluetooth*- und *XBee*-Module nutzen drahtlose Übertragungstechnik gemäß dem IEEE-802.15-Standart [37].
[24] *IP-Cores* (englisch: *intellectual property core*) sind vorgefertigte Funktionsblöcke, die als fertige Komponenten in ein Design integriert werden können.
[25] Bei der *One-Hot-Kodierung* (auch als 1-aus-n-Code bezeichnet) wird jeder Zustand über ein separates Bit repräsentiert.

5.2 Entwicklungsumgebung

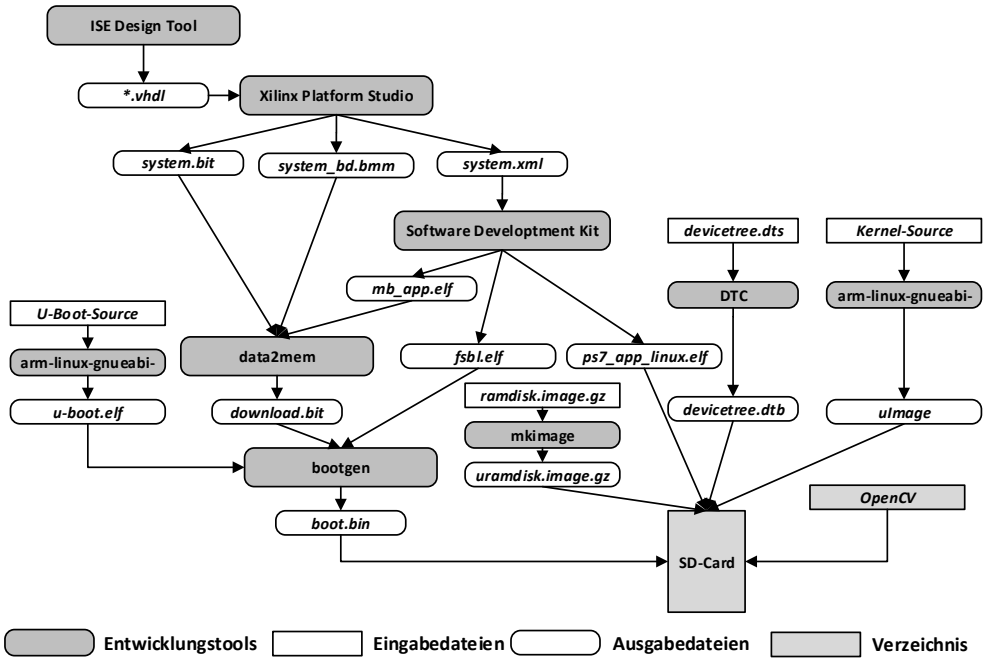


Abbildung 47 – Entwicklungstoolkette

Die Toolkette zur Entwicklung des Systems zeigt Abbildung 47. Mit ihr werden die Dateien generiert, die zur Inbetriebnahme des Systems benötigt werden[26]. Die einzelnen Entwicklungstools werden durch die *ISE Design Suite* von *Xilinx* bereitgestellt [29]. Zu Beginn des Entwicklungsprozesses werden die in Kapitel 4.1 beschriebenen Hardwarekomponente mit dem *ISE Design Tool* in der Hardwarebeschreibungssprache *VHDL* erstellt und ihre Funktion mit Hilfe des integrierten Simulators *ISim* verifiziert. Anschließend werden die Komponenten als sogenannte *pcores*[27] im VHDL-Format in ein *Xilinx-Platform-Studio*-Projekt importiert. Im *Xilinx Platform Studio (XPS)* werden die erstellten Komponenten mit dem *MicroBlaze* und dem *ARM-Prozessor-System* verbunden. Außerdem werden noch fertige Komponenten aus dem sogenannten IP-Katalog[28] hinzugefügt. Dazu gehören ein UART-Kontroller und ein BRAM-Block (256 Kilobyte) für den *MicroBlaze*. Anschließend wird das Projekt synthetisiert. Durch die Synthese werden drei wesentliche Dateien generiert: Die Datei *system.bit*, welche die

[26] Alle Quell- und Projektdateien, die zur Implementierung des Systems entwickelt wurden, befinden sich in Anhang C.
[27] Als *pcores* werden Hardwaremodule bezeichnet, die als eigenständige Komponenten in ein Design integriert werden können.
[28] Der *IP-Katalog* enthält eine Reihe von vorgefertigten IP-Cores, die von *Xilinx* bereitgestellt werden.

Konfiguration des *Zynq*-Chips enthält, eine Speicherabbilddatei des BRAMs (*system_bd.bmm*) und die Datei *system.xml*. Letztere beinhaltet eine Beschreibung des Systems im *XML*-Format. Sie wird anschließend in das *Software Development Kit* (*SDK*) importiert.

Mit Hilfe des *SDK*s wird die Software für das System in der Programmiersprache *C* erstellt und kompiliert. Insgesamt werden drei Softwareanwendungen erstellt: Die Anwendung für den *MicroBlaze* (*mb_app.elf*), die Anwendung für das Linux-System (*ps7_linux_app.elf*), die sowohl den Prozess zum Erstellen als auch zum Einlesen von Steganogrammen enthält und ein *First-Stage-Bootloader* (*fsbl.elf*), der zum einen den FPGA mit der bit-Datei konfiguriert und zum anderen einen weiteren Bootloader (*u-boot.elf*) in den Speicher lädt und ausführt. Dieser Bootloader, bezeichnet als *The Universal Bootloader* (kurz: U-Boot), ist unter anderem für den Startvorgang von Linux zuständig.

Die *MicroBlaze*-Anwendung (*mb_app.elf*) sowie die beiden zuvor generierten Dateien *system.bit* und *system_bd.bmm* werden durch das Tool *data2mem* zu einer Datei mit dem Namen *download.bit* zusammengefügt. Dies ist die *bit*-Datei, mit der beim Bootvorgang der FPGA konfiguriert wird. Aus dieser Datei wird zusammen mit den beiden Bootloader-Anwendungen eine Bootimage-Datei erstellt (*boot.bin*). Um die Sicherheit des Systems zu erhöhen, kann die Bootimage-Datei während des Vorgangs durch das *Bootgen*-Tool verschlüsselt werden.

Neben dem Bootimage werden noch drei weitere Dateien für den Bootvorgang benötig. Dazu gehört ein Linux-Kernel (*uImage*), ein komprimiertes Linux-Dateisystem (*uramdisk.image.gz*), welches beim Bootvorgang in den Arbeitsspeicher geladen wird und einen device-tree-blob (*devicetree.dtb*). Dieser dient als Schnittstelle zwischen dem Linux-System und der vorhandenen Zielhardware. Diese vier Dateien werden auf einer SD-Karte gespeichert, von der das System dann gebootet werden kann. Auf der SD-Karte müssen sich außerdem noch die Linux-Anwendungen und eine *OpenCV*-Bibliothek[29] befinden, mit deren Hilfe die Linux-Anwendung die Bilddaten erzeugt und verarbeitet.

Die Quellcodes der beiden Bootloader und des Linux-Kernels sind frei verfügbar und laufen unter der *GNU-General-Public*-Lizenz. Diese Quellcodes müssen, genauso wie die andere Anwendung, mit einem Cross-Compiler (*arm-linux-gnueabi-gcc*) für die ARM-Architektur kompiliert werden. Der Kernel-Quellcode muss zuvor noch so konfiguriert werden, dass der fertige Kernel anschließend die Treiber für die V4L-USB-Kameras

[29] *OpenCV* (*Computer Vision*) ist eine freie Bibliothek für die Programmiersprachen *C* und *C++*. Sie enthält unter anderem Algorithmen zur Bildverarbeitung.

integriert hat. Der *device-tree-blob* muss mit einem speziellen Compiler erstellt werden (*device tree compiler*, kurz: *DTC*). Die entsprechende Quelldatei (*devicetree.dts*) wird ebenfalls von *Xilinx* mit der *GNU-General-Public*-Lizenz bereitgestellt. In der Datei können auch generelle Einstellung, wie beispielsweise die Festlegung der IP- und MAC-Adresse, vorgenommen werden.

5.3 Implementierung der Hardwarekomponenten

In diesem Abschnitt wird erläutert, wie die in Kapitel 4.1 beschriebenen Hardware-Komponenten auf der Zielplattform implementiert werden. Alle Komponenten wurden in der Hardwarebeschreibungssprache VHDL umgesetzt. Die Hardwarekomponenten lassen sich im Wesentlichen in zwei Module zusammenfassen: dem Verschlüsselungsmodul, mit dem die Nachrichten verschlüsselt und in die Containerdaten eingebettet werden, und dem Entschlüsselungsmodul, mit dem die Nachrichten aus den Steganogrammen extrahiert und anschließend entschlüsselt werden.

5.3.1 Verschlüsselungsmodul

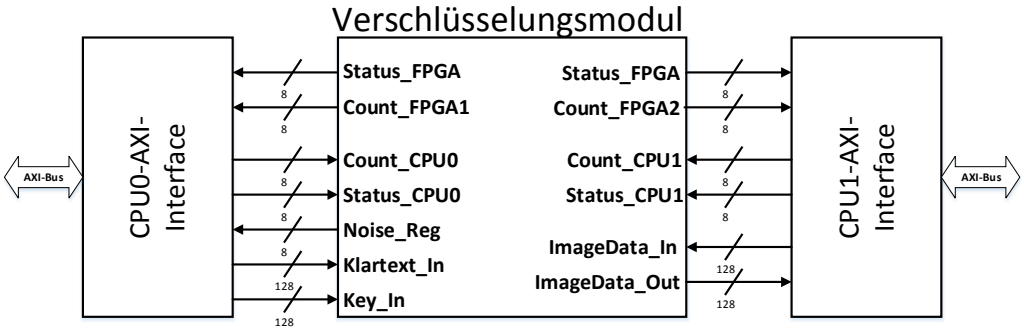


Abbildung 48 - Ein- und Ausgangsports des Verschlüsselungsmoduls

Die Ein- und Ausgänge des Verschlüsselungsmoduls zeigt Abbildung 48. Diese werden jeweils über separate AXI-Interface-Module mit den jeweiligen CPUs verbunden. Die 8-Bit-Status- und Count-Ein- und Ausgänge werden zur Synchronisation verwendet. Wie sich die einzelnen Elemente des Systems synchronisieren wird im nachfolgenden Abschnitt näher beschrieben. Über die 128-Bit-Dateneingänge *Klartext_In* und *Key_In* werden die benötigten Daten von *CPU0* in das Modul geladen. Dazu gehören das Zufallssteganogramm mit der Nachricht und der Kommunikationsschlüssel für das *AES*-Verfahren. Außerdem kann die *CPU0* über den 8-Bit-Ausgang *Noise_Reg* einen aktuellen

Ausgangswert des Zufallsgenerators auslesen. Die 128-Bit-Ports *ImageData_In* und *ImageData_Out* dienen zur Übertragung der Bilddaten zwischen dem Modul und *CPU1*.

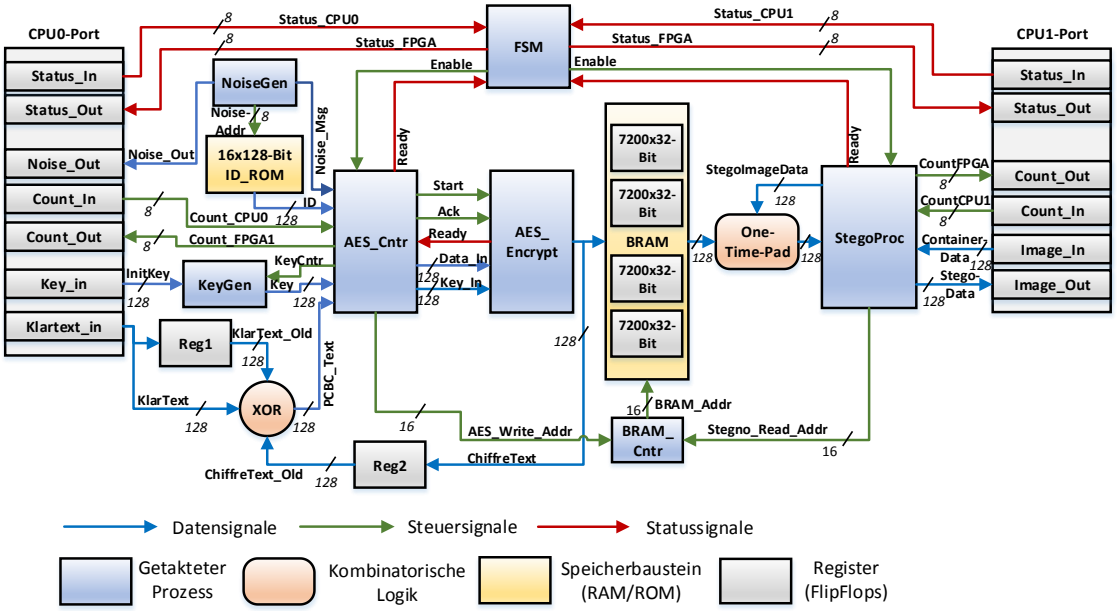


Abbildung 49 - Top-Level des Verschlüsselungsmoduls

Die oberste Ebene (*Top-Level*) des Entschlüsselungsmoduls zeigt Abbildung 49. Für eine bessere Übersicht beschränkt sich die Abbildung auf die Komponenten, die hauptverantwortlich für den Vorgang sind. Gesteuert werden die Komponenten durch einen endlichen Zustandsautomaten (*FSM*). Dieser durchläuft seine Zustände zyklisch und ist dabei fest mit der Anwendung auf *CPU1* synchronisiert. Eine Synchronisation mit *CPU0* erfolgt nur, wenn eine Nachricht übertragen werden soll. Dies prüft der Automat in jedem Zyklus, in dem er das Statusregister von *CPU0* ausliest. Anschließend wird die Komponente *AES_Cntr* aktiviert. Soll eine Nachricht von *CPU1* übertragen werden, liest die Komponente jeweils ein 128 Bit breites Datenwort parallel ein, verschlüsselt dieses und speichert es in den Speicherstellen des BRAMs zwischen. Die beiden Register *Reg1* und *Reg2*, sowie die XOR-Verknüpfung dienen zur Umsetzung des *PCBC*-Betriebsmodus (siehe Kapitel 3.3). Ist die Verschlüsselung des Blocks abgeschlossen, wird der nächste 128-Bit-Block eingelesen.

Zur Synchronisation dieses Vorgangs werden zwei verschiedene Zählerwerte zwischen *CPU0* und dem Verschlüsselungsmodul übertragen. Hat die Anwendung auf *CPU0* einen neuen Block über den AXI-Bus in das entsprechende Register geschrieben, so erhöht die Anwendung ihren Zählerwert um Eins. Dies wird von dem Modul erkannt, der Block wird ausgelesen und verschlüsselt im BRAM gespeichert. Anschließend inkrementiert

die Komponente ihren Zählerwert. Erkennt die Anwendung auf *CPU0* nun, dass beide Zählerwerte gleich sind, schreibt sie den nächsten Block in das Register. Die 16 Bit breiten Zähler dienen außerdem zur Adressierung des BRAMs und um zu erkennen, wann die Datenübertragung beendet ist. Von den 16 Bit werden nur die untersten 8 Bit zwischen *CPU0* und dem Modul übertragen.

Wird die Komponente aktiviert, ohne dass *CPU0* Daten übertragen möchte, wird der Ausgangswert von dem Pseudozufallsgenerator ausgelesen, verschlüsselt und in dem BRAM gespeichert. Den Ablauf des beschriebenen Vorgangs zeigt Abbildung 50.

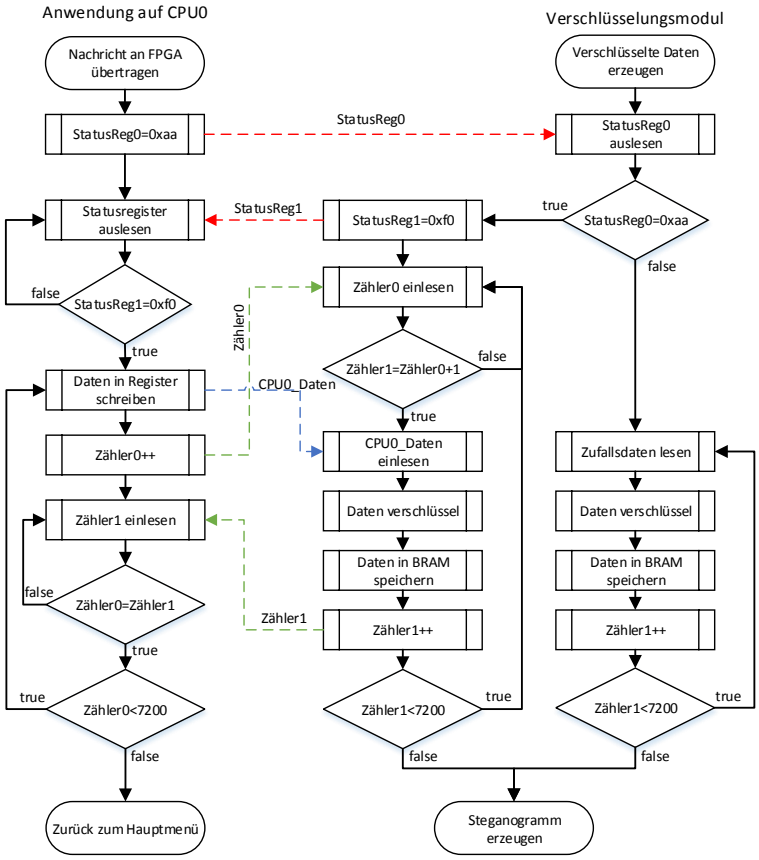


Abbildung 50 – Erzeugung der verschlüsselten Daten

Ist der Vorgang abgeschlossen, befindet sich nun in dem BRAM entweder eine verschlüsselte Nachricht oder verschlüsselte Zufallsdaten. Daraufhin aktiviert der Zustandsautomat die Komponente *StegoProc*. Da in jedem Zyklus ein Steganogramm erzeugt werden soll, erfolgt nun eine Synchronisation mit der Anwendung auf *CPU1*. Ist diese hergestellt, überträgt *CPU1* die Bilddaten, ebenfalls in 128-Bit-Blöcken, an das

Verschlüsselungsmodul. Dort wird aus jedem Block der One-Time-Pad-Schlüssel extrahiert und es werden jeweils 16 Bit der verschlüsselten Daten in dem Bilddatenblock eingebettet. Anschließend werden die Daten wieder zur *CPU1* übertragen. Dies wird solange wiederholt, bis alle Bilddaten verarbeitet wurden. Daraufhin werden die Bilddaten als Steganogramm gespeichert. Den Ablauf zur Erzeugung eines Steganogramms zeigt Abbildung 51.

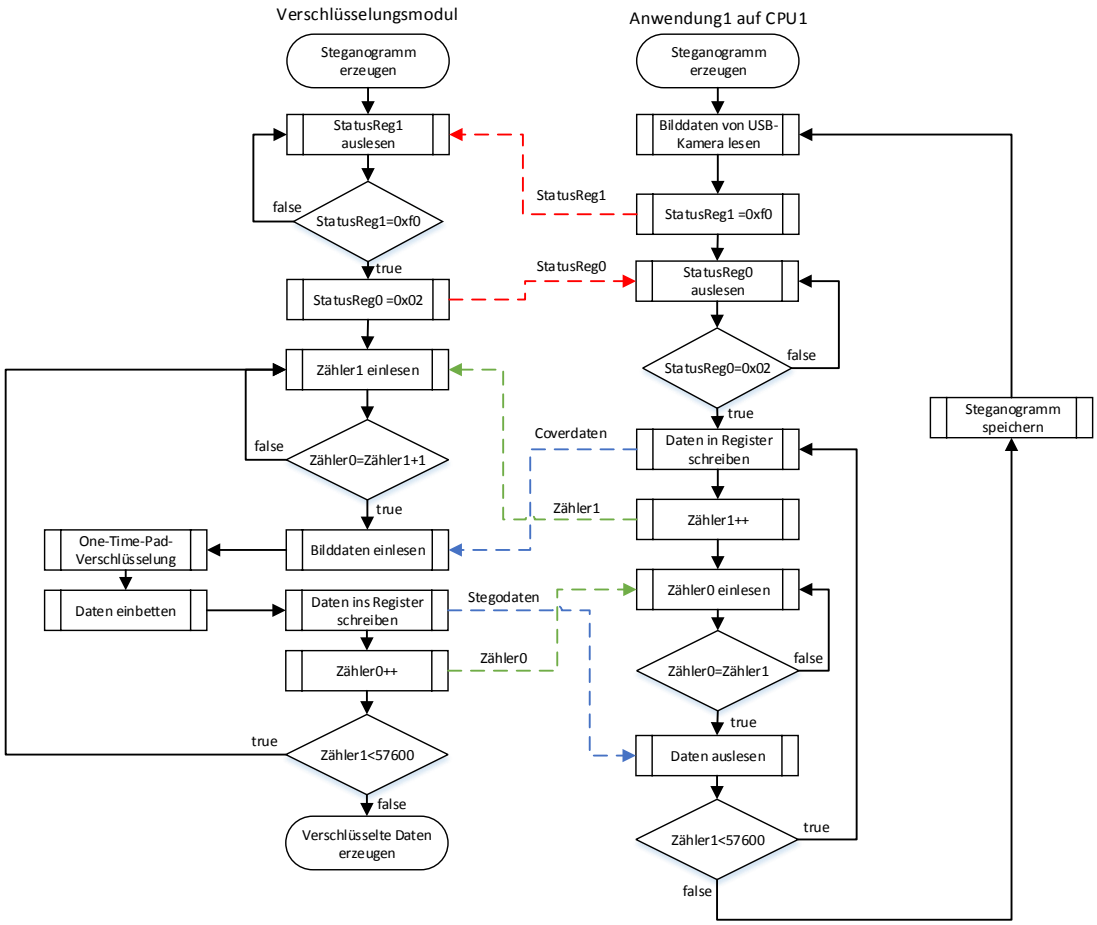


Abbildung 51 - Erzeugung eines Steganogramms

Nach der Erzeugung eines Steganogramms ist der Zyklus beendet und beginnt erneut mit der Generierung verschlüsselter Daten.

Die einzelnen Komponenten, die bei dem Vorgang beteiligt sind, werden im folgenden Abschnitt genauer beschrieben.

Zustandsautomat (FSM)

Zur Steuerung des zeitlichen Ablaufs wird ein Moore-Zustandsautomat verwendet. Dieser zeichnet sich dadurch aus, dass kein kombinatorischer Pfad zwischen den Ein- und Ausgängen des Automaten existiert und die Ausgänge somit Taktsynchron sind [30]. Das Zustandsdiagramm des Automaten zeigt Abbildung 52.

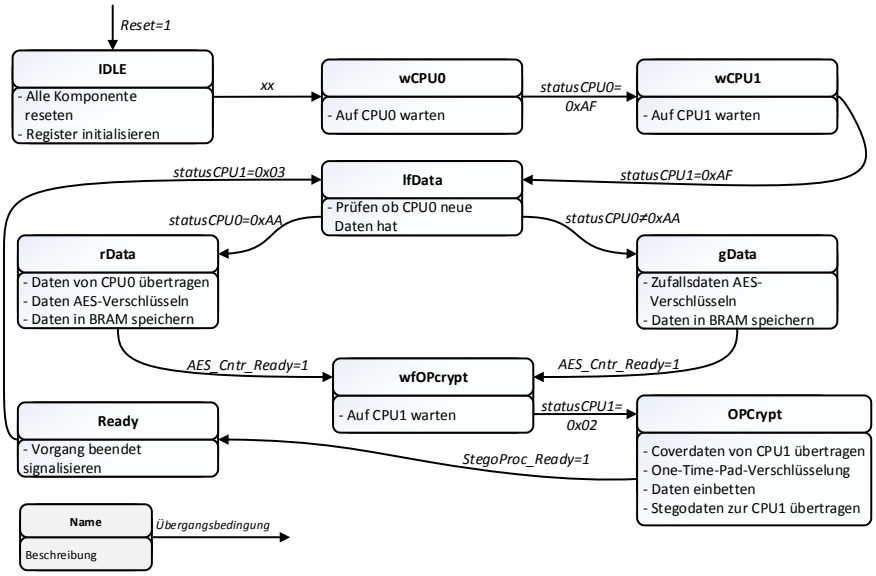


Abbildung 52 - Zustandsdiagramm des Verschlüsselungsmoduls

Die ersten drei Zustände dienen zur Initialisierung der Register und zur Synchronisation zwischen dem Modul und den beiden CPUs. Die darauffolgenden Zustände werden zyklisch durchlaufen. Im Zustand *lfData* wird geprüft, ob *CPU0* neue Daten zur Verfügung hat. Ist dies der Fall (*statusCPU0=0xAA*) werden die Daten im Zustand *rData* übertragen, *AES*-verschlüsselt und im BRAM zwischengespeichert. Ist dies nicht der Fall (*statusCPU0≠0xAA*) werden im Zustand *gData* *AES*-verschlüsselte Zufallsdaten im BRAM gespeichert. In beiden Fällen wird anschließend im Zustand *wfOPcrypt* darauf gewartet, dass die Anwendung1 auf *CPU1* neue Bilddaten generiert hat (*statusCPU1=0x02*). Anschließend erfolgt ein Übergang zu dem Zustand *OPcrypt*, wo die Daten aus dem BRAM mit den Schlüsselbits der Bilddaten verknüpft und anschließend in die Bilddaten eingebettet werden. Ist der Vorgang abgeschlossen (*StegoProc_Ready=1*) wechselt der Automat in den Zustand *Ready,* in dem der *CPU1* die Beendigung des Vorgangs signalisiert wird. Erfolgt daraufhin eine Bestätigung von *CPU1* (*statusCPU1=0x03*), wechselt der Automat zurück in den Zustand *lfData*, und prüft erneut, ob *CPU0* neue Daten zur Verfügung hat.

AES-Verschlüsselung (AES_Encrypt)

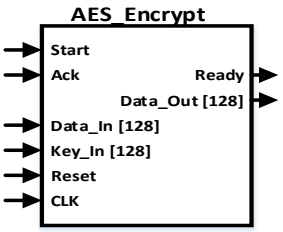


Abbildung 53 - AES_Encrypt-Komponente

Mit der Komponente *AES_Encrypt* wird der *AES*-Verschlüsselungsalgorithmus auf dem FPGA implementiert. Die Ein- und Ausgangsports der Komponente zeigt Abbildung 53. Auch in dieser Komponente wird ein Moore-Automat zur Steuerung der zeitlichen Abläufe eingesetzt. Das Zustandsdiagramm zeigt Abbildung 54.

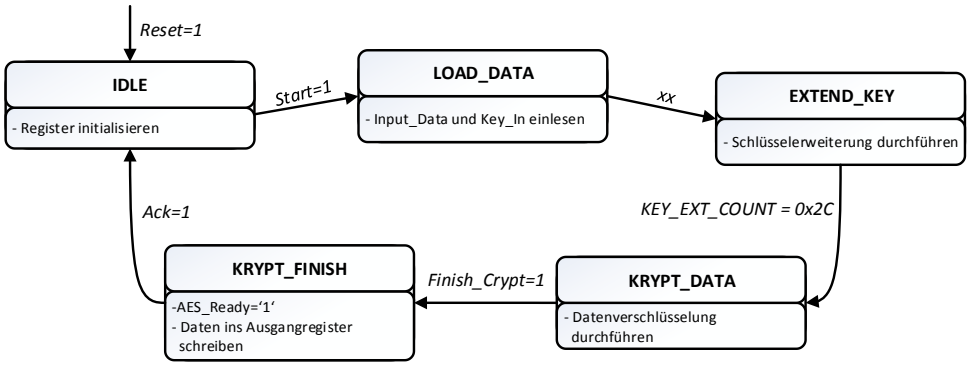


Abbildung 54 - Zustandsdiagramm AES_Encrypt

Sobald der Eingang *Start* auf ‚1' gesetzt wird, werden die Werte an den Eingängen *Data_In* und *Key_In* abgetastet und gespeichert (*Load_Data*). Anschließend wird der Verschlüsselungsvorgang gestartet. Dazu werden im Zustand *Extend_Key* die Rundenschlüssel generiert und im Zustand *Krypt_Data* die Daten verschlüsselt. Im Zustand *Krypt_Finish* wird das *Ready*-Signal auf ‚1' gesetzt und das Ergebnis der Verschlüsselung in dem Ausgangsregister *Data_Out* gespeichert.

Um die Funktionsweise der Komponente zu überprüfen, wurde eine Simulation mit *ISim* durchgeführt. Das Ergebnis dieser funktionellen Simulation zeigt Abbildung 55.

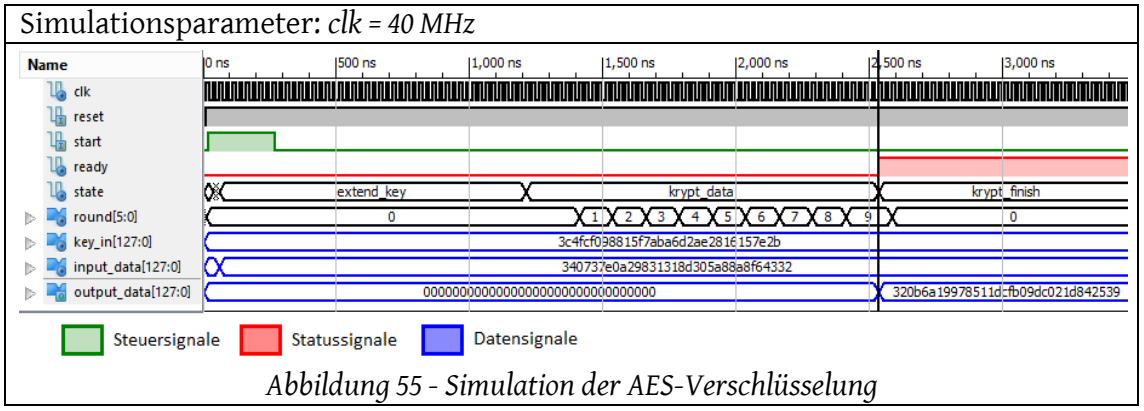


Abbildung 55 - Simulation der AES-Verschlüsselung

In der Simulation ist ein kompletter Verschlüsselungsvorgang dargestellt. Mit dem Setzen des Startsignals (*start=1*) werden die Eingangsdaten geladen. Anschließend werden die Rundenschlüssel erzeugt (*state=extend_key*) und die Daten verschlüsselt (*state=krypt_data*). Der Cursor markiert den Zeitpunkt, an dem der Verschlüsselungsvorgang beendet ist *(ready='1')* und die verschlüsselten Daten in das Ausgangsregister (*ouput_data*) gespeichert werden. Mit einer Taktfrequenz von $f_{clk} = 40\ MHz$ werden dafür $\sim 2{,}5\ \mu s$ benötigt (dies entspricht ungefähr 100 Taktzyklen). Für die Simulation wurden die Beispielswerte aus Kapitel 3.1 übernommen. Der Ausgangswert im Register *output_data* verifiziert die korrekte Implementierung des *AES*-Algorithmus.

Theoretisch ließe sich der Algorithmus als reine kombinatorische Logik implementieren, sodass sich die Gesamtdauer des Vorgangs auf wenige Takte reduzieren ließe. Dies würde jedoch den Ressourcenverbrauch auf dem FPGA erhöhen und auch die maximal mögliche Taktfrequenz reduzieren. Da die Anwendung nicht auf einen möglichst hohen Datendurchsatz ausgelegt ist, wird eine Variante implementiert, die zwar eine etwas längere Laufzeit besitzt, jedoch auch weniger Ressourcen beansprucht. In dieser Variante werden die einzelnen Schritte des Algorithmus seriell durchlaufen und die jeweiligen Zwischenergebnisse in einem eigenen BRAM zwischengespeichert.

AES-Kontroller (AES_Cntr)

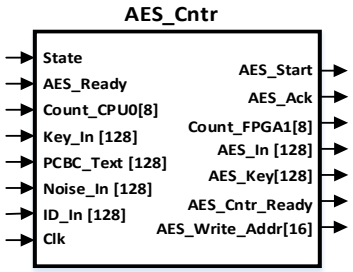


Abbildung 56 - AES_Cntr-Komponente

Zur Steuerung der *AES*-Verschlüsselungskomponente, wird ein *AES*-Kontroller implementiert. Die Ein- und Ausgänge des Kontrollers zeigt Abbildung 56. Er wird aktiviert, sobald der Zustandsautomat den Zustand *rDtata* oder *gData* erreicht (siehe Abbildung 52). Der aktuelle Zustand wird über den Eingang *State* abgefragt. Je nach dem welcher der beiden Zustände gerade aktiv ist, werden entweder Daten von dem Zufallsgenerator (*Noise_In*) oder Daten von *CPU0* (*PCBC_Text*) eingelesen. Die Ansteuerung der AES-Komponente erfolgt über die Steuerausgänge *AES_Start* und *AES_Ack* sowie über den Statuseingang *AES_Ready*.

Innerhalb des Kontrollers befindet sich ein 16-Bit-Zähler. Dieser zählt die einzelnen Blöcke, die bereits verschlüsselt wurden. Inkrementiert wird der Zähler, sobald das *AES_Ready*-Signal auf ‚1' gesetzt wird. Von den 16 Bits des Zählers werden die untersten 8 Bits auf den Ausgang *Count_FPGA1* gelegt. Dieser dient zusammen mit dem Eingang *Count_CPU0* zur Synchronisation der Datenübertragung zwischen dem Verschlüsselungsmodul und *CPU0* (siehe Abbildung 50). Mit dem Zähler wird außerdem die Speicherstelle des BRAMs adressiert (*AES_Write_Addr*), in dem der aktuelle Ausgangswert der *AES*-Komponente gespeichert wird. Erreicht der Zähler seinen Endstand von 7199, wird das *AES_Cntr_Ready*-Signal gesetzt, um dem Zustandsautomaten die Beendigung des Vorgangs zu signalisieren.

Wird die Komponente im Zustand *rData* aktiviert (es soll also eine Nachricht von *CPU0* verschlüsselt werden), wird im ersten Block eine Zufallszahl und im zweiten Block eine zufällig ausgewählte ID-Nummer verschlüsselt, die über den Eingang *ID_In* eingelesen wird (hier unterscheidet sich der Ablauf vom *SVZ*, welche im ersten Block die ID und im zweiten Block eine Zufallszahl verschlüsselt einträgt).

Simulationsparameter: *clk = 40 MHz*

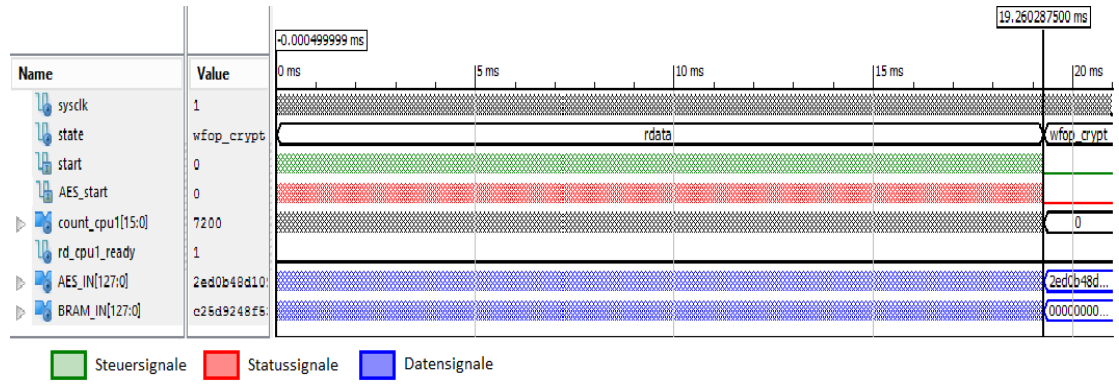


Abbildung 57 - Simulation AES_Cntr (gesamt)

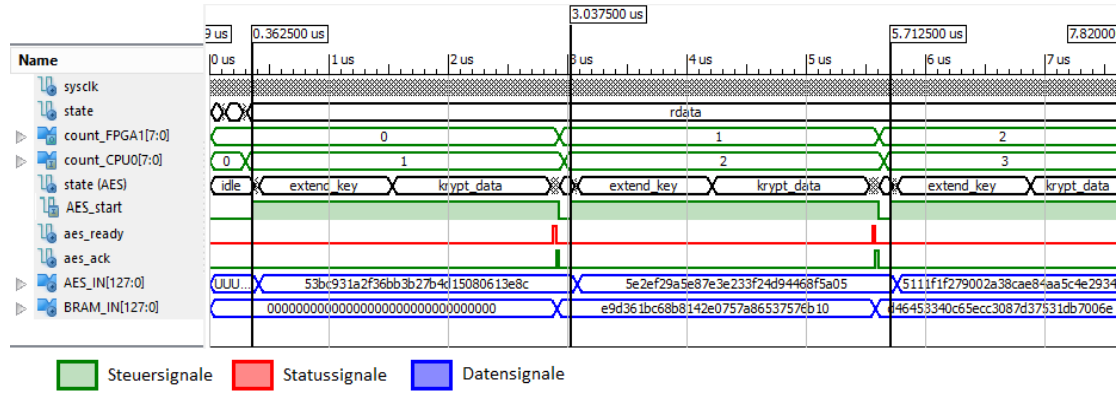


Abbildung 58 - Simulation AES_Cntr (Ausschnitt)

Tabelle 12 - Simulation des AES-Kontrollers

Das Ergebnis einer Verhaltenssimulation der Komponente zeigt Tabelle 12. Im oberen Teil der Tabelle (Abbildung 57) ist ein kompletter Durchlauf dargestellt. Dem Simulationsergebnis ist zu entnehmen, dass bei einer Taktfrequenz von $f_{clk} = 40\ MHz$ $\sim 20\ ms$ für die Verschlüsselung aller 7200 Blöcke benötigt werden. Dabei werden insgesamt $7200 \cdot 128 = 921\,600$ Bit verarbeitet. Daraus ergibt sich ein maximaler Datendurchsatz von $\sim 43\ Mbit/s$ für den kompletten Verschlüsselungsvorgang. Im unteren Teil der Tabelle (Abbildung 58) ist die Synchronisation zwischen den Komponenten *AES_Cntr* und *AES_Encrypt* dargestellt. Dafür wird ein 3-Wege-Handschlag-Verfahren verwendet. Das Signal *AES_Start* wird von dem Kontroller auf ‚1' gesetzt, woraufhin der Verschlüsselungsvorgang startet. Ist dieser beendet, setzt die *AES*-Komponente das Signal *AES_Ready* auf ‚1'. Wird dies von dem Kontroller erkannt, erfolgt eine Bestätigung mit dem *Ack*-Signal (engl.: acknowledgement). Daraufhin geht die *AES*-Komponente wieder in den IDLE-Zustand über und setzt das Signal *AES_Ready*

zurück. Dieses Synchronisationsverfahren ermöglicht es, beide Komponente mit unterschiedlicher Taktfrequenz zu betreiben, falls dies durch zukünftige Änderungen nötig sein sollte.

Schlüsselgenerator (KeyGen)

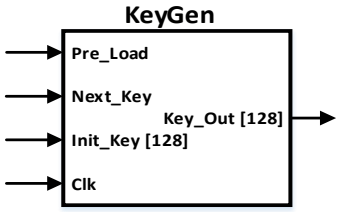


Abbildung 59 - KeyGen-Komponente

Der Schlüsselgenerator (*KeyGen*) produziert für jeden Block einen eigenen 128-Bit-*AES*-Schlüssel. Er besteht im Wesentlichen aus dem in Kapitel 3.5 beschriebenen Multiplexer-Generator. Da in nur einem Takt ein 128 Bit großer Zufallswert erzeugt werden soll, sind 128 separate 8-Bit-Schieberegister parallel geschaltet. Von diesen Schieberegistern wird jeweils ein Bit in das 128 Bit breite Ausgangsregister (*Key_Out*) geschrieben. Welches Bit dafür ausgewählt wird, bestimmt ein Multiplexer. Dessen Adressleitungen sind mit drei Bit eines weiteren Schieberegisters (*noise_reg_mux*) verbunden. Dieses besitzt eine Breite von 128 Bit, was eine lange Periodendauer garantiert. Wenn der Schlüsselgenerator Initialisiert wird (*Pre_Load='1'*), wird der Initialschlüssel (*Init_Key*) in das 128 Bit breite Schieberegister geladen. Die 8 Bit breiten Register werden mit konstanten Werten aus einem ROM initialisiert. Wird von außen das Signal *Next_Key* gesetzt, werden die Schieberegister getaktet und ein neuer Zufallswert in das Ausgangsregister geschrieben. Das Signal *Next_Key* wird mit dem Signal *AES_Ready* verknüpft, sodass nach jedem Verschlüsselungsvorgang ein neuer Schlüssel generiert wird. Eine Simulation dieses Vorgangs zeigt Abbildung 60.

Simulationsparameter: *clk = 40 MHz*

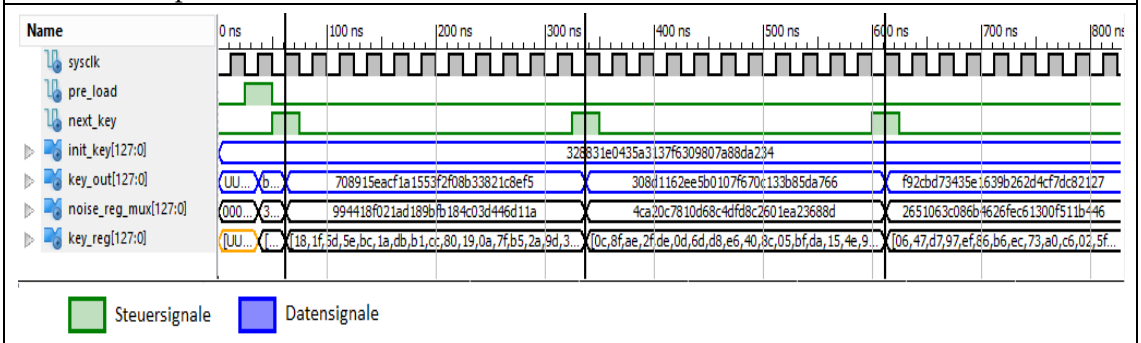


Abbildung 60 - Simulation des Schlüsselgenerators

Der Zufallsgenerator *NoiseGen* funktioniert nach dem gleichen Prinzip wie der Schlüsselgenerator *KeyGen*, nur dass dieser mit jedem Takt einen neuen Zufallswert in sein ebenfalls 128 Bit breites Ausgangsregister schreibt.

Steganographischer Algorithmus (StegoProc)

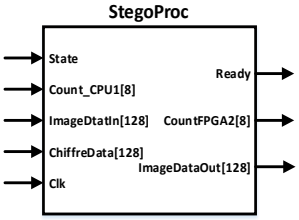


Abbildung 61 - StegoProc-Komponente

Mit der Komponente *StegoProc* (Abbildung 61) werden die verschlüsselten Daten aus dem BRAM mit den Schlüsselbits verknüpft und in die Bilddaten eingebettet. Dies geschieht, wenn sich der Zustandsautomat im Zustand *OPcrypt* befindet (siehe Abbildung 52).

Der aktuelle Zustand wird über den Eingang *State* abgefragt. Ist der entsprechende Zustand erreicht, werden die Bilddaten in 128-Bit-Blöcken von *CPU1* übertragen. Zur Synchronisation der Übertragung dient der Zählerausgang *CountFPGA2* und der Zählereingang *Count_CPU1* (siehe Abbildung 51). Die 128-Bit-Blöcke werden jeweils byteweise betrachtet. Somit besteht jeder Block aus 16 Bytes. Jedes Byte wird nun so in-, beziehungsweise dekrementiert, dass ihre niederwertigsten Bits mit den entsprechenden Bits der verschlüsselten Nachricht im BRAM übereinstimmen (siehe Kapitel 3.4). Anschließend wird aus jeden Byte ein Schlüsselbit ausgewählt. Das Auswahlverfahren wird in Kapitel 4.2.2 näher beschrieben. Die 16 Schlüsselbits werden

dann mit den niederwertigsten Bits des jeweiligen Bytes XOR-Verknüpft und das Ergebnis wieder an der niederwertigsten Stelle gespeichert. Ein Beispiel zeigt Abbildung 62.

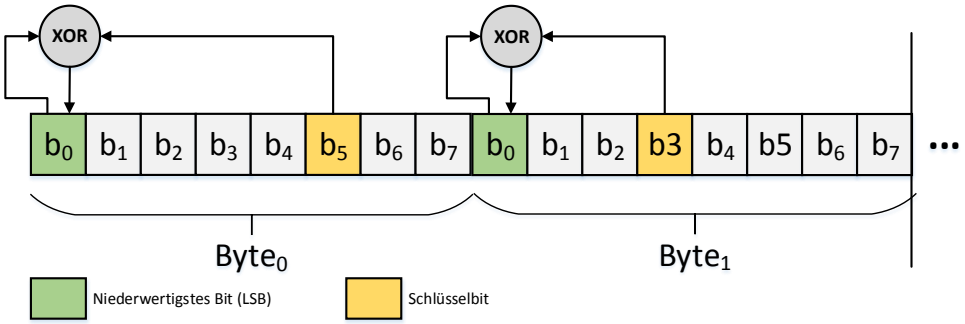


Abbildung 62 - Verknüpfung mit den Schlüsselbits

Die 16 so manipulierten Bytes werden anschließend zu *CPU1* übertragen. Für diesen Vorgang werden zwei interne Zähler verwendet. Der eine zählt die Anzahl der verarbeiteten Blöcke (*Count_Row*) und der andere die Anzahl der verarbeiteten Datenbits (*Count_Bit*). Da in jedem Block 16 Bits der Daten eingebettet werden, zählt der Zähler *Count_Bit* in 16er Schritten. Erreicht er den Wert 128 wurde ein Datensatz aus einer 128-Bit-Speicherstelle des BRAMs verarbeitet und der Zähler *Count_Row* wird um eins inkrementiert. Dieser zählt wie viele 128-Bit-Blöcke der verschlüsselten Daten bereits verarbeitet wurden und wird daher auch zur Adressierung des BRAM verwendet. Da bei dem Vorgang achtmal so viele Daten übertragen werden müssen, wie während des Verschlüsselungsvorgangs, werden hierfür insgesamt $7200 \cdot 8 = 57\,600$ Übertragungsschritte benötigt. Wurden alle Daten verarbeitet, wird das *Ready*-Signal der Komponente auf ‚1' gesetzt.

Simulationsparameter: *clk = 40 MHz*

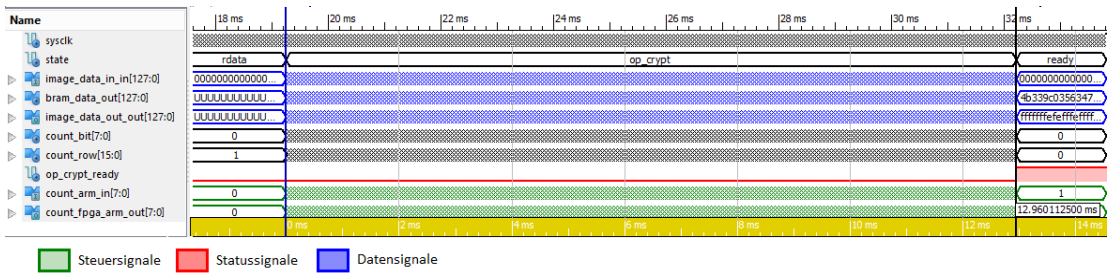


Abbildung 63 - Simulation StegoProc (gesamt)

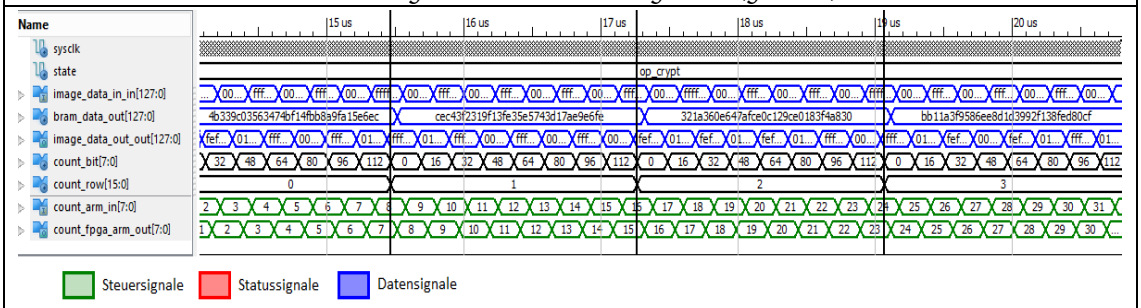


Abbildung 64 - Simulation StegoProc (Ausschnitt1)

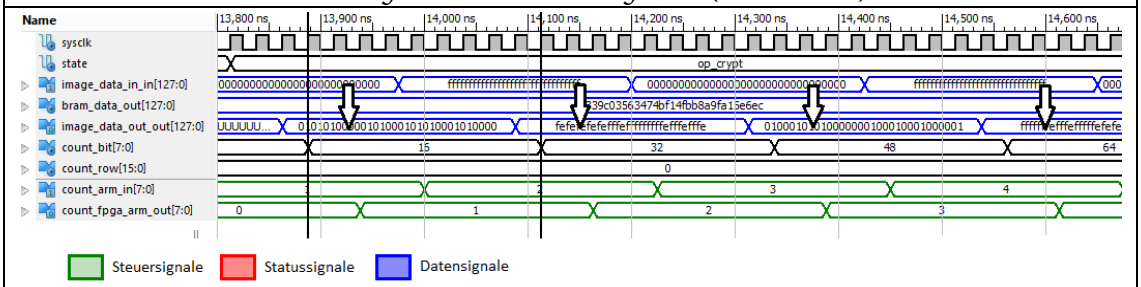


Abbildung 65 - Simulation StegnoProc (Ausschnitt2)

Tabelle 13 - Simulationsergebnisse des steganographischen Algorithmus

Die Simulationsergebnisse der Komponente zeigt Tabelle 13. Im unteren Abschnitt der Tabelle ist zu erkennen, wie der Zähler *Count_Bit* inkrementiert wird und wie die Daten zwischen dem Eingang *ImageData_In* und dem Ausgang *ImageData_Out* modifiziert werden (Abbildung 65). Im mittleren Teil der Tabelle (Abbildung 64) ist ein etwas größerer Ausschnitt abgebildet. Hier ist die Inkrementierung des Zählers *Count_Row* dargestellt. Die Cursor markieren die Zeitabschnitte, in denen jeweils ein 128-Bit-Block der verschlüsselten Nachricht verarbeitet wird. Der obere Teil der Tabelle zeigt die

Simulation des Gesamtvorgangs (Abbildung 63). Ihr ist zu entnehmen, dass bei einer Taktfrequenz von $f_{clk} = 40\ MHz$ etwa $13\ ms$ für einen kompletten Durchlauf benötigt werden. In dieser Zeit werden 921 600 Bit Nutzdaten übertragen. Daraus ergibt sich ein maximaler Datendurchsatz von $\sim 67{,}6\ Mbit/s$ für die Übertragung vom Verschlüsselungsmodul zur *CPU1*.

Um den Datendurchsatz des Gesamtsystems bestimmen zu können, also die Geschwindigkeit, mit der ein kompletter Datensatz (921 600 Bit) von *CPU0* zu *CPU1* übertragen wird, müssen die Laufzeiten von *AES_Cntr* und von *StegoProc* addiert werden. Mit $f_{clk} = 40\ MHz$ liegt dieser Wert bei $\sim 33\ ms$, woraus sich ein Durchsatz von $\sim 26{,}6\ Mbit/s$ für das Gesamtsystem ergibt.

Weitere Komponenten

Da das System so ausgelegt ist, dass *CPU0*, *CPU1* und das Verschlüsselungsmodul mit unterschiedlichen Taktfrequenzen betrieben werden können, sind noch weitere Komponenten auf dem Modul implementiert. Mit ihnen werden sämtliche Eingangssignale auf die entsprechende Frequenz synchronisiert um metastabile Zustände der Flip-Flops zu vermeiden. Außerdem wird bei einigen Signalen eine Flankendetektion durchgeführt, falls dies für die Funktion erforderlich ist.

5.3.2 Entschlüsselungsmodul

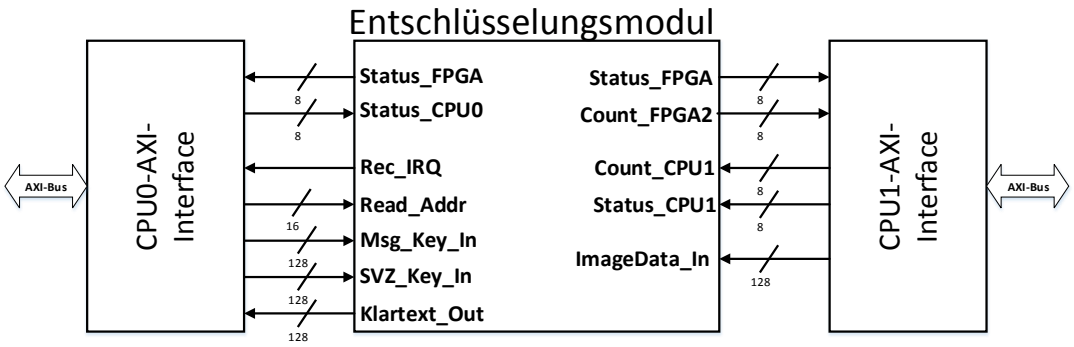


Abbildung 66 - Ein- und Ausgangsports des Entschlüsselungsmoduls

Das Entschlüsselungsverfahren ist als separates Modul auf dem FPGA implementiert. Die Ein- und Ausgangsports sind auf Abbildung 66 dargestellt. Zur Synchronisation während der Datenübertragung werden wieder Status- und Zähler-Ports verwendet.

Die Bilddaten des Steganogramms werden über den Dateneingang *ImageData_In* von *CPU1* in das Modul übertragen. Dort werden die Nutzdaten extrahiert und entschlüsselt. Erkennt das Modul in den Daten eine Nachricht vom SVZ oder einem anderen Teilnehmer, so wird das Interrupt-Signal *Rec_IRQ* gesetzt und die Nachricht über den Datenausgang *Klartext_Out* an *CPU0* übertragen.

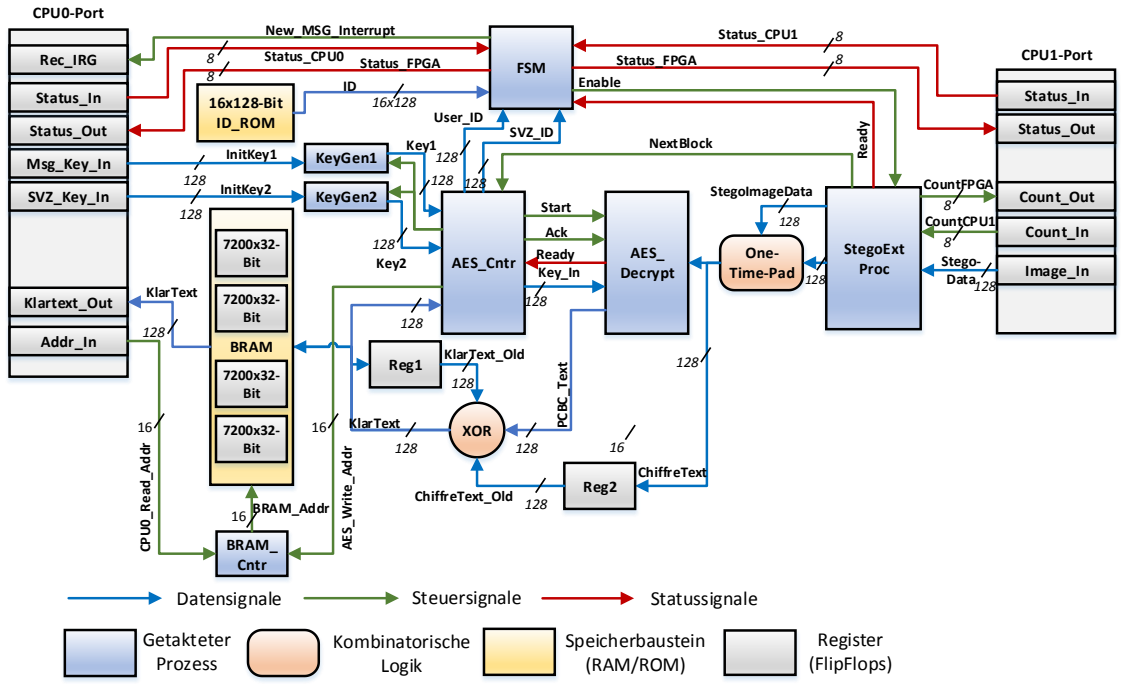


Abbildung 67 - Top-Level des Entschlüsselungsmoduls

Die oberste Ebene des Entschlüsselungsmoduls zeigt Abbildung 67. Es ist ähnlich aufgebaut wie das Verschlüsselungsmodul, nur dass die Daten die Komponenten in umgekehrter Reihenfolge durchlaufen und die Operationen der Komponenten invertiert werden. Ein Steganogramm, das von der Anwendung auf *CPU1* abgetastet wurde, wird in das Modul übertragen. Von der Komponente *StegoExtProc* werden die Nutzdaten aus dem Steganogramm extrahiert und anschließend mit den Schlüsselbits aus den Bilddaten verknüpft, um die *One-Time-Pad-Verschlüsselung* rückgängig zu machen.

Aus jedem übertragendem Bilddatenblock werden 16 Nachrichtenbits extrahiert. Nach acht Übertragungsschritten ist somit ein 128-Bit-Nachrichtenblock vollständig. Ist dies der Fall wird der *AES*-Kontroller *AES_Cntr* über das Signal *NextBlock* informiert. Anschließend wird der Nachrichtenblock mit der Komponente *AES_Decrypt*

entschlüsselt und in dem BRAM gespeichert. Daraufhin werden die nächsten 8 Bilddatenblöcke eingelesen. Die ersten beiden entschlüsselten Nachrichtenblöcke werden in den Registern *SVZ_ID*, beziehungsweise *User_ID,* gespeichert. Anhand deren Inhalt wird durch den Zustandsautomat geprüft, ob es sich bei den Daten um eine gültige Nachricht vom SVZ oder von einem anderen Teilnehmer handelt. Ist der Vorgang abgeschlossen, befindet sich der komplette Klartext im BRAM. Hat die Prüfung der Daten ergeben, dass es sich um eine Nachricht handelt, wird *CPU0* über ein Interrupt-Signal (*Rec_IRQ*) benachrichtig. Die CPU kann anschließend über den Adresseingang *Addr_In* und den Datenausgang *Klartext_Out* den Inhalt des kompletten BRAMs auslesen und weiter verarbeiten.

Für das Entschlüsselungsmodul werden zwei parallele Schlüsselgeneratoren verwendet, da das Modul in der Lage sein muss, die Nachricht sowohl mit dem SVZ-Schlüssel, als auch mit dem persönlichen Schlüssel des Teilnehmers zu entschlüsseln. Welcher Schlüssel genutzt wird, entscheidet der *AES*-Kontroller nach der Überprüfung des ersten entschlüsselten Blocks, der immer mit dem SVZ-Schlüssel entschlüsselt wird. Erkennt es dort eine von den 16 vordefinierten ID-Nummern, wird für die restlichen Blöcke der SVZ-Schlüsselstrom verwendet, ansonsten der Schlüsselstrom, der aus dem persönlichen Kommunikationsschlüssel generiert wird.

Zustandsautomat (FSM)

Wie im Verschlüsselungsmodul wird auch für das Entschlüsselungsmodul ein Zustandsautomat für die zeitliche Ansteuerung der Komponenten verwendet. Die einzelnen Zustände sind in dem Diagramm in Abbildung 68 dargestellt.

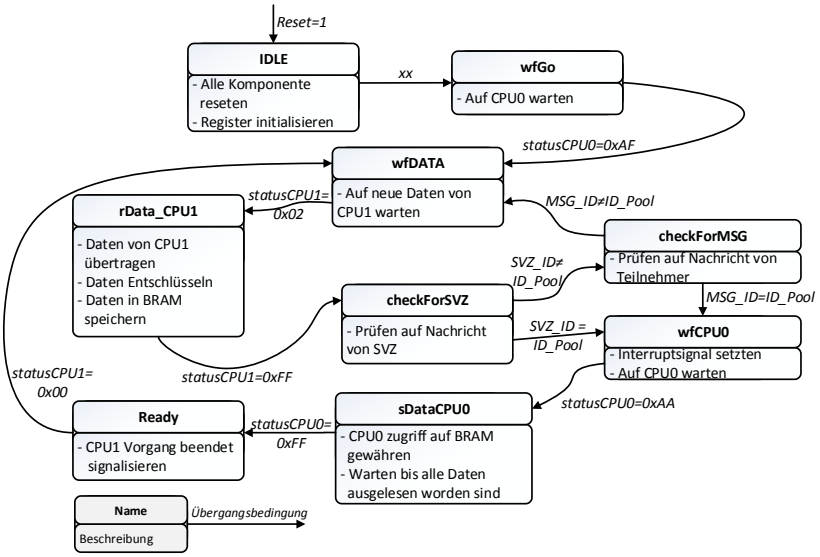


Abbildung 68 - Zustandsdiagramm des Entschlüsselungsmoduls

Nach dem im *IDLE*-Zustand alle nötigen Register in einen definierten Zustand gebracht worden sind und eine Synchronisation mit *CPU0* erfolgte (*wfGo*), wartet der Automat darauf, dass der Anwendung auf *CPU1* neue Bilddaten zur Verfügung stehen (*wfData*). Ist dies der Fall (*statusCPU1=0x02*), so werden die Bilddaten im Zustand *rData_CPU1* blockweise zum Modul übertragen. Dort werden die Nutzdaten extrahiert, entschlüsselt und im BRAM gespeichert. Ist der Vorgang beendet (*statusCPU1=0xFF*) prüft der Automat im Zustand *checkForSVZ*, ob sich im ersten BRAM-Register eine ID aus dem ID-Pool befindet (besteht aus 16 vordefinierten ID-Nummern). Ist dies der Fall, wird in den Zustand *wfCPU0* gewechselt, ansonsten in den Zustand *checkForMSG*. Dort wird geprüft, ob sich im zweiten Register eine ID-Nummer befindet. Befindet sich auch dort keine ID-Nummer, erfolgt ein Sprung zurück in den Zustand *wfData*, sonst zum Zustand *wfCPU0.* In diesem Zustand wird ein Interrupt-Signal gesetzt und auf die Antwort von *CPU0* gewartet (*statusCPU0=0xAA*). Daraufhin werden die Daten im Zustand *sDataCPU0* von *CPU0* aus dem BRAM ausgelesen. Ist der Vorgang beendet (*statusCPU0=AA*), setzt der Automat im *Ready*-Zustand *CPU1* darüber in Kenntnis. Erfolgt eine Bestätigung von *CPU1*, geht der Automat wieder zurück in den Zustand *wfData* und der Zyklus beginnt erneut.

AES-Entschlüsselung (AES_Decrypt)

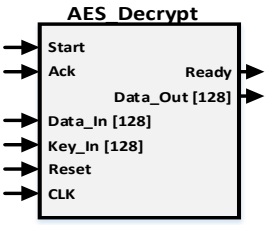


Abbildung 69 - AES_Decrypt-Komponente

Die Komponente *AES_Decrypt* dient dazu, die *AES*-Verschlüsselung wieder rückgängig zu machen. Sie besitzt die gleichen Ein- und Ausgangsports wie ihr Pendant im Verschlüsselungsmodul. Auch der interne Zustandsautomat ist identisch.

Um die Funktionsweise der Komponente zu überprüfen, wurde auch hier eine Simulation mit *ISim* durchgeführt. Dazu wurde der Ausgangsvektor, der durch die Simulation der *AES_Encrypt*-Komponente generiert wurde, als Eingangsvektor für die *AES_Decrypt*-Komponente verwendet. Das Ergebnis der Simulation zeigt, dass der Entschlüsselungsalgorithmus korrekt implementiert wurde (Abbildung 70), da der Wert im Ausgangsregister (*output_data*) identisch mit dem Wert ist, der für die *AES_Encrypt*-Komponente als Eingangsvektor genutzt wurde (siehe Abbildung 55).

Simulationsparameter: *clk = 40 MHz*

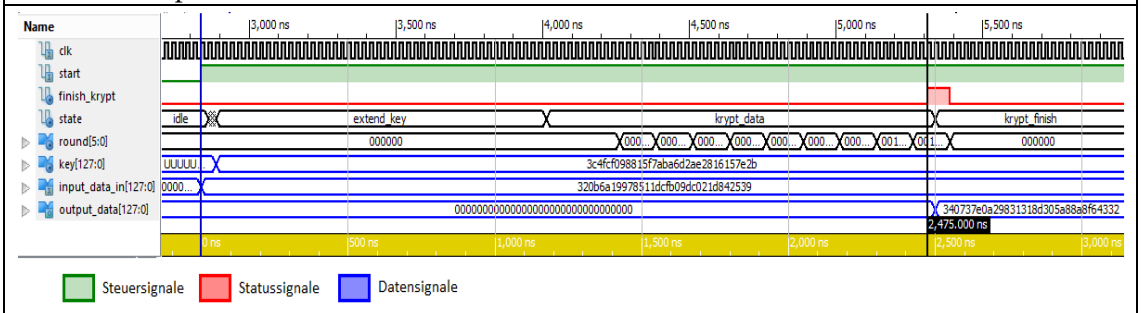


Abbildung 70 - Simulation der AES-Entschlüsselung

Weitere Komponenten

Die Funktionen der bisher nicht beschriebenen Komponenten des Entschlüsselungsmoduls sind entweder trivial oder ähneln stark denen des Verschlüsselungsmoduls, sodass ihre Funktionsweisen hier nur zusammenfassend beschrieben werden. Zu diesen Komponenten gehört die Komponente *StegoExtProc*. Deren Funktion besteht darin, aus dem 128-Bit-Bilddatenblock jedes 8te Bit (also die LSBs jedes Bytes) zu extrahieren und in einem Register zwischen zu speichern, bis 128 Nachrichtenbits zusammen sind. Die Übertragung der Bilddatenblöcke von *CPU1* wird wieder mit zwei Zählern realisiert.

Ähnlich wie im Verschlüsselungsmodul wird auch im Entschlüsselungsmodul ein Kontroller für die *AES*-Komponente implementiert (*AES_Cntr*). Die Synchronisation zwischen der *AES*-Komponente und dem Kontroller erfolgt wieder über die drei Signale *AES_Start*, *AES_Ready* und AES_*Ack*. Sobald die Komponente *StegoExtProc* einen 128-Bit-Nachrichtenblock extrahiert hat, wird der Entschlüsselungsvorgang von dem Kontroller initiiert und das Ergebnis anschließend im BRAM gespeichert. Da im Entschlüsselungsmodul der *AES*-Kontroller keine direkte Datenübertragung zur *CPU0* vornimmt, verfügt er nicht über die Zähler-Ein- und Ausgänge *Count_CPU0* und *Count_FPGA1*. Die Aktivierung erfolgt außerdem nicht über den Zustandsautomaten sondern über die *StegoExtProc*-Komponente.

Die beiden Schlüsselgeneratoren *KeyGen1* und *KeyGen2* sind identisch mit dem Schlüsselgenerator auf dem Verschlüsselungsmodul. Sie produzieren parallel die beiden Schlüsselströme aus dem *SVZ*- und dem persönlichen *AES*-Schlüssel.

Simulationsparameter: *clk = 40 MHz*

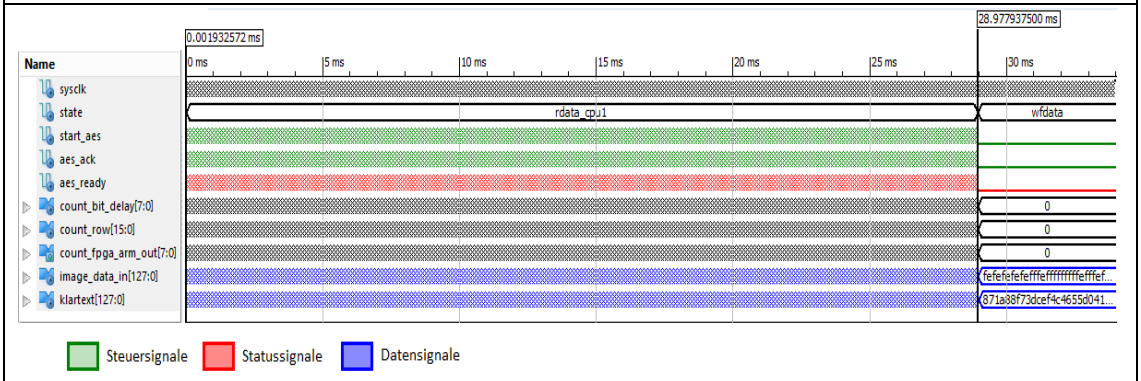


Abbildung 71 - Simulation des Entschlüsselungsvorgangs (gesamt)

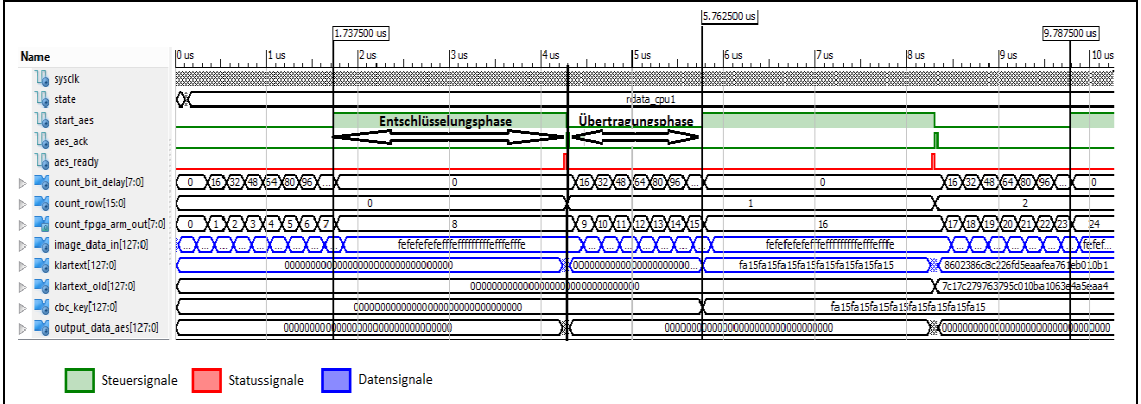


Abbildung 72 - Simulation des Entschlüsselungsvorgangs (Ausschnitt)

Tabelle 14 - Simulation des Entschlüsselungsmoduls

Tabelle 14 zeigt das Ergebnis der Verhaltenssimulation des Entschlüsselungsmoduls. Im oberen Teil der Tabelle (Abbildung 71) ist der komplette Entschlüsselungsvorgang eines Steganogramm dargestellt. Hierbei ist zu beachten, dass die Daten noch nicht zur *CPU0* übertragen, sondern lediglich im BRAM gespeichert wurden. Handelt es sich bei den Daten um eine Nachricht, so folgt noch ein Zustand, in dem die Daten von *CPU0* aus dem BRAM ausgelesen werden. In dem unteren Teil der Tabelle (Abbildung 72) ist ein Teilausschnitt des Gesamtvorgangs dargestellt. Hier ist zu erkennen, dass nach acht Übertragungsschritten eine Entschlüsselungsphase folgt, in der keine weiteren Daten übertragen werden. Erst wenn die Entschlüsselung beendet ist, folgt eine weitere Übertragungsphase.

5.3.3 Shared-Memory

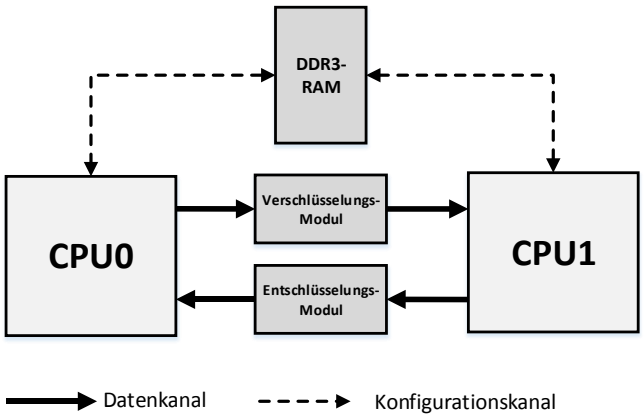


Abbildung 73 - Übertragungskanäle zwischen CPU0 und CPU1

Der Datenverkehr zwischen den beiden CPUs erfolgt hauptsächlich über das Ver-, beziehungsweise Entschlüsselungsmodul auf dem FPGA. Es wird jedoch noch ein weiterer Kanal implementiert, mit dem ein direkter Datenaustausch zwischen den beiden CPUs stattfinden kann (siehe Abbildung 73). Dieser Kanal besteht aus einem gemeinsam genutzten Speicherbereich im DDR3-RAM (*shared memory*). Über diesen Kanal übermittelt *CPU0* beispielsweise die vom *SVZ* empfangenen IP-Adressen, sowie die Anzahl der Teilnehmer, zur *CPU1*. Diese Informationen werden von der *Anwendung2* auf *CPU1* benötigt, da diese für das Abtasten der anderen Teilnehmer zuständig ist. Außerdem können so Linux-Anweisungen von *CPU0* aus initiiert werden, die beispielweise einen Neustart des Systems bewirken.

5.3.4 Systemkonfiguration

Jedes System besitzt nach der Initialisierung eine identische IP- und MAC-Adresse. Damit eine Kommunikation zwischen den Systemen stattfinden kann, müssen diese Adressen für jedes System individuell angepasst werden. Dies erfolgt mittels eines Linux-Skriptes, das während des Bootvorgangs aufgerufen wird. Dieses Skript (*init.sh*) befindet sich mit auf der SD-Karte und muss für jedes System so verändert werden, dass eine im Netzwerk einmalige MAC-Adresse vergeben wird. Die IP-Adresse kann entweder statisch oder durch einen entsprechenden Router dynamisch zugewiesen werden. Die jeweiligen IP-Adressen, über die das System im Netz erreichbar ist, müssen dem *SVZ* mitgeteilt werden. Von dort werden sie an die anderen Teilnehmer verteilt.

5.4 Schlüsselverteilungszentrale (SVZ)

Die Aufgabe der Schlüsselverteilungszentrale (*SVZ*) besteht darin, die Kommunikationsdaten der Teilnehmer über das Netz an diese zu verteilen (siehe Kapitel 4.2.1). Zu den Kommunikationsdaten gehören die IP-Adressen, die ID-Nummern und die persönlichen *AES*-Schlüssel der Teilnehmer. Die IP-Adressen und ID-Nummern werden von einer Textdatei eingelesen, die *AES*-Schlüssel werden bei Bedarf mit Hilfe des Zufallsgenerators neu erzeugt. Die Erzeugung und Verteilung neuer Kommunikationsdaten kann entweder über einen Taster oder zeitgesteuert initiiert werden.

Die Schlüsselverteilungszentrale benötigt in der in Kapitel 4.2.1 beschriebenen Realisierung nur ein Verschlüsselungsmodul, da lediglich eine unidirektionale Verbindung zu den Systemen der Teilnehmer aufgebaut wird. Auf ein Entschlüsselungsmodul kann daher verzichtet werden. Auch die separate *MicroBlaze*-CPU wurde eingespart. Dies reduziert den Ressourcenverbrauch des Systems, sodass die *SVZ* auch auf einem kleineren Chip der *Zynq-7000-All-Programmable-SoC*-Serie implementiert werden kann, wie beispielsweise dem *Z-7010*, der auf dem *ZYBO-Board* verbaut ist.

Das *ZYBO-Board*, welches ebenfalls von der Firma *DIGILENT* entwickelt wurde [31], besitzt ähnliche Peripheriegeräte wie das *Zedboard* (siehe Kapitel 5.1), mit Ausnahme des OLED-Displays. Dieses wird für die Funktion der *SVZ* auch nicht benötigt.

Im Anhang C dieser Arbeit befinden sich sowohl die Projektdateien für eine Realisierung der *SVZ* auf dem *Zedboard*, als auch für das *ZYBO*-Board. Letzteres hat den Vorteil, dass es kostengünstiger zu beschaffen und auf Grund der kleineren Bauform leichter zu transportieren ist. Die Systeme der Teilnehmer können nicht auf dem *ZYBO-Board* implementiert werden, da der FPGA in dem *Z-7010-Chip* nicht genügend Anwendungslogik zur Verfügung stellt.

6 Ergebnis/Fazit

In diesem Abschnitt wird das Ergebnis der in Kapitel 5 beschriebenen Implementierung untersucht. Dabei wird hauptsächlich der Aspekt der Sicherheit beurteilt.

6.1 Kryptographische Methoden

Den größten Anteil zur Sicherheit des Systems trägt das standardisierte Verschlüsselungsverfahren *AES* bei. Solange dieses Verfahren nicht gebrochen wird, kann das gesamte System als sicher eingestuft werden, da alle anderen implementierten Funktionen das Verfahren nicht beeinflussen sondern lediglich ergänzen. Hier könnten jedoch Fehler in der Implementierung des *AES*-Algorithmus die Sicherheit des Systems gefährden. Daher sollte vor einem Einsatz des Systems der Quellcode, mit dem das Verfahren implementiert wurde, noch von einer Drittperson überprüft werden[30]. Mehrere Tests und Vergleiche mit im Internet veröffentlichten *AES*-Verfahren (beispielsweise in [32]) deuten jedoch darauf hin, dass das Verfahren korrekt implementiert wurde.

Laut eines Berichtes von dem Nachrichtenportal Spiegel-Online, welcher während der Bearbeitungszeit dieser Arbeit veröffentlicht wurde, besitzt die NSA bereits eine „Handvoll Methoden [den AES] kryptographisch anzugreifen“ [33]. Inwiefern dies die Sicherheit des *AES* beeinflusst, kann anhand dieser Aussage jedoch nur schwer beurteilt werden. Offiziell gilt der *AES* noch als ein sicheres Verfahren. Sollte sich jedoch herausstellen, dass der *AES* gebrochen wurde, kann das Verfahren auf Grund des modularen Aufbaus des Systems leicht durch ein anderes Verschlüsselungsverfahren ersetzt werden.

Bei einem ungebrochenen symmetrischen Verschlüsselungsverfahren hängt die Sicherheit des Systems von der Geheimhaltung des verwendeten Schlüssels ab. Da die Schlüsselverteilung keinen Schwerpunkt dieser Arbeit darstellt, wurde ein einfaches Verfahren realisiert (siehe Kapitel 4.2.1). Dieses Verfahren sollte vor dem tatsächlichen

[30] Der gesamte Quellcode des Systems befindet sich in Anhang C.

Einsatz des Systems erweitert werden, in dem ein komplexeres Schlüsselverteilungsprotokoll umgesetzt wird.

Als zweite Sicherheitsstufe dient in dem System die *One-Time-Pad-Verschlüsselung*. Diese bietet jedoch nur Schutz, solange ein Angreifer nichts über den genutzten Algorithmus weiß, mit dem der Schlüssel aus den Bilddaten generiert wird. Mit dem Verfahren wird also das Prinzip „*Security through Obscurity*“ umgesetzt, welches sich besonders für Systeme eignet, deren Teilnehmerzahl beschränkt ist. Bei weit verbreiteten Systemen steigt die Gefahr, dass das Verfahren durch das sogenannte *reverse engineering* gebrochen wird. Daher sollte das Verfahren für jede Teilnehmergruppe modifiziert werden. Das in Kapitel 4.2.2 beschriebene Verfahren sollte also nur als Beispiel einer möglichen Implementierung betrachtet werden.

6.2 Steganographische Methoden

Um die Sicherheit der kryptographischen Methoden zu erhöhen, werden zusätzlich steganographische Methoden eingesetzt. Diese sollen dazu führen, dass das Versenden von verschlüsselten Nachrichten bei einem Gegner keinen Verdacht hervorruft und dieser somit keinen Aufwand betreibt, die kryptographischen Methoden zu brechen. Die steganographischen Methoden werden ebenfalls in zwei Stufen eingesetzt. Die erste Stufe besteht darin, den Klartext in Abhängigkeit eines steganographischen Schlüssels in einen Zufallsdatenstrom einzubetten (siehe Kapitel 4.2.3). Dieses Verfahren alleine bietet keinen besonders guten Schutz, da sich durch statistische Verfahren die Existenz von Klartext in Zufallsdaten leicht nachweisen lässt. Es dient hauptsächlich dazu, die Kryptoanalyse der Verschlüsselungsverfahren zu erschweren, da ein Klartext so nicht direkt als solcher zu erkennen ist. Außerdem demonstriert das Verfahren die Anwendung eines steganographischen Schlüssels.

In der zweiten Stufe wird die verschlüsselte Nachricht in Bilddaten einer USB-Kamera eingebettet. Diese Stufe dient der eigentlichen Aufgabe, die Existenz von verschlüsselten Nachrichten zu verschleiern. Da dies die letzte Stufe des Sicherheitssystems darstellt, muss hier die Sicherheit genauer untersucht werden. Daher werden im folgenden Abschnitt die in Kapitel 2.2.1 beschrieben Angriffe auf das System durchgeführt und die Ergebnisse analysiert.

6.2.1 Visueller Angriff

Um einen visuellen Angriff durchzuführen, wird der in Kapitel 2.2.1 beschriebene Filter eingesetzt.

Abbildung 74 - Containerbild

Abbildung 75 - Steganogramm

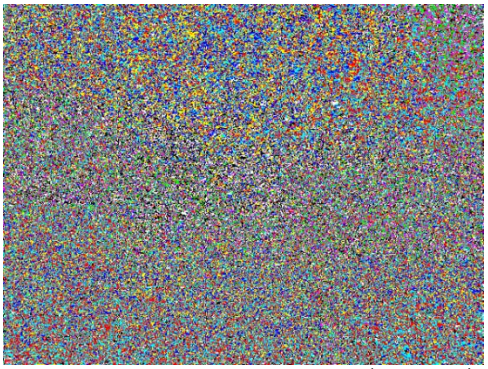


Abbildung 76 – Containerbild (gefiltert)

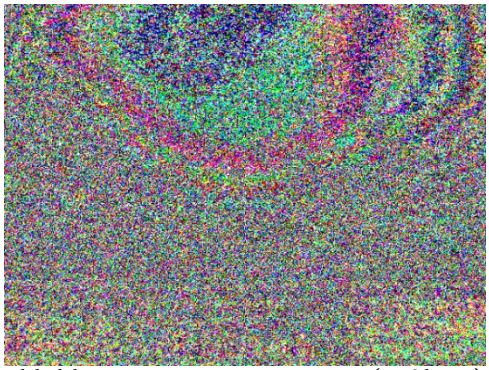


Abbildung 77 – Steganogramm (gefiltert)

Tabelle 15 - Visueller Angriff

Das Ergebnis des visuellen Angriffs zeigt Tabelle 15. Durch die Filterung ist zwar ein Unterschied zwischen dem originalen Containerbild (Abbildung 76) und dem Steganogramm (Abbildung 77) zu erkennen, doch zeigt sich hier auch ein weiterer positiver Aspekt der *One-Time-Pad-Verschlüsselung*. Da die verwendeten Schlüsselbits stochastisch abhängig vom Bildinhalt sind, zeigt sich in dem gefilterten Steganogramm nicht das Muster einer herkömmlich verschlüsselten Nachricht (siehe Kapitel 2.2.1). Ein Angreifer ohne Zugang zu dem Containerbild würde wohl auf Grund dieser Analyse

nicht den Verdacht schöpfen, dass eine Nachricht in den Bilddaten eingebettet wurde. Das Ergebnis könnte sogar einen Anfangsverdacht bei einem Angreifer beseitigen.

6.2.2 Statistischer Angriff

Für einen statistischen Angriff auf das System wird der in Kapitel 2.2.1 beschriebene *chi-square-Angriff* durchgeführt. Das Ergebnis des Angriffs zeigt Tabelle 16.

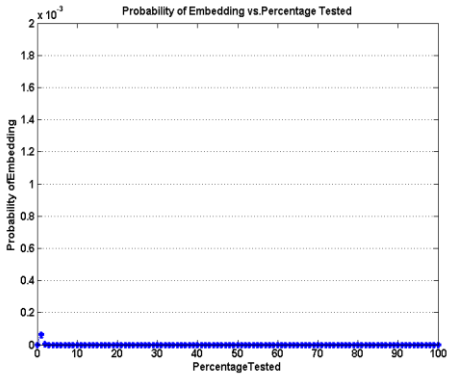


Abbildung 78 - Containerbild

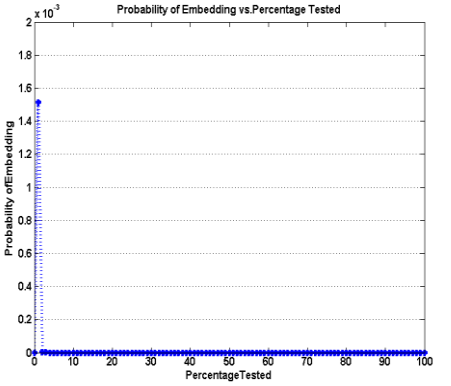


Abbildung 79 - Steganogramm

Tabelle 16 - Chi-square-Angriff

Auch hier zeigt das Ergebnis des Angriffs einen leichten Unterschied zwischen dem Containerbild (Abbildung 78) und dem Steganogramm (Abbildung 79). Dieser Unterschied zeigt sich durch einen etwas höheren Ausschlag zu Beginn der Analyse. Ein *stego-only-Angriff* würde jedoch auch hier wahrscheinlich keinen Verdacht auslösen, denn das Ergebnis zeigt lediglich mit einer Wahrscheinlichkeit von $\sim 0,15$ % an, dass sich in den ersten ~ 2 % des Bildes eine Nachricht befindet.

Das System hält den beschriebenen Angriffen also stand, solange die Containerdaten nicht in den Besitzt des Angreifer gelangen. Dies wird erschwert, in dem die Containerdaten vom System selbst erzeugt werden und dieses somit nicht verlassen. Hier muss jedoch auf eine eventuelle Schwachstelle der Implementierung hingewiesen werden. Die Containerdaten werden nach dem Auslesen der USB-Kamera in dem Arbeitsspeicher der Linux-CPU zwischengespeichert. Ein Angreifer, der sich Zugriff auf das Linux-System verschafft, könnte also die originalen Bilddaten auslesen und mit dem Steganogramm vergleichen. Ein solcher Angriff wird in der Regel jedoch nur dann stattfinden, wenn bereits ein hoher Anfangsverdacht vorliegt.

Sollte einem Angreifer bekannt sein, dass durch das System Steganogramme erzeugt werden, greift ein weiterer Sicherheitsmechanismus. Dieser besteht darin, dass nicht nur dann Steganogramme erzeugt und ausgetauscht werden, wenn eine Nachricht verschickt werden soll, sondern dass dies kontinuierlich geschieht. Da in jedem Steganogramm verschlüsselte Daten eingebettet werden, ist es für einen Angreifer nur sehr schwer festzustellen, in welchem Steganogramm sich tatsächlich eine verschlüsselte Nachricht befindet. Ein Angreifer müsste also jedes Steganogramm aufwändig kryptographisch analysieren um zu erkennen, ob es sich bei dem Klartext um eine Nachricht oder um Zufallsdaten handelt. Dies kann nur dann geschehen, wenn das kryptographische Verfahren bereits gebrochen wurde.

6.3 Ausblick

Im vorherigen Abschnitt wurden Aspekte angesprochen, welche die Sicherheit des Systems gefährden könnten. Diese sollten vor einem tatsächlichen Einsatz des Systems genauer geprüft und gegebenenfalls noch überarbeitet werden. Hinzu kommen Sicherheitsmechanismen, die im Rahmen dieser Arbeit nicht behandelt wurden, welche die Sicherheit des Systems jedoch noch weiter verbessern könnten. Dazu gehört beispielsweise der Schutz des Linux-Systems vor Hacker-Angriffen. Dieser könnte unter anderem durch den Einsatz von Firewalls erhöht werden. Auch sollten die Systemdateien, welche sich auf der SD-Karte befinden, vor äußerer Manipulation geschützt werden.

Zusätzlich könnte das Linux-System durch eine Client-Anwendung für das sogenannte Tor-Netzwerk (engl.: „*The-Onion-Routing-Network*“) erweitert werden. Die Nutzung des Tor-Netzes ermöglicht die Anonymisierung der Verbindungsdaten, wie beispielsweise die IP-Adressen der Teilnehmer, und würde das „Belauschen“ des kryptographischen Kanals für einen Angreifer erschweren [34].

Eine weitere Erweiterung des Systems könnte eine Update-Funktion für die Bootimage-Datei beinhalten. Eine mögliche Realisierung der Update-Funktion wäre, dass eine zentrale Updateverteilung eine neue Bootimage-Datei mittels Steganogramme an die jeweiligen Systeme verteilt. Dort könnte sie von *CPU0* über den gemeinsamen Speicherbereich an eine Linux-Anwendung weitergegeben werden, die die Bootimagedatei auf der SD-Karte durch die neue Version ersetzt und anschließend einen Neustart des Systems durchführt. Somit könnte sowohl die Anwendung für *CPU0* als auch die Hardwarekomponenten auf dem FPGA über das Netz aktualisiert werden. Ersteres ist besonders sinnvoll, wenn die Kommunikationsdaten des SVZ geändert werden sollen, da diese fest in der Anwendung von *CPU0* implementiert sind.

7 Zusammenfassung

In dieser Arbeit wird ein System beschrieben und umgesetzt, welches einen sicheren Nachrichtenaustausch zwischen mehreren Teilnehmern über ein öffentliches Netz ermöglicht. Sicherheit bedeutet dabei, dass der Inhalt der ausgetauschten Nachrichten vor Drittpersonen geschützt werden soll. Dafür wird eine Kombination aus steganographischen und kryptographischen Methoden eingesetzt. Die kryptographischen Methoden dienen dem Zweck, den Inhalt der Nachrichten für unberechtigte Personen unleserlich zu machen, während durch die steganographischen Methoden die bloße Existenz der Nachrichten verschleiert werden soll. Letzteres dient dazu, dass Angreifer keinen Aufwand betreiben, den kryptographischen Schutz der Nachricht zu brechen. Beide Methoden werden in einem zweistufigen Verfahren umgesetzt.

Um die Forderung nach einem transparenten Verschlüsselungsverfahren zu erfüllen, deren Sicherheit durch eine öffentliche Auseinandersetzung bestätigt werden konnte, wird der *Advanced Encryption Standard* eingesetzt. Dieser gilt nach aktuellem Stand als sicheres Verfahren. Für die Erzeugung der benötigten Schlüssel wird eine Schlüsselverteilungszentrale realisiert, welche die Schlüssel über das Netz an die Teilnehmer verteilt.

Um die Sicherheit des Systems weiter zu erhöhen, wird in einer zweiten Stufe ein Verschlüsselungsverfahren eingesetzt, welches auf dem Prinzip „*Security through Obscurity*“ beruht. Dieses basiert auf der *One-Time-Pad-Verschlüsselung* und bietet besonders bei kleinen Teilnehmergruppen einen zusätzlichen Schutz.

Da verschlüsselte Nachrichten leicht als solche zu erkennen sind, werden steganographische Methoden eingesetzt, um ihre Existenz zu verbergen. In einer ersten Stufe wird der Klartext einer Nachricht in einen Datenstrom aus Zufallszahlen eingebettet. Dazu wird ein steganographischer Schlüssel verwendet. Dies dient hauptsächlich dem Schutz der Verschlüsselungsverfahren vor kryptoanalytischen Angriffen. In einer zweiten Stufe wird die verschlüsselte Nachricht in einen Container aus Bilddaten eingebettet. Die so erzeugten Steganogramme können dann über ein öffentliches Netz unter den Teilnehmern ausgetauscht werden. Um den Zeitpunkt einer Nachrichtenübertragung zu verschleiern, werden kontinuierlich und mit

annährend konstanter Frequenz neue Steganogramme erzeugt und übertragen. Das System erkennt selbständig, in welchen dieser Steganogramme sich eine Nachricht befindet.

Das beschriebene System wird auf einem Entwicklungsboard, dem *Zedboard* von *DIGILENT*, implementiert. Dabei werden alle sicherheitskritischen Funktionen als Hardwarekomponente auf einem FPGA realisiert um die Gefahr vor dem Ausspähen der Funktionsweise, dem sogenannten *reverse engineering*, zu reduzieren.

Bei der Entwicklung des Systems wurde darauf geachtet, dass möglichst wenig externe Komponenten benötigt werden, um den Transport und den Aufbau des Systems zu erleichtern. Für den vollen Funktionsumfang werden lediglich ein externer USB-Kontroller zur Anbindung eines Datenträgers und eine USB-Kamera zur Erzeugung der Containerdaten benötigt. Alle weiteren Komponenten sind bereits auf dem Zedboard integriert.

Nach der Implementierung wurde das entwickelte System in einem Netz mit drei Teilnehmern und einer Schlüsselverteilungszentrale erfolgreich getestet und die Funktionsweise somit verifiziert.

Literaturverzeichnis

[1] C. Stobitzer, „Kryptowissen.de“. URL: http://www.kryptowissen.de/kryptologie.html. (Zugriff am 22.12.2014).

[2] W. F. /. H. P. Rieß, „Kryptographie“, München: Oldenbourg, 1994.

[3] U. Hebisch, „Skytale“. URL: http://www.mathe.tu-freiberg.de/~hebisch/cafe/kryptographie/skytale.html.(Zugriff am 10.11.2014).

[4] B. Schneier, „Angewandte Kryptographie“, Bonn: Addison-Weslay, 1996.

[5] „Spiegel Online“ (16.01.2000). URL: http://www.spiegel.de/netzwelt/web/kryptographie-usa-lockern-exportbeschraenkungen-a-59981.html. (Zugriff am 22.12.2014).

[6] C. E. Shannon, „Communication Theory of Secrecy Systems“, 1945.

[7] Wikipedia, „Kryptographie“. URL: http://de.wikipedia.org/wiki/Kryptographie. (Zugriff am 04.11.2014).

[8] F. Petitcolas, „The information hiding homepage“. URL: http://www.petitcolas.net/kerckhoffs/index.html. (Zugriff am 10.11.2014).

[9] W. J. M. T. Karen Scarfon, „Guide to General Server Security“. URL: http://csrc.nist.gov/publications/nistpubs/800-123/SP800-123.pdf. (Zugriff am 10.11.2014).

[10] H. B. N. T. S. Albrecht Beutelspacher, „Kryptografie in Theorie und Praxis“, Wiesbaden: GWV Fachverlagsgruppe GmbH, 2005.

[11] H. G. Côme Berbain, „On the Security of IV Dependent Stream Ciphers“. URL: http://www.iacr.org/archive/fse2007/45930256/45930256.pdf. (Zugriff am 10.11.2014).

[12] C. E. Shannon, „Mathematical Theory of Communication“ *Bell Systems Technical Journal* , Nr. 27, 1948.

[13] A. Klein, „Visuelle Kryptographie“, Heidelberg: Springer-Verlag, 2007.

[14] Otto-von-Guericke Universität Magdeburg C. Haberland, „Steganalyse“. URL: http://www-ivs.cs.uni-magdeburg.de/~dumke/Security/Haberland/steganalysis.html. (Zugriff am 11.12.2014).

[15] Technische Universität Dresden, A. Westfeld, „Angriffe auf steganographische Systeme“, Dresden .

[16] Technische Universität Dresden, A. Westfeld, A. Pfitzmann, „Attacks on Steganographic Systems“ . URL: http://users.ece.cmu.edu/~adrian/487-s06/westfeld-pfitzmann-ihw99.pdf. (Zugriff am 11.12.14).

[17] Iowa State University, C. A. Stanley, „Pairs of Values and the Chi-squared Attack“ (01.05.2005). URL: http://orion.math.iastate.edu/dept/thesisarchive/MSCC/CStanleyMSSS05.pdf. (Zugriff am 11.12.2014).

[18] K. Schmeh, „Kryptografie“, Heidelberg: dpunkt.verlag, 2009.

[19] E. Zabala, „Rijndeal Cipher“ CryptTool. URL: http://www.formaestudio.com/rijndaelinspector/archivos/Rijndael_Animation_v4_eng.swf. (Zugriff am 05.12.2014).

[20] V. R. Joan Daemen, „The Design of Rijndeal“, Berlin: Springer Verlag, 2001.

[21] D. Kahn, „The Codebreakers: The Story of Secret Writing“, New York: Macmillan Publishing Co., 1967.

[22] S. K. S. Labitzke, „*Block- und Stromchiffren*“, TU-Berlin - Fakultät IV - Informatik.

[23] AVNET, „ZedBoard“. URL: http://www.em.avnet.com/en-us/design/drc/Pages/Zedboard.aspx. (Zugriff am 31.01.2015).

[24] Xilinx, „Zynq-7000 All Programmable SoC Overview“. URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. (Zugriff am 31.01.2015).

[25] Xilinx, „MicroBlaze Processor Reference Guide“. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug984-vivado-microblaze-ref.pdf. (Zugriff am 31.01.2015).

[26] „LinuxTV“. URL: http://linuxtv.org/wiki/index.php/Gspca_devices. (Zugriff am 15.01.2015).

[27] FTDI, „Vinculum VNC1L Module Datasheet“. URL: http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_VDIP1.pdf. (Zugriff am 31.01.2015).

[28] F. Abdolian, „GitHub: Zedboard_Oled_Display_Demo“. URL: https://github.com/faab64?tab=repositories. (Zugriff am 23.01.2015).

[29] Xilinx, „ISE Design Suite“. URL: http://www.xilinx.com/products/design-tools/ise-design-suite.html. (Zugriff am 31.01.2015).

[30] J. Reichardt, „Lehrbuch Digitaltechnik“, München: Oldenbourg Wissenschaftsverlag GmbH, 2009.

[31] DIGILENT, „Zybo Reference Manual“. URL: http://www.digilentinc.com/data/products/zybo/zybo_rm_b_v6.pdf. (Zugriff am 08.02.2015).

[32] „AES – Symmetric Ciphers Online“. URL: http://aes.online-domain-tools.com. (Zugriff am 09.02.2015).

[33] Spiegel Online, „NSA, GCHQ und Co.: Die Unsicherheitsbehörden“ (29.12.2014). URL: http://www.spiegel.de/netzwelt/netzpolitik/nsa-attacke-gegen-das-internet-angriff-auf-unsere-sicherheit-a-1010550.html. (Zugriff am 02.02.2015).

[34] Wikipedia, „Tor (Netzwerk)“. URL: http://de.wikipedia.org/wiki/Tor_(Netzwerk). (Zugriff am 03.02.2015).

[35] Wikipedia, „Reverse Engineering“. URL: http://de.wikipedia.org/wiki/Reverse_Engineering. (Zugriff am 22.12.2014).

[36] D. Tiger, „Fotocommunity“. URL: http://www.fotocommunity.de/pc/pc/mypics/694800/display/5106829. (Zugriff am 21.01.2015).

[37] Wikipedia, „Chi-Quadrat-Test“. URL: http://de.wikipedia.org/wiki/Chi-Quadrat-Test. (Zugriff am 21.01.2015).

[38] Xilinx, „ZedBoard Hardware User's Guide v1.1“ (01.08.2012). URL: http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf. (Zugriff am 09.02.2015).

[39] Wikipedia, „IEEE 802“. URL: http://de.wikipedia.org/wiki/IEEE_802. (Zugriff am 09.02.2015).

Anhang

Der komplette Anhang dieser Arbeit wird als DVD-ROM mitgeliefert. An dieser Stelle erfolgt lediglich eine Übersicht über den Inhalt der DVD.

Anhang A

Im Anhang A befinden sich sämtliche Matlab-Skripte, die für diese Arbeit entwickelt, beziehungsweise eingesetzt wurden. Die Skripte wurden mit der Matlab-Version R2013a erstellt und getestet.

Steganalyse

- *StegoProc.m*
  Matlab-Skript zum Testen des steganographischen Algorithmus und der *One-Time-Pad-Verschlüsselung*.
- *VisualAttack.m*
  Matlab-Skript zur Durchführung eines visuellen Angriffs auf ein Testbild.
- *pov3.m*
  Matlab-Skript zur Durchführung eines *Chi-Square-Angriffs* auf ein Testbild.
- *POVs.m*
  Matlab-Skript zur Analyse eines Testbildes auf POVs.
- *SubLSBs.m*
  Matlab-Skript zum Testen eines steganographischen Algorithmus durch Substitution der LSBs.
- *Container.bmp*
  Ein durch das System erzeugtes Containerbild.
- *Steganogramm.bmp*
  Ein durch das System erzeugtes Steganogramm.
- *TigerGrey.bmp*
  Ein Testbild.

Kryptoanalyse

- *simApost.m*
  Matlab-Skript zur Simulation eines statistischen Angriffs.
- *apost.m*
  Funktion zur Berechnung der a posteriori Wahrscheinlichkeiten.
- *encrypt.m*
  Funktion zur Berechnung der Verschlüsselung.

Anhang B

Im Anhang B befinden sich die C-Programme, die im Rahmen der Entwicklung genutzt wurden, um verschiedene Funktionsweisen zu Testen. Dabei handelt es sich jedoch nicht um die Software, die für den Betrieb des Systems erstellt wurde. Entwickelt wurden die Programme mit dem Entwicklungstool *Dev-C++* (Version 5.6.2).

- *AES-Encryption* (Ordner)
  C-Programm zum Testen des AES-Verschlüsselungsalgorithmus.
- *AES-Decryption* (Ordner)
  C-Programm zum Testen des AES-Entschlüsselungsalgorithmus.
- *Betriebmodi* (Ordner)
  C-Programm zum Testen des ECB- und PCBC-Betriebsmodus

Anhang C

In diesem Anhang befinden sich die Quell- und Projektdateien, mit dem das System implementiert wurde. Die Dateien wurden mit der *ISE Design Suit 14.7* entwickelt.

- *System_Zedboard* (Ordner)
  Enthält die Projekt-Ordner (*ISE*, *XPS*, *SDK*) zur Implementierung des Systems auf dem *Zedboard*.
- *SVZ_Zybo* (Ordner)
  Enthält die Projekt-Ordner (*XPS*, *SDK*) zur Implementierung der *SVZ* auf dem ZYBO-Board.
- *SVZ_ZedBoard* (Ordner)
  Enthält die Projekt-Ordner (*XPS*, *SDK*) zur Implementierung der *SVZ* auf dem *Zedboard*.

- *Quelldateien* (Ordner)
  Enthält sämtliche Quelldateien, die für das System entwickelt wurden (*Software*, *Hardware*).

Anhang D

Im Anhang D befinden sich die Dateien, die für eine Inbetriebnahme des Systems auf einem *Zedboard* benötigt werden. Diese Dateien müssen auf eine SD-Karte kopiert und das Zedboard über die SD-Karte gebootet werden.

- *SD_SystemZedBoard* (Ordner)
  Systemdateien für das System eines Teilnehmers auf dem *Zedboard*.
- *SD_SVZ_ZedBoard* (Ordner)
  Systemdateien für das *SVZ* auf dem *Zedboard*.
- *SD_SVZ_ZyboBoard* (Ordner)
  Systemdateien für das *SVZ* auf dem *ZYBO-Board*.