Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Noushin Mohammadi

Implementierung eines neuen JPEG-basierten Steganographie- und Kryptographie-Verfahrens mit Mikrocontroller-realisierter Webseite

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Noushin Mohammadi

Implementierung eines neuen JPEG-basierten Steganographie- und Kryptographie-Verfahrens mit Mikrocontroller-realisierter Webseite

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
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. rer. nat. Henning Dierks

Abgegeben am 10. April 2017

Noushin Mohammadi

Thema der Bachelorthesis

Implementierung eines neuen JPEG-basierten Steganographie- und Kryptographie-Verfahrens mit Mikrocontroller-realisierter Webseite

Stichworte

JPEG, PNG, Huffman, Steganographie, Kryptographie, RSA, Raspberry Pi2, HTML, Python, Flask

Kurzzusammenfassung

Diese Arbeit beschreibt die Entwicklung einer Software, die steganographische und kryptographische Verfahren kombiniert, um einen sicheren Austausch von Nachrichten zwischen mehreren Teilnehmern über das Internet zu ermöglichen. Zur Erhöhung der Sicherheit wird auf einem Microcontroller ein eigener Webserver realisiert.

Noushin Mohammadi

Title of the paper

Implementation of a new JPEG-based steganography and cryptography process with microcontroller-realized website

Keywords

JPEG, PNG, Huffman, Steganography, Cryptography, RSA, Raspberry Pi2, HTML, Python, Flask

Abstract

Inside this report the development of a system which uses a combination of steganographic and cryptographic methods to enable secure communication between multiple users over the Internet is described. To increase security, a separate web server is implemented on a microcontroller.

Inhaltsverzeichnis

Tabellenverzeichnis 7

Abbildungsverzeichnis 8

Abkürzungsverzeichnis 10

I. Einleitung 12

1. Zielsetzung 13
1.1. Gliederung 14

II. Analyse 15

2. Bildformat 16
2.1. PNG 17
2.2. JPEG 21

3. JPEG Kompression 23
3.1. Farbkonvertierung 25
3.2. Subsampling 26
3.3. DCT - Diskrete Kosinustransformation 30
3.4. Quantisierung 32
3.5. Zick-Zack-Umsortierung 35
3.6. Koeffizienten-Kodierung 36
3.7. Untersuchung eines Beispielbildes 39

4. Kryptographie 49
4.1. Symmetrische Verschlüsselung vs. Asymmetrische Verschlüsselung 49
4.2. Hybridverfahren 51
4.3. RSA 52
4.3.1. RSA-Schlüssel generieren 53
4.3.2. Eulersche Phi-Funktion 53

4.3.3. Erweiterter euklidischer Algorithmus 54
4.3.4. Satz von Euler-Fermat 56
4.3.5. Square-and-Multiply-Algorithmus 57
4.3.6. CRT in RSA 58

5. Huffman **59**
5.1. Huffman-Codierung 59
5.1.1. Huffman-Baum 59
5.1.2. Huffman-Code 60
5.2. Huffman-Decodierung 60

III. Entwicklung **61**

6. Huffman-Codierung **62**
6.1. Huffman-Decodierung 63

7. Kryptogeraphie **64**
7.1. Schlüssel-Generator 65
7.2. Verschlüsseln und Entschlüsseln 65

8. Steganographie **67**
8.1. Realisierung Encoder 70
8.2. Realisierung Decoder 72

9. GUI - Graphical User Interface **77**

10. Webserver auf Raspberry Pi zur Übermittlung der Bilddateien **79**
10.1. Raspberry Pi 79
10.2. Programmiersprache Python 79
10.3. Webframework Flask 80
10.4. Webserver 80
10.5. Raspberry Pi im www 82

IV. Auswertung **84**

11. Testfälle **85**
11.1. Fall 1: Vergleich der Histogramme 85
11.2. Fall 2: Filter auf Bild und Stego 87
11.3. Fall 3: Symmetrie-Vergleich 89

12. Fazit **90**

13. Ausblick **92**

V. Software Anleitung **93**

14. Anleitung **94**
14.1. Fehler-Meldungen 98

Literaturverzeichnis **99**

Anhang **102**

Tabellenverzeichnis

2.1. Hexadezimal-Darstellung der Bilddateien 17
2.2. PNG Datei-Format-Analyse 20
2.3. Marker im JPEG-Format 22

3.1. Verschiedene Chrominanz-Subsampling-Methoden 28
3.2. Standard-Quantisierungs-Tabelle für Luminanz 33
3.3. Standard-Quantisierungs-Tabelle für Chrominanz 33
3.4. Darstellung der DC-Koeffizienten-Differenz-Werte 36
3.5. Darstellung der AC-Koeffizienten-Werte 37
3.6. Erläuterungen zum DQT-Abschnitt 40
3.7. Huffman-Luminanz-DC(**DHT 00**) ohne Bitfolgen 43
3.8. Huffman-Luminanz-DC(**DHT 00**) mit Bitfolgen 43
3.9. Huffman-Luminanz-AC(**DHT 10**) ohne Bitfolge 45
3.10. Huffman-Luminanz-AC(**DHT 10**) mit Bitfolge 46
3.11. Luminanz Koeffizienten Y00 von MCU00 48

4.1. Merkmale symmetrischer Kryptographie vs. asymmetrischer Kryptographie . 50

7.1. Liste der Funktionen im Bereich Kryptographie 64

8.1. Funktionsliste(1.Teil) der Steganographie 75
8.2. Funktionsliste(2.Teil) der Steganographie 76
8.3. Liste der Ausgabe-Funktionen 76

9.1. Liste der Funktionen der GUI 78

11.1. Bild vs. Steganogramm 86
11.2. Bild-Histogramm vs. Steganogramm-Histogramm 86
11.3. Filter auf Bild und Stego 88

Abbildungsverzeichnis

2.1. Chunk layout mit vier Feldern 18
2.2. Chunk layout mit drei Feldern 18
2.3. PNG-Datei in Hexadezimal-Ansicht(Quelle: Testbild1.png) 19
2.4. Test-Bild.jpg /640x480/ 50% Qualität 21
2.5. Test-Bild.jpg/Hex Ansicht 22

3.1. JPEG Kompressions-Schema 24
3.2. Verschiedene Chrominanz-Subsampling-Methoden 29
3.3. Beispiel für die Informationsverluste durch die Quantisierung und Rundung . 34
3.4. Zick-Zack-Ablesung der Koeffizienten 35
3.5. Umsortiertes Koeffizienten-Array 35
3.6. 8x8-Matrix 38
3.7. Das Beispielbild mit 50% Qualität 39
3.8. DQT-Marker-Abschnitt(Define Quantization Table) 40
3.9. SOF0-Marker-Abschnitt 41
3.10. DHT-Marker-Abschnitt 42
3.11. SOS-Marker-Abschnitt 47

4.1. Hybridverfahren mit RSA 51

6.1. Ablauf Huffman-Encodierung 62
6.2. Beginn einer Huffman-codierten Textdatei 63
6.3. Ablauf Huffman-Decodierung 63

7.1. Schlüssel-Generator 65
7.2. Nachrichtenpakete vor und nach Verschlüsselung 66
7.3. Die gesamte Nachricht nach der RSA-Kodierung 66

8.1. JPEG-Encoder mit Manipulationsfunktion 67
8.2. JPEG-Decoder mit Manipulationsfunktion 68
8.3. init()Funktion 70
8.4. G_main1()Funktion 70
8.5. G_main2()-Funktion und die aufgerufenen Funktionen 71
8.6. Ablaufplan zur Herstellung eines Steganogramms 71

8.7. Datei im Container Bild 72
8.8. Steganogramm-Decoder: Header-Erkennung 73
8.9. Ablaufplan Decoder 74

10.1. Verzeichnisstruktur auf dem Raspberry Pi 80
10.2. Feste IP-Adresse für Raspberry Pi 83
10.3. Port Forwarding für Raspberry Pi 83
10.4. Externe IP-Adresse des Routers 83

11.1. Auswertung der Symmetrie-Änderung 89
11.2. Auswertung der Symmetrie-Änderung 89

12.1. Erstellte Webseite mit Testbildern 91

14.1. Hauptfenster mit Options-Menü 94
14.2. Key-Generator 95
14.3. Decrypt-Fenster zur Entschlüsselung von Stegos 95
14.4. Encrypt-Fenster 96
14.5. JPG-Info-Fenster 97

Abkürzungsverzeichnis

NN	Not Null
N3	Not 0, Not 1, Not -1
GUI	Graphical user interface
MCU	Minimum coded unit
JPEG	Joint Photographic Experts Group
PNG	Portable Network Graphics
DCT	Discrete Cosine Transform
Stego	Steganogramm
CRC	Cyclic redundancy check
IHDR	Image header
IDAT	Image data
sRGB	Standard RGB colour space
pHYs	Physical pixel dimensions
gAMA	Image gamma
IEND	Image trailer
PLTE	Palette
JFIF	JPEG File Interchange Format
Exif	Exchangeable image file format
WSGI	Web Server Gateway Interface
bpp	bit pro Pixel
RLZ	Run length Zero
LSB	Least significant bit

EOB End Of Block

SOI Start of Image

DQT Define Quantization Table

SOF Start of Frame

DHT Define Huffman Table

SOS Start of Scan

EOI End of Image

VLI Variable Length Integer

VLC Variable Lenght Code

RSA Ron Rivest, Adi Shamir, Leonard Adleman

CRT Chinese remainder theorem

FLINT Fast Library for Number Theory

ggT Größter gemeinsamer Teiler

kgV Kleinste gemeinsame Vielfache

ECC Elliptic Curve Cryptography

DES Data Encryption Standard

3DES Triple DES

AES Advanced Encryption Standard

RGB Red, Green, Blue

YCbCr Luminance, Blue/Yello, Red/Green

dpi dots per inch

BBS-Generator Blum-Blum-Shub-Generator

EOP End of Paket

HTML Hyper Text Markup Language

WinAPI Windows application programming interfaces

Teil I.

Einleitung

1. Zielsetzung

Ziel dieser Arbeit ist es, eine Software zu implementieren, die ein steganographisches Verfahren mit einem kryptographischen Verfahren kombiniert, um eine sichere Kommunikation bzw. einen sicheren Austausch von geheimen Botschaften im Internet zwischen bestimmten Teilnehmern zu ermöglichen. Dazu muss die Software so entwickelt sein, dass mit ihr eine Reihe von Untersuchungen möglich sind. Durch die Bereitstellung vieler Informationen, soll überprüft werden können, ob ein Bild für das Verfahren geeignet ist und ob ein erstelltes Steganogramm[1] sicher ist. Es werden verschiedene Aspekte der Originalbilder und der Steganogramme untersucht und verglichen.
Unter Steganographie versteht man das Verstecken einer Datei in einer anderen Datei. Die Datei, in der eine andere Datei versteckt wird, wird „Container Medium“ oder kurz Container[2] genannt. Für Steganographie gibt es verschiedene Träger-Medien z.B. Text, Audio, Video oder Bilder.
Da die Nachricht in einem anderen Medium versteckt wurde, kann ein eventueller Angreifer nicht ohne weiteres erkennen, dass hier vertrauliche Informationen versendet wurden. In dieser Arbeit werden Bild-Dateien als Container benutzt, da Bilder im Vergleich zu Audio- oder Video-Dateien deutlich kleiner sind und am unauffälligsten im Netz ausgetauscht werden können.
Um die hergestellten Steganogramme sicher im Netz austauschen zu können, wird ein Mikrocontroller als eigener Webserver dienen. So wird verhindert, dass die Stego auf einem fremden Server gespeichert werden müssen.
Für den Fall, dass ein Angreifer doch erkennt, dass eine vertrauliche Nachricht in dem Bild versteckt wurde, wird die Nachricht mit einer kryptographischen Methode verschlüsselt, bevor sie im Bild versteckt wird.

Verwendete Hard- und Software
Als Systemkamera für diese Arbeit wird die „Hercules Classic Silver Webcam“ genutzt. Die Aufnahmen werden mit der Software „WebCamImageSave“ gemacht.
Als Microcontroller für den Webserver wird der „Raspberry Pi 2 Model B“ eingesetzt. Für den Datenaustausch mit dem Raspberry werden die Programme „WinSCP“ und „Putty“ genutzt.

[1] Ein Steganogramm (Stego) ist das Ergebnis einer steganographischen Methode
[2] Deutsch: Träger

Die Entwicklung der Software erfolgt unter „Microsoft Visual Studio 2012“ in den Programmiersprachen C und C++. Hier werden die FLINT- und die OpenCV-Bibliothek genutzt.

1.1. Gliederung

II. Analyse
In Kapitel 2 wird eine erste Analyse der Bildformate PNG und JPEG durchgeführt. Da das JPEG-Format mit seiner hohen, aber verlustbehafteten Kompression wesentlich interessanter ist und das JPEG-Format wesentlich verbreiteter ist, als das PNG-Format, fiel die Entscheidung, dieses näher zu untersuchen.
In Kapitel 3 folgt eine genaue Analyse der JPEG-Kompression, um eine Möglichkeit zu finden, JPEG-Bilder als Container für Nachrichten zu verwenden.
Da zusätzlich zur Steganographie auch ein Kryptographie-Verfahren zur Erhöhung der Sicherheit zum Einsatz kommen soll, werden in Kapitel 4 einige Kryptographie-Verfahren untersucht und vorgestellt.
Aufgrund der hohen Kompressionsraten des JPEG-Formats, bieten JPEG-Dateien als Container nicht sehr viel Platz. Daher soll die Huffman-Codierung angeboten werden, um die Nachricht vorab zu komprimieren. Diese wird in Kapitel 5 vorgestellt.

III. Entwicklung
Der dritte Teil beschreibt die Entwicklung der Software. Hier wird zuerst die Umsetzung der Huffman-Codierung zur Komprimierung der Original-Nachricht kurz beschrieben(Kapitel 6).
Das Kapitel 7 beschreibt das eingesetzte Kryptographie-Verfahren. In Kapitel 8 wird dann genauer beschrieben, wie die Nachricht in den entschlüsselten Koeffizienten der JPEG-Datei versteckt wird.
Um die Software anwendungsfreundlicher zu machen, wurde eine graphische Benutzeroberfläche(GUI) entwickelt, die in Kapitel 9 beschrieben wird.
In Kapitel 10 wird ein kleiner Webserver auf dem Raspberry Pi aufgebaut, um die Stegos temporär auf einer eigenen Webseite anzubieten.

IV. Auswertung
Im vierten Teil werden einige Testfälle ausgewertet(Kapitel 11) und ein Fazit(Kapitel 12) gezogen. Es folgt ein kurzer Ausblick(Kapitel 13).

V. Software Anleitung
Im letzten Teil wird eine Anleitung(Kapitel 14) zur Nutzung der Software zur Verfügung gestellt.

Teil II.

Analyse

2. Bildformat

Joint Photographic Experts Group (JPEG) und Portable Network Graphics (PNG) sind die am häufigsten benutzten Bildformate für digitale Bilder im Internet. Dabei besitzen sie sehr verschieden Merkmale, wie zum Beispiel verschiedene Kompressionsmethoden.
PNG ist ein verlustfreies Grafikformat, während JPEG eine verlustbehaftete Kompression[1] nutzt. Werden Testbilder vom gleichen Objekt in diesen beiden Formaten verglichen, sind die Bilder im PNG-Format ungefähr um den Faktor 4 größer(Tabelle 2.1). Somit ist JPEG durch die Beanspruchung von weniger Speicherplatz für Bildansichten mit geringer Auflösung im Internet ideal.
PNG ist zudem nicht mit allen Anwendungen kompatibel, wohingegen JPEG zu nahezu allen Anwendungen kompatibel und damit das universell nutzbarste Bildformat ist.

Um einen besseren Blick auf den Aufbau der beiden Bildformate zu schaffen, wird das PNG-Format in Kürze und das JPEG-Format ausführlich unter die Lupe genommen. In Tabelle 2.1 sind zwei Test-Bilder[2] in den beiden Formaten dargestellt. Die rechte Seite der Tabelle zeigt die byteweise Darstellung der Bilddaten. In Kapitel 2.1 folgt ein kurzer Einblick und einige Erläuterungen über den Aufbau des PNG-Bildformats. Die Betrachtung des JPEG-Formats folgt ausführlich im Hauptteil dieser Arbeit.

[1] Datenkompression bedeutet, die Codierung von Informationen, so dass diese mit weniger Bits dargestellt werden können, als die Originaldaten. Man unterscheidet zwischen verlustfreier Kompression(engl.: lossless compression) und verlustbehafteter Kompression (engl.: lossy compression).
Die Algorithmen, die eine perfekte Rekonstruktion der Originaldaten ohne Datenverlust aus den komprimierten Daten erlauben, sind **verlustfreie Kompressionsmethoden**. Diese Methoden schreiben die Daten effizienter und verkleinern dadurch die Größe einer Datei ohne Datenverlust. Beispiele hierfür sind das ZIP-Datei-Format, gzip bei GNU, Huffman-Codierung.
Die Algorithmen, bei denen die Rekonstruktion der Daten mit Approximation passiert und bei denen je nach Kompressionsgrad(engl.: Compression ratio) Datenverlust entsteht, sind **verlustbehaftete Kompressionsmethoden**. Sie reduzieren die Dateigröße stark und ermöglichen damit einen schnelleren Transport im Netz. JPEG und mp3 gehören zu diesen Methoden.

[2] Aufgenommen mit der beschafften USB-Kamera

Bild	Hex Ansicht
JPEG Bildformat Größe: 101KB [1] 	Address 0 1 2 3 4 5 6 7 8 9 a b c d e f Dump 00000000 ff d8 ff e0 00 10 4a 46 49 46 00 01 01 01 00 60 ÿØÿà..JFIF.....` 00000010 00 60 00 00 ff db 00 43 00 01 01 01 01 01 01 01 .`..ÿÛ.C........ 00000020 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .... 00000030 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .... 00000040 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .... 00000050 01 01 01 01 01 01 01 01 01 ff db 00 43 01 01 01 ....ÿÛ.C... 00000060 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .... 00000070 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .... 00000080 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .... 00000090 01 01 01 01 01 01 01 01 01 01 01 01 01 01 ff c0 ....ÿÀ 000000a0 00 11 08 01 e0 02 80 03 01 22 00 02 11 01 03 11 ....à.€.."...... 000000b0 01 ff c4 00 1f 00 00 01 05 01 01 01 01 01 01 00 .ÿÄ............. 000000c0 00 00 00 00 00 00 00 01 02 03 04 05 06 07 08 09 .... 000000d0 0a 0b ff c4 00 b5 10 00 02 01 03 03 02 04 03 05 ..ÿÄ.µ.......... 000000e0 05 04 04 00 00 01 7d 01 02 03 00 04 11 05 12 21 ......}........! 000000f0 31 41 06 13 51 61 07 22 71 14 32 81 91 a1 08 23 1A..Qa."q.2.‘¡.#
PNG Bildformat Größe: 477KB [2] 	Address 0 1 2 3 4 5 6 7 8 9 a b c d e f Dump 00000000 89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52 ‰PNG........IHDR 00000010 00 00 02 80 00 00 01 e0 08 06 00 00 00 35 d1 dc ...€...à.....5ÑÜ 00000020 e4 00 00 00 01 73 52 47 42 00 ae ce 1c e9 00 00 ä....sRGB.®Î.é.. 00000030 00 04 67 41 4d 41 00 00 b1 8f 0b fc 61 05 00 00 ..gAMA..±..üa... 00000040 00 09 70 48 59 73 00 00 0e c3 00 00 0e c3 01 c7 ..pHYs...Ã...Ã.Ç 00000050 6f a8 64 00 00 ff a5 49 44 41 54 78 5e 84 fd 67 o¨d..ÿ¥IDATx^„ýg 00000060 93 24 39 96 68 09 e6 3f 1f 59 91 99 79 55 95 99 “$9–h.æ?.Y‘™yU•™ 00000070 11 e1 9c 1a e7 9c 3b 67 e1 c1 39 27 c9 8a 74 bf .á œ.çœ;gáÁ9'ÉŠt¿ 00000080 9e d9 dd 3f 81 3d e7 9a c3 cb ab 5e 8f ec 07 88 žÙÝ?.=çšÃË«^.ì.ˆ 00000090 31 35 33 55 28 80 7b 2e c5 0f 85 ad 49 da 5d 1f 153U(€{.Å.…-IÚ]. 000000a0 a6 cd 07 dd b4 b3 36 48 a5 9d 59 6a 55 4e d3 a0 ¦Í.Ý´³6H¥.YjUNÓ. 000000b0 f5 30 8d bb 8f 53 ad 70 98 ea c5 a3 54 dd 3f 48 õ0.».S-p.êÅ£TÝ?H 000000c0 1e eb 71 eb f7 da 71 6c 71 7b 9a 6a fb 27 a9 55 .ëqë÷Úqlq{šjû'©U 000000d0 be 48 ed ca 65 6a 96 ce a3 f9 bc b2 7b 94 b6 57 ¾HíÊej–Î£ù¼²{”¶W 000000e0 46 d1 b6 1e 0c 53 71 6b 91 7a f5 eb 34 6c 3d 89 FÑ¶..Sqk‘zõë4l=‰ 000000f0 63 0a 9b f3 b4 b7 3e 4d d5 bd 53 9e 1f 70 dc 84 c.›ó´·>MÕ½Sž.pÜ„

[1] 103 424 Byte
[2] 489 049 Byte

Tabelle 2.1.: Hexadezimal-Darstellung der Bilddateien

2.1. PNG

Die ersten acht Byte sind die PNG-Signatur.
0x89 0x50 0x4e 0x47 0x0d 0x0a 0x1a 0x0a

Nach der PNG-Signatur folgt eine Reihe von Chunks[3]. Für das PNG-Format wurden 18 Chunk-Types definiert, vier davon sind „Critical Chunks“ und die restlichen 14 sind optionale Chunks. „Critical Chunks“ müssen in jeder PNG-Datei vorhanden sein und von jedem PNG-Decoder verstanden werden. Dagegen können Decoder die optionalen Chunks ignorieren, wenn sie diese nicht verstehen. Dies sind die vier „Critical Chunks“:

- IHDR(Image header): Erster Chunk in der Datei(Adresse 0x0c bis 0x0f). Entsprechend den vier Buchstaben IHDR 0x49 0x48 0x44 0x52

[3] Chunks sind Datenblöcke, die in manchen Multimedia-Formaten, wie z.B PNG oder MP3 benutzt werden

- PLTE(Palette): Definition der genutzten Farbpalette
- IDAT(Image data): Beginn der Bilddaten
- IEND(Image end): Letzter Chunk in der PNG-Datei

Chunk layout: Jeder Chunk besteht aus vier Feldern(Abb.2.1): Chunk data length, Chunk type, Chunk data und CRC. Im ersten Feld(4 Byte) wird die Größe der Chunk-Daten in Byte angegeben. Die Länge der Chunk-Daten kann auch 0 Byte sein, so dass ein Chunk praktisch auch nur aus drei Feldern bestehen kann(Abb. 2.2). Das zweite Feld(4 Byte) beinhaltet den Namen des Chunk. Im dritten Feld folgen die Chunk-Daten. Zum Abschluss folgt ein 4-Byte Cyclic redundancy check (CRC)[4].

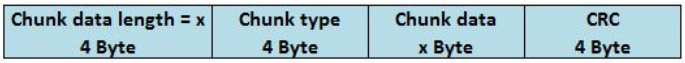


Abbildung 2.1.: Chunk layout mit vier Feldern

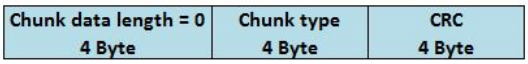
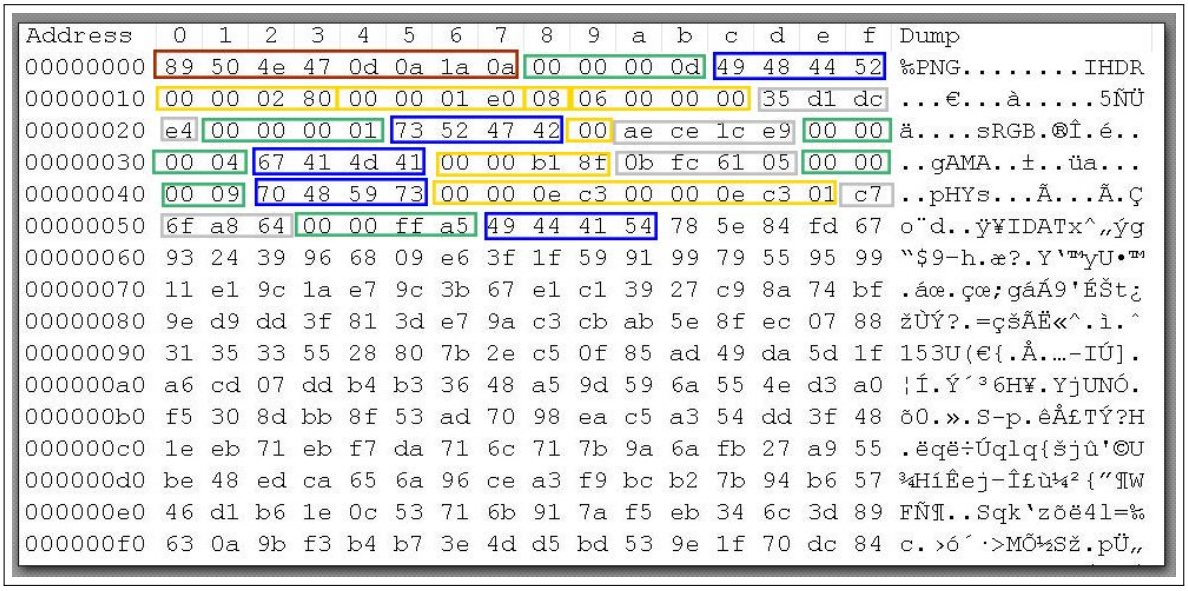


Abbildung 2.2.: Chunk layout mit drei Feldern

In Abb. 2.3 ist der Aufbau der PNG-Datei vergrößert dargestellt und die verschiedenen Chunks farbig markiert. In Tabelle 2.2 folgen die Erläuterungen.

[4]Der 4-Byte-CRC wird über Chunk-type und Chunk-data berechnet. Es wird das folgende Polynom genutzt:
$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$



Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	89	50	4e	47	0d	0a	1a	0a	00	00	00	0d	49	48	44	52	%PNG........IHDR
00000010	00	00	02	80	00	00	01	e0	08	06	00	00	00	35	d1	dc	...€...à.....5ÑÜ
00000020	e4	00	00	00	01	73	52	47	42	00	ae	ce	1c	e9	00	00	ä....sRGB.®Î.é..
00000030	00	04	67	41	4d	41	00	00	b1	8f	0b	fc	61	05	00	00	..gAMA..±..üa...
00000040	00	09	70	48	59	73	00	00	0e	c3	00	00	0e	c3	01	c7	..pHYs...Ã...Ã.Ç
00000050	6f	a8	64	00	00	ff	a5	49	44	41	54	78	5e	84	fd	67	o¨d..ÿ¥IDATx^„ýg
00000060	93	24	39	96	68	09	e6	3f	1f	59	91	99	79	55	95	99	"$9–h.æ?.Y'™yU•™
00000070	11	e1	9c	1a	e7	9c	3b	67	e1	c1	39	27	c9	8a	74	bf	.áœ.çœ;gáÁ9'ÉŠt¿
00000080	9e	d9	dd	3f	81	3d	e7	9a	c3	cb	ab	5e	8f	ec	07	88	žÙÝ?.=çšÃË«^.ì.ˆ
00000090	31	35	33	55	28	80	7b	2e	c5	0f	85	ad	49	da	5d	1f	153U(€{.Å.…-IÚ].
000000a0	a6	cd	07	dd	b4	b3	36	48	a5	9d	59	6a	55	4e	d3	a0	¦Í.Ý´³6H¥.YjUNÓ.
000000b0	f5	30	8d	bb	8f	53	ad	70	98	ea	c5	a3	54	dd	3f	48	õ0.».S-p.êÅ£TÝ?H
000000c0	1e	eb	71	eb	f7	da	71	6c	71	7b	9a	6a	fb	27	a9	55	.ëqë÷Úqlq{šjû'©U
000000d0	be	48	ed	ca	65	6a	96	ce	a3	f9	bc	b2	7b	94	b6	57	¾HíÊej–Î£ù¼²{"¶W
000000e0	46	d1	b6	1e	0c	53	71	6b	91	7a	f5	eb	34	6c	3d	89	FÑ¶..Sqk'zõë4l=‰
000000f0	63	0a	9b	f3	b4	b7	3e	4d	d5	bd	53	9e	1f	70	dc	84	c.›ó´·>MÕ½Sž.pÜ„

Abbildung 2.3.: PNG-Datei in Hexadezimal-Ansicht(Quelle: Testbild1.png)
Rot: PNG-Signatur, Grün: Chunk data length, Blau: Chunk type, Gelb:Chunk data, Grau:CRC

Adresse	Wert	Bedeutung
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07	0x89 0x50 0x4e 0x47 0x0d 0x0a 0x1a 0x0a	PNG-Signatur
0x08 0x09 0x0a 0x0b	0x00 0x00 0x00 0x0d	Beginn des IHDR-Chunks, Chunk data lenght = 0x0d = 13 Byte
0x0c 0x0d 0x0e 0x0f	0x49 0x48 0x44 0x52	Chunk type = „IHDR“: Image-Header-Chunk
0x10 0x11 0x12 0x13	0x00 0x00 0x02 0x80	Chunk data(4 Byte): Breite des Bildes, hier 0x0280 = 640 Pixel
0x14 0x15 0x16 0x17	0x00 0x00 0x01 0xe0	Chunk data(4 Byte): Höhe des Bildes, hier 0x01e0 = 480 Pixel
0x18	0x08	Chunk data(1 Byte): Number of Bits per sample, hier 0x08 = 8
0x19 0x1a 0x1b 0x1c	0x06 0x00 0x00 0x00	Chunk data(4 Byte): Color type, Compression method, Filter method, Interlace method
0x1d 0x1e 0x1f 0x20	0x35 0xd1 0xdc 0xe4	CRC des IHDR-Chunk
0x21 0x22 0x23 0x24	0x00 0x00 0x00 0x01	Beginn des PLTE-Chunks: Chunk data length = 1 Byte

Adresse	Wert	Bedeutung
0x25 0x26 0x27 0x28	0x73 0x52 0x47 0x42	Chunk type = „sRGB“: standard RGB color space
0x29 0x2a 0x2b 0x2c 0x2d	0x00 0xae 0xce 0x1c 0xe9	Chunk data(1Byte) und CRC(4 Byte)
0x2e 0x2f 0x30 0x31	0x00 0x00 0x00 0x04	Beginn des 3. Chunks: Chunk data length = 4 Byte
0x32 0x33 0x34 0x35	0x67 0x41 0x4d 0x41	Chunk type = „gAMA“: Image gamma
0x36 - 0x3d		Daten und CRC des „gAMA“-Chunks
0x3e - 0x52		4. Chunk: „pHYs“-Chunk, Physical pixel dimensions
0x53 0x54 0x55 0x56	0x00 0x00 0xff 0xa5	Beginn des IDAT-Chunks: Chunk data length = 0xffa5 = 65.445 Byte
0x57 0x58 0x59 0x5a	0x49 0x44 0x41 0x54	Chunk type = „IDAT“: Ab hier folgen 65.445 Byte Bilddaten

Tabelle 2.2.: PNG Datei-Format-Analyse

2.2. JPEG

Ein erster Blick in das JPEG-Format zeigt, dass JPEG-Dateien aus verschiedenen Teilen zusammengesetzt sind. Jeder Teil wird mit einem sogenannten „Marker“ gekennzeichnet. In diesem Kapitel wird ein Bild, das mit der System-Kamera aufgenommen wurde, angeschaut.

Marker
Alle Marker in JPEG-Dateien haben eine Größe von zwei Byte. Das erste Byte ist immer „0xff“ und das zweite Byte ist der eigentliche „Identifier“.

Anmerkung
Ein Byte mit dem Wert 0xff wird im JPEG-Format extra behandelt. Entweder ist dieses Byte ein Teil von einem Marker und es wird der Identifier des Markers erwartet. Oder es gehört zu den Bild-Daten[5] und in diesem Fall wird ein Null-Byte 0x00 hinzugefügt. Dieses Verfahren nennt sich „**Byte Stuffing**“.

In Tabelle 2.3 sind die enthaltenen Marker in dem untersuchten Test-Bild (Abb. 2.4, 2.5), also die Marker, die für diese Arbeit relevant sind, aufgelistet.

Abbildung 2.4.: Test-Bild.jpg /640x480/ 50% Qualität

[5] Codierter Pixel-Wert

Name	Identifeir	Beschreibung
SOI	0xff 0xd8	Start of Image
APP0	0xff 0xe0	JFIF application segments
DQT	0xff 0xdb	Define Quantization Table
SOF0	0xff 0xc0	Start of Frame
DHT	0xff 0xc4	Define Huffman Table
SOS	0xff 0xda	Start of Scan
EOI	0xff 0xd9	End of Image

Tabelle 2.3.: Marker im JPEG-Format

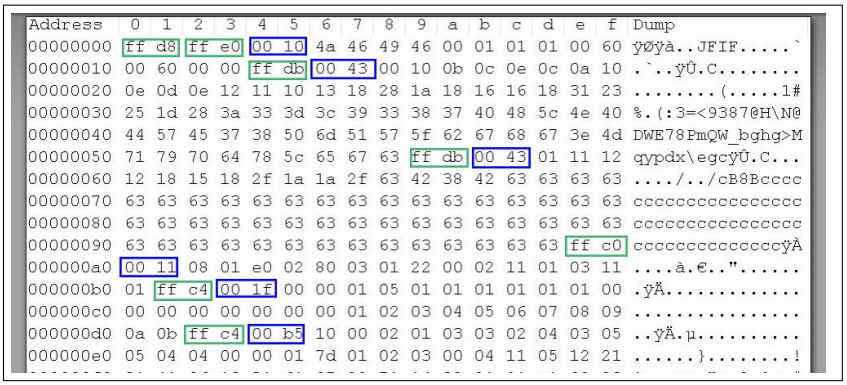


Abbildung 2.5.: Hex-Ansicht von Abb. 2.4.Die grün markierten Kästchen sind die Marker, die blauen Kästchen sind die Größen des jeweiligen Abschnitts

3. JPEG Kompression

In diesem Kapitel wird das JPEG-Format im Detail untersucht. Um den Aufbau des Datei-Formats zu verstehen und später manipulieren zu können, werden die Test-Bilder[1] Byte für Byte analysiert.

Datenkompression bedeutet, die Menge digitaler Daten zu verdichten oder zu reduzieren. Es gibt zwei Arten der Kompression:

- Verlustfreie Kompression[2] z. B. PNG, Huffman-Kodierung
  - Es wird keine Information verändert oder entfernt
  - Es wird die Redundanz der Daten entfernt
  - Der Decoder kann aus den komprimierten Daten, die Original-Daten ohne jegliche Verluste rekonstruieren.
- Verlustbehaftete Kompression[3] z. B. JPEG, mp3
  - Information gehen bei der Kompression verloren
  - Es werden irrelevante Informationen, die z. B. vom menschlichen Auge bzw. Ohr nicht wahrgenommen werden, entfernt
  - Die Original-Daten können nicht komplett aus den komprimierten Daten rekonstruiert werden.

Die **verlustbehaftete** Kompression orientiert sich an der Wahrnehmungsfähigkeit des Betrachters bzw. des Zuhörers. So können, aufgrund der menschlichen Gehirnfähigkeit und der Auflösungsmöglichkeit von Farben, bei Bildern Informationen verändert oder entfernt werden, ohne dass sich die Wahrnehmung wesentlich ändert (mehr dazu in Kapitel 3.1).

[1] Die Test-Bilder wurden mit der für diese Arbeit beschafften USB-Kamera gemacht.
[2] Engl.: lossless data compression
[3] Engl.: lossy data compression

Daher ermöglicht die verlustbehaftete Kompression wesentlich höhere Kompressionsraten[4] als die verlustfreie Kompression.

Das JPEG-Kompressions-Schema ist in die folgenden Stufen aufgeteilt(Abb.3.1), die in den nächsten Kapiteln genauer beschrieben werden:

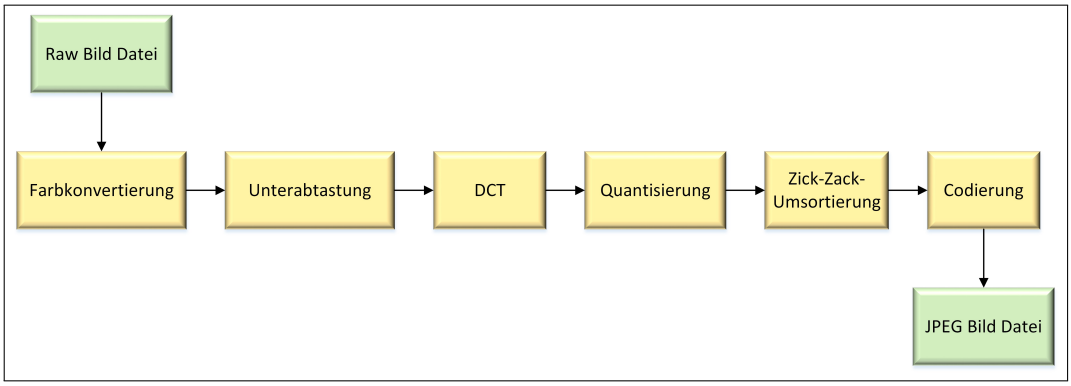


Abbildung 3.1.: JPEG Kompressions-Schema

1. Farbkonvertierung (Kapitel 3.1) → RGB zu YCbCr (Farb- und Helligkeitsinformation getrennt)
2. Subsampling (Kapitel 3.2) → Unterabtastung der Chrominanz (Farb-Komponenten Cb und Cr)
3. Transformation (Kapitel 3.3) → Discrete Cosine Transform (DCT) der 8x8-Pixel-Blöcke, Pixel-Werte werden vom Raumbereich in den Frequenzbereich transformiert
4. Quantisierung (Kapitel 3.4) → Die 8x8-DCT-Koeffizienten-Blöcke werden mit verschiedenen Quantisierungs-Tabellen quantisiert
5. Zick-Zack-Umsortierung (Kapitel 3.5) → Die 8x8-Blöcke werden nach einem Zick-Zack-Muster ausgelesen und in ein Array der Größe 64 gespeichert.
6. Kodierung (Kapitel 3.6) → Durch die Umsortierung entstehen große Gruppen von 0-Werten, so dass eine effizientere Darstellung der 64 Werte erfolgen kann. Durch Huffman-Codierung werden zusätzlich Redundanzen[5] entfernt.

In Kapitel 3.7 wird ein Beispiel-Bild untersucht, um herauszufinden, wie die Daten nach der Kompression gespeichert werden.

[4] Kompressionsrate, auch als Kompressionsfaktor bezeichnet, gibt das Verhältnis der komprimierten Dateigröße zur Original-Dateigröße an z. B. bedeutet ein Kompressionsfaktor von 1:10, dass die Dateigröße nach der Kompression ein Zehntel der Original-Dateigröße beträgt.

[5] Eine Informationseinheit ist redundant, wenn sie ohne Informationsverlust weggelassen werden kann[**?**]

3.1. Farbkonvertierung

Das RGB-Modell baut auf den drei Grundfarben Rot, Grün und Blau auf, aus denen sich jede andere Farbe mischen lässt. Wenn z. B. alle drei Farben zu gleichen Teilen gemischt werden, ergibt sich immer ein Grauton. Auch bei Schwarz(0,0,0) und Weiß(255,255,255) haben alle drei Farben den gleichen Anteil.
Ein Pixel beinhalt für jede der 3 Farben, auch 3 Kanäle gennant, einen Farbwert[6]. Der Wert für jeden Kanal wird in einem Byte gespeichert. Das bedeutet, dass jede Farbe $2^8 = 256$ [0 ... 255] verschiedene Intensitätsstufen annehmen kann. Dies wird auch 8-Bit Farbtiefe genannt[7]. Es ergeben sich daraus $(2^8)^3 = 16.777.216$ Kombinationsmöglichkeiten.

Das YCbCr-Modell unterteilt in Helligkeit und Farbigkeit:

Y = Luminanz[8] (Helligkeit)
CbCr = Chrominanz[9] (Farbigkeit), Cb = Blau-Gelb-Chrominanz, Cr = Rot-Grün-Chrominanz

Diese Farbkonvertierung von RGB in YCbCr verschafft der JPEG-Kompression große Vorteile. Während das menschliche Auge sehr empfindlich für Helligkeitsunterschiede ist, kann es kleinere Farbunterschiede nicht so gut wahrnehmen. Durch die Aufteilung in Luminanz und Chrominanz kann bei den Chrominanzwerten Cb und Cr durch Unterabtastung viel Platz eingespart werden, während die Luminanzwerte Y ohne Verluste erhalten bleiben.(Siehe Kapitel 3.2)
Die folgenden Formeln beschreiben die Umrechnung zwischen den Modellen:
RGB $\Rightarrow$ YCbCr

$$Y = 0,299 \cdot R + 0,587 \cdot G + 0,114 \cdot B \tag{3.1}$$

$$Cb = \frac{B - Y}{2 - 2 \cdot 0,114} \tag{3.2}$$

$$Cr = \frac{R - Y}{2 - 2 \cdot 0,299} \tag{3.3}$$

YCbCr $\Rightarrow$ RGB

$$R = Cr \cdot (2 - 2 \cdot 0,299) + Y \tag{3.4}$$

[6] Hier wird eine vereinfachte Betrachtung der Pixel-Speicherung vorgenommen, ohne weitere Kanäle, wie z. B. den α-Kanal. Dies entspricht der Art der Speicherung der Test-Bilder mit der benutzten Kamera
[7] Es gibt auch digitale Bilder, die höhere Farbtiefen benutzen
[8] Engl.: Luminance
[9] Engl.: Chrominance

$$G = \frac{Y - 0,114 \cdot B - 0.299 \cdot R}{0,587} \tag{3.5}$$

$$B = Cb \cdot (2 - 2 \cdot 0,114) + Y \tag{3.6}$$

Nach der Farbkonvertierung wird eine sogenannte **Indexverschiebung** vorgenommen. Die Werte aus dem Wertebereich [0 . . . 255] (8 Bit unsigned) werden in den Wertebereich[-128 . . . 127] (8 Bit signed) verschoben. Dies ist vorteilhaft für die in Kapitel 3.3 beschriebene diskrete Kosinus Transformation, da es sich bei der Kosinus-Funktion um eine symmetrische Funktion handelt.

3.2. Subsampling

Die Unterabtastung basiert auf der menschlichen visuellen Wahrnehmung von Licht und Farbe. Das menschliche Gehirn ist sehr empfindlich für Helligkeit, kann aber geringe Unterschiede zwischen verschiedenen Farb- und Grautönen nicht erkennen.[10]
Die JPEG-Kompression nutzt diese Eigenschaft und verringert die Farbinformationen des Bildes durch Unterabtastung. Zuerst wird das Bild in quadratische Blöcke geteilt. Jeder Block ist 8x8 Pixel groß(falls nötig, werden Pixel zu den Reihen oder Spalten hinzugefügt).
So wird ein Bild in JPEG als Anzahl von Matrizen der Größe 8x8 gespeichert. Alle weiteren Operationen werden auf diesen Matrizen durchgeführt.

Eine **Minimum coded unit (MCU)** repräsentiert einen Pixel-Bereich mit einer bestimmten Anzahl an Pixeln in horizontale und vertikale Richtung.
Die typischen MCU-Größen sind: 8x8, 16x8 oder 16x16 Pixel.
So ergibt z. B. ein Bild mit 640x480 Pixeln:

- Bei MCU-Größe 8x8 Pixel:
  - $640 \div 8 = 80 \Rightarrow$ Anzahl MCU horizontal = 80
  - $480 \div 8 = 60 \Rightarrow$ Anzahl MCU vertikal = 60
  - Das komplette Bild besteht aus 4800 MCU.
- Bei MCU-Größe 16x8 Pixel:
  - $640 \div 16 = 40 \Rightarrow$ Anzahl MCU horizontal = 40

[10] Das menschliche Auge kann etwas weniger als 100 Grautöne unterscheiden[**?**], das heißt eine 8 Bit Grau-Abstufung, mit 256 verschiedenen Grautönen kann gar nicht komplett wahrgenommen werden.

  - $480 \div 8 = 60 \Rightarrow$ Anzahl MCU vertikal = 60
  - Das komplette Bild besteht aus 2400 MCU.
- Bei MCU-Größe 16x16 Pixel:
  - $640 \div 16 = 40 \Rightarrow$ Anzahl MCU horizontal = 40
  - $480 \div 16 = 30 \Rightarrow$ Anzahl MCU vertikal = 30
  - Das komplette Bild besteht aus 1200 MCU.

Wie bereits erwähnt, wird das Subsampling nur bei der Chrominanz eingesetzt, da der Informationsverlust bei der Farbe einen für das menschliche Auge nicht bemerkbaren Qualitätsverlust verursacht. So werden zum Beispiel für 4 Pixel 4 Helligkeitswerte gespeichert, aber, je nach Methode, nur 1 oder 2 Farbwerte. Hierzu wird jeweils der Durchschnitt der zusammengefassten Werte gebildet. Dadurch kann schon eine Kompression von bis zu 50% erreicht werden.
Die vier verschiedenen Subsampling-Methoden werden in Tabelle 3.1 und Abbildung 3.2 beschrieben:

Subsampling	1x1	2x1	1x2	2x2
Bezeichnung	4:4:4	4:2:2	4:2:2	4:2:0
Bedeutung	Kein Subsampling	Horizantales Subsampling	Vertikales Subsampling[2]	Horizontales & vertikales Subsampling
Pixel pro MCU	8x8 = 64	16x8 = 128	8x16 = 128	16x16 = 256
MCU-Komponenten **ohne** Subsampling	Y, Cb, Cr	Y00, Y10, Cb00, Cb10, Cr00, Cr10	Y00, Y01, Cb00, Cb01, Cr00, Cr01	Y00, Y10, Y01, Y11, Cb00, Cb10, Cb01, Cb11, Cr00, Cr10, Cr01, Cr11
Anzahl MCU-Komponenten	**3**	**6**	**6**	**12**
Benötigte Bit[1] pro MCU **ohne** Subsampling	**3** x 64 x 8 = 1536	**6** x 64 x 8 = 3072	**6** x 64 x 8 = 3072	**12** x 64 x 8 = 6144
MCU-Komponenten **mit** Subsampling	Y, Cb, Cr	Y00, Y10, Cb, Cr	Y00, Y01, Cb, Cr	Y00, Y10, Y01, Y11, Cb, Cr
Anzahl MCU-Komponenten	**3**	**4**	**4**	**6**
Benötigte Bit[1] pro MCU **mit** Subsampling	**3** x 64 x 8 = 1536	**4** x 64 x 8 = 2048	**4** x 64 x 8 = 2048	**6** x 64 x 8 = 3072
Einsparung pro MCU [Bit]	0	1024 (33,33%)	1024 (33,33%)	3072 (50%)

[1] 8x8-Matrix = 64 Elemente, 8 Bit Farbtiefe

[2] für hochformatige Bilder

Tabelle 3.1.: Verschiedene Chrominanz-Subsampling-Methoden

Abbildung 3.2.: Verschiedene Chrominanz-Subsampling-Methoden

A:B:C-Notation
A := Breite eines zweireihigen Pixelblocks,
B:= Abtastrate von Cb und Cr in Relation zu A in der oberen Pixelreihe,
C:= Abtastrate von Cb und Cr in Relation zu A in der unteren Pixelreihe

3.3. DCT - Diskrete Kosinustransformation

In dieser Stufe liegen die Bildinformationen in 8x8-Blöcken von Luminanz und Chrominanz vor. Auf jeden dieser Blöcke wird jetzt die zweidimensionale DCT angewendet. Die diskrete Kosinustransformation wandelt die diskrete Ort-Werte Zuordnung in eine diskrete Frequenz-Amplituden Zuordnung um.
Hier werden die 64 Werte in jedem Block als ein diskretes Signal mit 64 Werten betrachtet, wobei jedes Signal eine zweidimensionale x-y-Raumfunktion ist. Diese Funktion wird jetzt durch überlagerte Kosinusfunktionen mit unterschiedlichen Amplituden und Frequenzen nachgebildet. Die Amplitudenwerte werden in die neue 8x8-Matrix eingetragen. Dabei wird oben links, also an Position (0,0), die Amplitude der Kosinusfunktion mit der geringsten Frequenz eingetragen. Dieser wird auch DC-Koeffizient(Gleichanteil) genannt. Da die DC-Koeffizienten benachbarter Blöcke meist in einem ähnlichen Bereich liegen, werden die DC-Koeffizienten jeweils als Differenz zum DC-Koeffizienten des vorherigen Blocks gespeichert.
Die Frequenzen in der 8x8-Matrix steigen von links nach rechts und von oben nach unten an, so dass unten rechts die Amplitude der Funktion mit der höchsten Frequenz steht. Die 63 Koeffizienten mit höheren Frequenzen werden AC-Koeffizienten(Wechselanteile) genannt.
Da die Anteile mit hohen Frequenzen in den meisten Bildern geringer sind, ergeben sich in der neuen 8x8-Matrix in der Regel oben links die höchsten Werte, während die Werte nach unten rechts immer niedriger werden und gegen 0 gehen.

Anmerkung
Hier ist zu beachten, das die DCT verlustfrei ist. Das heißt, dass aus den Koeffizienten die ursprünglichen Pixelwerte mit der inversen DCT(IDCT) wieder hergestellt werden können. In diesem Schritt findet keine Kompression statt, sondern die Daten werden für die nächsten Schritte vorbereitet, um mit der Quantisierung und Kodierung eine möglichst hohe Kompression zu erzielen.
In der Gleichung 3.7 ist die DCT, auch FDCT(Forward DCT) genannt, dargestellt. In der Gleichung 3.8 folgt die IDCT(Inverse DCT), die im JPEG-Decoder benutzt wird.

Zweidimensionale FDCT:

$$F(u, v) = \frac{1}{4}C(u)C(v)[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x, y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2x+1)v\pi}{16}] \tag{3.7}$$

Zweidimensionale IDCT:

$$f(x, y) = \frac{1}{4}[\sum_{u=0}^{7}\sum_{v=0}^{7} f(x, y)C(u)C(v)F(u, v)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2x+1)v\pi}{16}] \tag{3.8}$$

mit:
$C(u), C(v) = \frac{1}{\sqrt{2}}$ wenn $u, v = 0$
$C(u), C(v) = 1$ wenn $u, v \neq 0$

3.4. Quantisierung

Nach der Transformation jedes einzelnen 8x8-Blocks folgt nun die Quantisierung jedes einzelnen Koeffizienten mit unterschiedlichen Werten.
Es wird für jeden einzelnen Wert (u, v) in der Matrix ein **Quantisierungsfaktor** $Q(u, v)$ definiert. Nach der Gleichung 3.9 werden die neuen Werte berechnet.

$$\widetilde{F}(u, v) = IntegerRound\frac{F(u, v)}{Q(u, v)} \tag{3.9}$$

Das menschliche Auge ist nicht sehr empfindlich für hochfrequente Schwingungen in Bildern und so werden für höhere Frequenzen auch höhere Quantisierungs-Faktoren genutzt. So steigen die Quantisierungs-Faktoren von links nach rechts und von oben nach unten an.
Durch die Integer-Rundung nach der Quantisierung entstehen die meisten Verluste. Hierbei ist zu beachten, dass die Ungenauigkeit bei der Wiederherstellung der ursprünglichen Werte umso größer wird, je größer der Quantisierungs-Faktor ist, da die Rundungsfehler bei der Dequantisierung im Decoder ebenfalls größer werden.
Daher wird die Qualität eines JPEG-Bildes durch die genutzten Quantisierungs-Tabellen definiert. So wird z. B. für ein JPEG-Bild mit 100% Qualität ein Quantisierungs-Faktor von eins für alle Werte genutzt. Hier ist zu beachten, dass trotz der eins als Quantisierungs-Faktor Informationen durch die Rundung der DCT-Koeffizienten verloren gehen. Also ist selbst die JPEG-Kompression bei einer Qualität von 100% verlustbehaftet.
Wie bereits erwähnt, ändern sich die räumlichen Frequenzen in einem natürlichen Bild[11] nur in kleinen Schritten und somit in einem 8x8-Block kaum. Daher werden die DCT-Koeffizienten tendenziell mit größeren Frequenzen immer kleinere Werte annehmen. So enthalten die 8x8-Blöcke nach der Quantisierung und Rundung sehr viele Nullwerte. Diese Eigenschaft wird in den nächsten Schritten (Kapitel 3.5 und 3.6) genutzt und verschaffen einen großen Vorteil bei der Kodierung.

In den Tabellen 3.2 und 3.3 sind die Standard-Quantisierungs-Tabellen[12] für Luminanz bzw. Chrominanz dargestellt.
In Abbildung 3.3 ist ein Beispiel für den Informationsverlust, der durch die Quantisierung mit anschließender Rundung entsteht, dargestellt.

[11] Das nicht mit einem Computer erstellt wurde, sondern mit einer Kamera und entsprechende Merkmale, wie z. B. Rauschen, enthält
[12] Diese Tabellen werden von der genutzten System-Kamera bei 50%-Qualität genutzt.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Tabelle 3.2.: Standard-Quantisierungs-Tabelle für Luminanz

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Tabelle 3.3.: Standard-Quantisierungs-Tabelle für Chrominanz

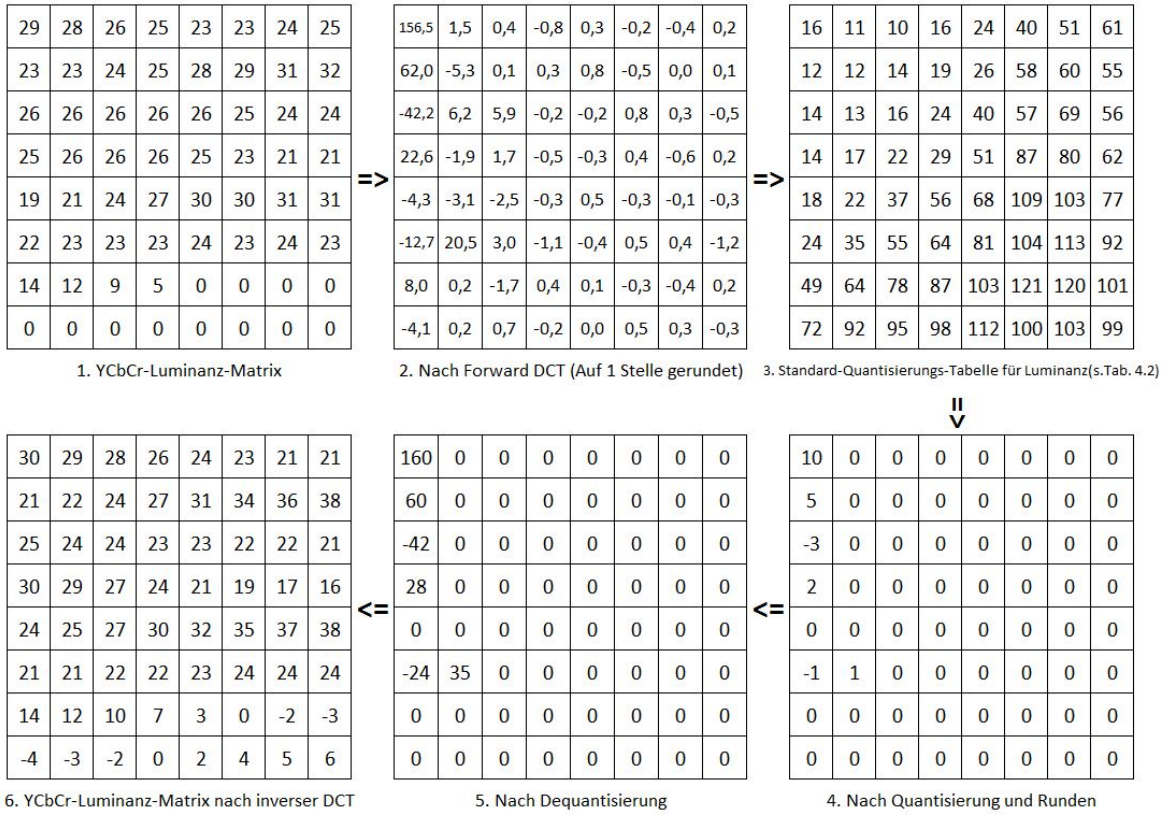


Abbildung 3.3.: Beispiel für die Informationsverluste durch die Quantisierung und Rundung

Dabei sind in der 1. Matrix oben links die Originalwerte im YCbCr-Format dargestellt. Im ersten Schritt erfolgt die diskrete Kosinus-Transformation(DCT). Die Werte nach der DCT sind in der zweiten Matrix, auf eine Nachkommastelle gerundet, dargestellt. Diese Werte werden dann durch die Werte in der Quantisierungs-Tabelle(3.) geteilt und auf einen vollen Integer-Wert gerundet(4.). Hier ist gut zu erkennen, dass sehr viele Nullen entstanden sind und diese Matrix sehr viel platzsparender gespeichert werden kann.
Bei der Decodierung der JPEG-Bilder werden diese Werte wieder mit den Werten aus der Quantisierungs-Tabelle(3.) multipliziert(5.) und darauf die inverse diskrete Kosinus-Transformation ausgeführt.
Die Werte der rekonstruierten Matrix unten links unterscheiden sich nur relativ gering von den Originalwerten oben links.

3.5. Zick-Zack-Umsortierung

Durch die Quantisierung und Rundung der DCT-Koeffizienten entstehen viele Nullen in den Blöcken. Um das bei der Kodierung auszunutzen, müssen so viele Nullen wie möglich nacheinander stehen. So können diese jeweils in einer Gruppe zusammengefasst werden.
Um dies zu erreichen, werden die Koeffizienten nach einem Zick-Zack-Muster(Abb. 3.4) abgelesen und so umsortiert in ein Array mit 64 Feldern geschrieben(Abb. 3.5). Bei diesem Muster werden die Werte in der oberen linken Ecke, wo am häufigsten Werte ungleich Null vorhanden sind, zuerst abgelesen und es entstehen zum Ende des Arrays sehr große Gruppen von Nullen.

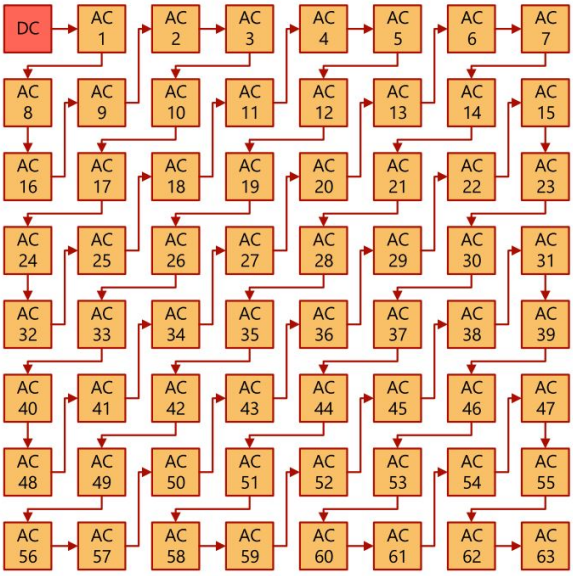


Abbildung 3.4.: Zick-Zack-Ablesung der Koeffizienten

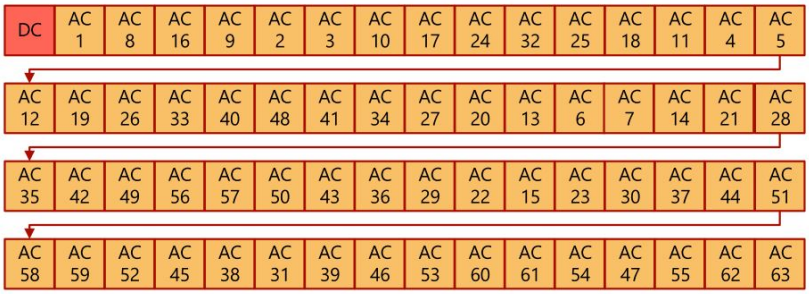


Abbildung 3.5.: Umsortiertes Koeffizienten-Array

3.6. Koeffizienten-Kodierung

Grundsätzlich werden die Koeffizienten mit Variable Length Integer (VLI) codiert. Dabei werden kleine Zahlen durch kurze Bitfolgen und größere Zahlen durch längere Bitfolgen repräsentiert.
Da in den 8x8-Blöcken jetzt überwiegend kleinere Zahlen vorkommen, kann dadurch erneut einiges an Speicherplatz gespart werden. Dafür muss aber für jeden Koeffizienten die Anzahl der für die Darstellung benötigten Bits angegeben werden.
Da die Werte der DC-Koeffizienten meist wesentlich größer sind, als die der AC-Koeffizienten, werden diese gesondert codiert.
Die Tabellen 3.4 und 3.5 zeigen die Kodierung der VLI. Dabei ist das hochwertigste Bit bei negativen Zahlen immer 0 und bei positiven Zahlen immer 1.
Für die DC-Koeffizienten stehen maximal 11 Bit zur Verfügung, so dass Zahlen zwischen -2047 und 2047 dargestellt werden können. Für die AC-Koeffizienten stehen nur 10 Bit zur Verfügung, also Zahlen zwischen -1023 und 1023.

Bit-Anzahl	DC-Differenz-Werte	Bitfolgen
0	0	
1	-1 , 1	0, 1
2	-3, -2 , 2, 3	00, 01, 10, 11
3	-7. . . -4, 4. . . 7	000, ..., 011, 100, ..., 111
4	-15. . . -8, 8. . . 15	0000, ..., 0111, 1000, ..., 1111
5	-31. . . -16, 16. . . 31	...
6	-63. . . -32, 32. . . 63	...
7	-127. . . -64, 64. . . 127	...
8	-255. . . -128, 128. . . 255	...
9	-511. . . -256, 256. . . 511	...
A	-1023. . . -512, 512. . . 1023	...
B	-2047. . . -1024, 1024. . . 2047	...

Tabelle 3.4.: Darstellung der DC-Koeffizienten-Differenz-Werte

Bit-Anzahl	AC Koeffizient	Bitfolgen
1	-1,1	0, 1
2	-3, -2, 2, 3	00, 01, 10, 11
3	-7...-4, 4...7	000, ..., 011, 100, ..., 111
4	-15...-8, 8...15	0000, ..., 0111, 1000, ..., 1111
5	-31...-16, 16...31	...
6	-63...-32, 32...63	...
7	-127...-64, 64...127	...
8	-255...-128, 128...255	...
9	-511...-256, 256...511	...
A	-1023...-512, 512...1023	...

Tabelle 3.5.: Darstellung der AC-Koeffizienten-Werte

DC-Koeffizienten Kodierung

Da die DC-Koeffizienten meist recht groß sind, aber für benachbarte Blöcke meist in einem ähnlichen Bereich liegen, werden die DC-Koeffizienten jeweils als Differenz zum DC-Koeffizienten des vorherigen Blocks gespeichert. In Gleichung 3.10 wird für den ersten Block Null als vorheriger DC-Wert eingesetzt.

$$DIFF_i = DC_i - DC_{i-1} \tag{3.10}$$

Es wird jetzt eine Zwischendarstellung aus zwei Symbolen genutzt: Das zweite Symbol ist der DC-Koeffizienten-Differenzwert in VLI-Darstellung und das erste Symbol die Anzahl der dafür benötigten Bits.
Wenn der Wert eines DC-Koeffizienten z. B. 10 beträgt, ergibt sich folgende Zwischendarstellung: $(4)(10)$, 4 Bit werden für die Darstellung der 10 benötigt.
Während das zweite Symbol (10) durch VLI mit 1010 codiert wird, wird für das erste Symbol ein Variable Lenght Code (VLC) genutzt. Bei JPEG wird hierfür die Huffman-Codierung eingesetzt, die in Kapitel 5 genauer beschrieben wird.

AC-Koeffizienten Kodierung

Die AC-Koeffizienten enthalten viele Nullen. Daher besteht das erste Symbol der Zwischendarstellung hier aus zwei Teilen: In den ersten 4 Bit wird die Run length Zero (RLZ) angegeben. Die RLZ sagt aus, wie viele ununterbrochene Nullen seit dem letzten Wert ungleich Null aufgetreten sind(4 Bit = Maximal 15). In den zweiten 4 Bit wird wieder die

Anzahl der benötigten Bits für den Wert in Symbol 2 gespeichert. In Abbildung 3.6 ist ein Beispiel einer 8x8-Matrix nach der Zick-Zack-Umsortierung.

10	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
-3	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

=>

DC	AC1	AC2	AC3	AC4	AC5	AC6	AC7	AC8	AC9	AC10-19	AC20	AC21	AC22	AC23-63
10	0	5	-3	0	0	0	0	0	2	10x 0	-1	0	1	41x 0

Abbildung 3.6.: 8x8-Matrix

Die Zwischendarstellung der Matrix lautet jetzt:

DC: $[(4)(10)]$ AC: $[(1,3)(5)]$, $[(0,2)(-3)]$, $[(5,2)(2)]$, $[(10,1)(-1)]$, $[(1,1)(1)]$, $[(0,0)]$

Es gibt zwei Sonderfälle für die Zwischendarstellung:

- Bis zum letzten Koeffizienten treten nur noch Nullen auf: Symbol 1 = (0,0), Symbol 2 entfällt. Dies wird auch „End-of-Block“ oder „EOB“ genannt.
- Mehr als 15 Nullen hintereinander: Symbol 1 = (15,0), Symbol 2 entfällt. Dies repräsentiert 16 Nullen.

Das erste Symbol, hier die Kombinationen aus RLZ und Bit-Anzahl, wird Huffman-Codiert. Für das zweite Symbol wird die VLI-Codierung genutzt.
Insgesamt gibt es vier Huffman-Tabellen für die Codierung des ersten Symbols: DC-Luminanz, AC-Luminanz, DC-Chrominanz, AC-Chrominanz

Im folgenden Kapitel wird ein Beispiel-Bild komplett analysiert und das Verfahren dabei genauer erklärt.

3.7. Untersuchung eines Beispielbildes

Um die Software entwickeln zu können, wird in diesem Teil ein Beispiel-Bild(Abb. 3.7) untersucht und die Koeffizienten, in denen dann die Nachricht versteckt werden soll, lokalisiert und dekodiert.

Abbildung 3.7.: Das Beispielbild mit 50% Qualität

Hierzu sind viele kleine Schritte notwendig, um alle benötigten Informationen aus dem codierten Bild zu bekommen. Die folgenden Informationen werden benötigt:

- Größe des Headers? Größe der Bilddaten[13]?
- Bildformat(Breite x Höhe)?
- Welches Subsampling? Wie viele MCU? Wie viele Koeffizienten?
- Welche Quantisierungs-Tabellen?
- Welche Tabellen für die Huffman-Decodierung?

Als erstes muss das Bildformat in seine verschiedenen Teile zerlegt werden. Dazu müssen die Marker identifiziert und ihre Größe und Informationen interpretiert werden. Damit das gelingt, wird in diesem Beispiel alles manuell untersucht und im nächsten Schritt durch die Software automatisiert.

[13] Hier sind die Daten gemeint, die Bildinformationen(Pixelwerte) enthalten, nicht die JPEG-Kompressionsinformationen

DQT-Marker: 0xff 0xdb

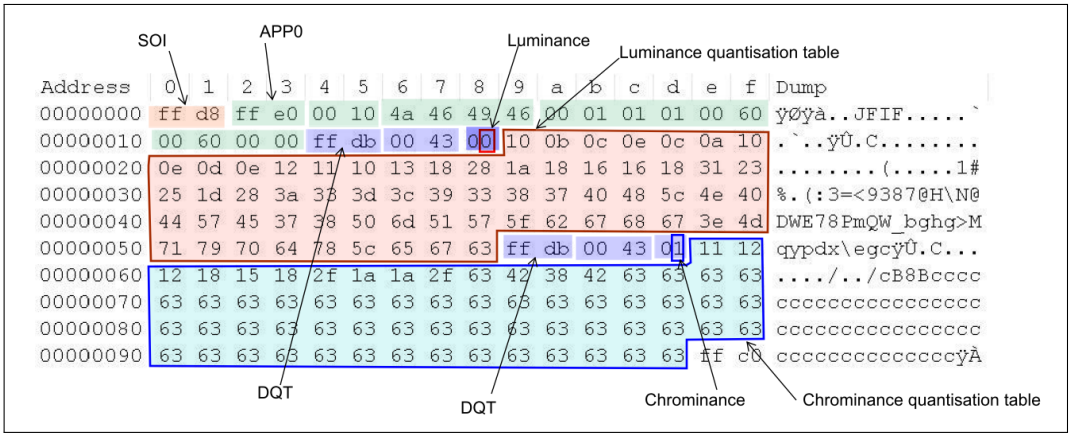


Abbildung 3.8.: DQT-Marker-Abschnitt(Define Quantization Table)

Das JPEG-Bild startet mit dem SOI-Marker 0xffd8 („Start of Image“). Direkt im Anschluss folgt der APP0-Marker, der Informationen zur JPEG-Version und zur Dichte der Pixel[14] beinhaltet. Ab Adresse 0x14 folgt der DQT-Abschnitt. Aufbau des „DQT“-Abschnitts:

- Zwei Byte für den Marker
- Zwei Byte für die Größe der Daten(inkl. dieser zwei Byte)
- Ein Byte für die ID der Quantisierungs-Tabelle
- Die 64 Werte der Quantisierungs-Tabelle

Aus der Abb. 3.8 werden die Werte in Tabelle 3.6 abgelesen:

Adresse	Wert	Beschreibung
0x14 0x15	0xff 0xdb	DQT-Marker
0x16 0x17	0x00 0x43	Größe der Daten. Hier 67 Byte: 2 Byte für die Größenangabe, 1 Byte für die ID, 64 Byte für die Tabelle
0x18	0x00	ID der Quantisierungs-Tabelle. 00 für Luminanz
0x19-0x58	64 Werte	Werte der Luminanz-Quantisierungs-Tabelle.
0x5d	0x01	ID der 2.Quantisierungs-Tabelle. 01 für Chrominanz

Tabelle 3.6.: Erläuterungen zum DQT-Abschnitt

[14] Wieviele Pixel pro Zentimeter oder Inch

SOF0-Marker, 0xff 0xc0

Der „Start Of Frame“-Abschnitt(Abb. 3.9) enthält Informationen über die Größe des Bildes, das Farbmodel, Subsampling und welche Quantisierungs-Tabelle wo benutzt wird.

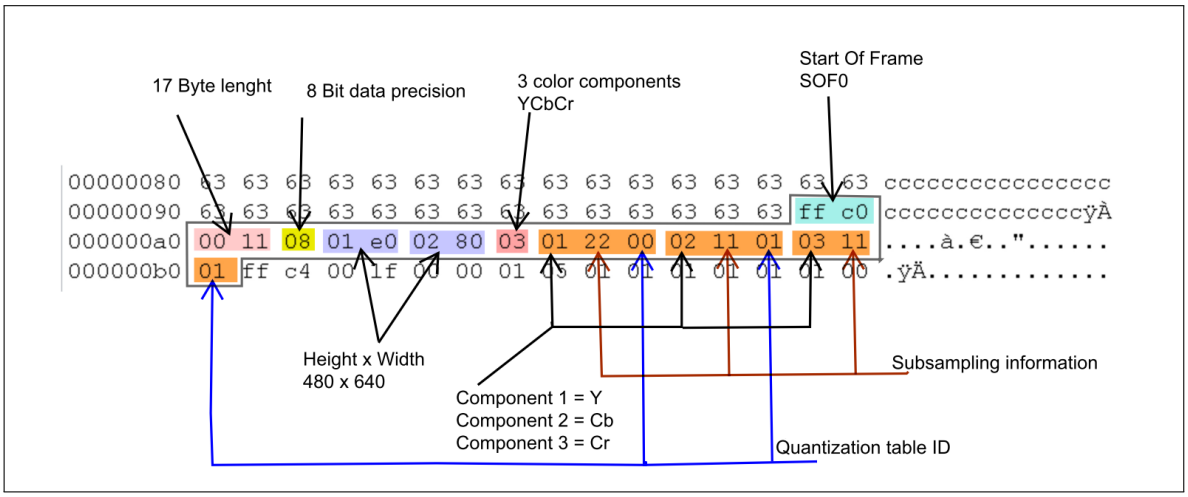


Abbildung 3.9.: SOF0-Marker-Abschnitt

DHT-Marker, 0xff 0xc4

„Define Huffman Table“. Das Ablesen und die Decodierung der Huffman-Tabellen sind ein wichtiger Teil dieser Arbeit, da die Software anhand dieser Tabellen die Koeffizienten im Bitstrom lokalisieren und decodieren kann.
Es gibt vier Huffman-Tabellen: Luminanz und Chrominanz haben jeweils Tabellen für AC- und DC-Koeffizienten. Diese Tabellen werden wie folgt identifiziert:

DHT x1x2 mit:
x1 = DHT-Class, 0 für DC, 1 für AC
x2 = DHT-ID, 0 für Luminanz, 1 für Chrominanz

z. B. die gelb markierten Byte aus der Tabelle 3.10:
DHT 00: DC, Luminanz(Y), Offset: 0xb1
DHT 10: AC, Luminanz(Y), Offset: 0xd2
DHT 01: DC, Chrominanz(Cb & Cr), Offset: 0x01 0x89
DHT 11: AC, Chrominanz(Cb & Cr). Offset: 0x01 0xaa

Huffman-Tabellen

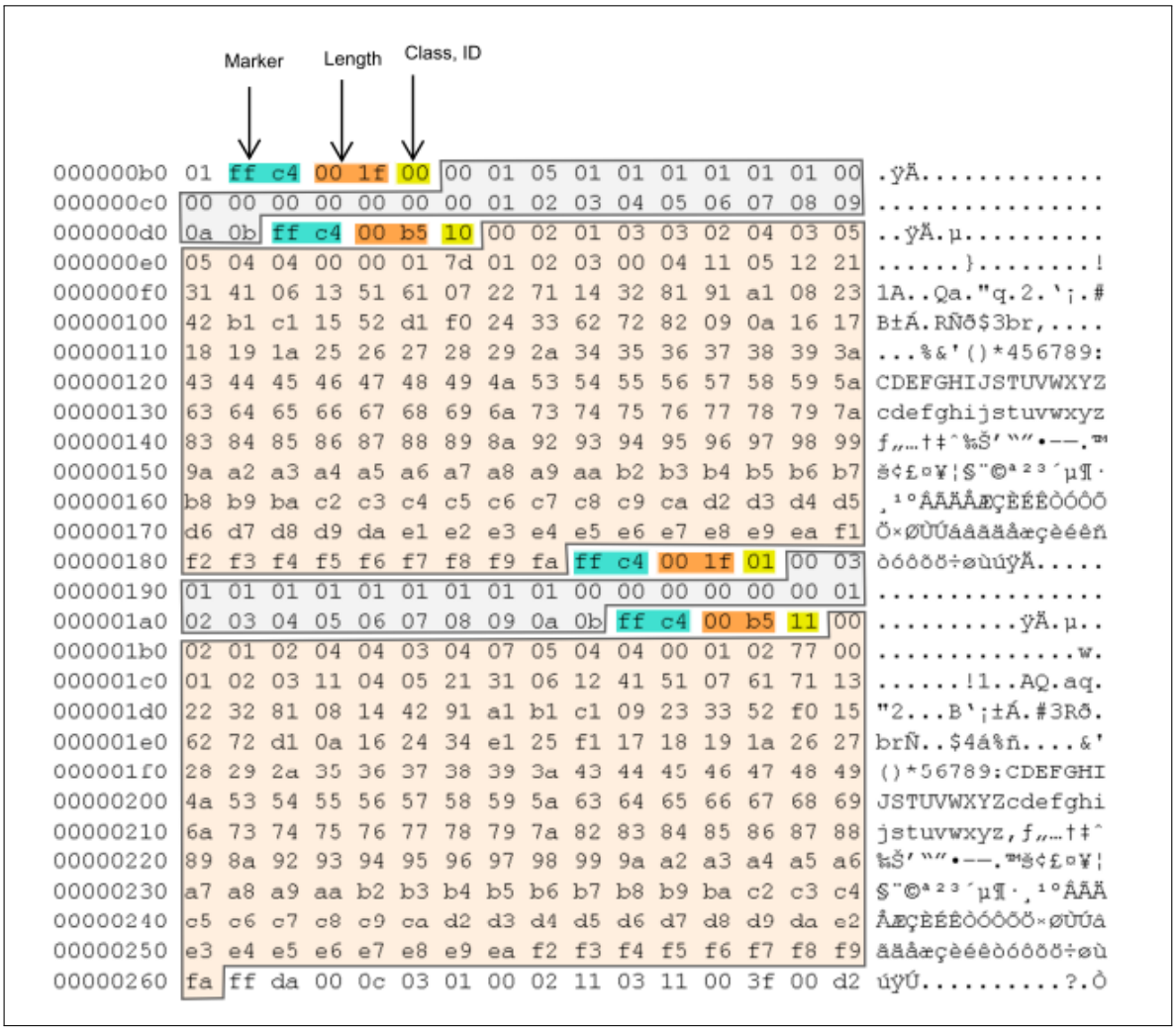


Abbildung 3.10.: DHT-Marker-Abschnitt

Die Hufmann-Codierung im JPEG-Format ist in einer ungewöhnlichen Art dargestellt. Anhand der folgenden Beispiel-Tabellen wird ein besserer Einblick verschafft.
Die Speicherung der Huffman-Tabellen erfolgt in zwei Teilen. In den ersten 16 Byte wird die Anzahl der Bitfolgen mit der entsprechenden Länge(1-16) gespeichert. Für die erste Huffman-Tabelle in Abb. 3.10 ergeben sich so 0x00 Bitfolgen der Länge 1, 0x01 Bitfolge der Länge 2, 0x05 Bitfolgen der Länge 3, sowie jeweils 0x01 Bitfolge der Längen 4...9, also insgesamt 12 Bitfolgen. In den folgenden 12 Byte stehen die Codes, die diesen Bitfolgen zugeordnet sind. Daraus ergibt sich im ersten Schritt Tabelle 3.7.
Im zweiten Schritt werden die entsprechenden Bitfolgen für die Codes in Tabelle 3.7 konstruiert. Das Ergebnis ist in Tabelle 3.8 zusammengefasst.

Länge der Bitfolge	Anzahl	Codes
1	0x00	-
2	0x01	0x00
3	0x05	0x01, 0x02, 0x03, 0x04, 0x05
4	0x01	0x06
5	0x01	0x07
6	0x01	0x08
7	0x01	0x09
8	0x01	0x0a
9	0x01	0x0b
10	0x00	-
⋮	⋮	⋮
16	0x00	-

Tabelle 3.7.: Huffman-Luminanz-DC(**DHT 00**) ohne Bitfolgen

Länge der Bitfolge	Code	Bitfolge
2	0x00	00
3	0x01	010
	0x02	011
	0x03	100
	0x04	101
	0x05	110
4	0x06	1110
5	0x07	11110
6	0x08	111110
7	0x09	1111110
8	0x0a	11111110
9	0x0b	111111110

Tabelle 3.8.: Huffman-Luminanz-DC(**DHT 00**) mit Bitfolgen

Wie in Kapitel 3.6 erklärt wurde, wird bei den DC-Koeffizienten nur die Bit-Anzahl Huffman-codiert, die benötigt wird, um den Koeffizienten VLI-codiert darzustellen. Hierfür stehen bei den DC-Koeffizienten 0...11 Bit(0x00...0x0b) zur Verfügung, daher sind die beiden Tabellen

für die DC-Koeffizienten relativ kurz.
Bei den AC-Koeffizienten werden zusätzlich noch die RLZ-Werte(Run Length Zero) codiert. Hierfür werden die ersten 4 Bit des Codes genutzt und es können Werte zwischen 0 und 15 auftreten. In den zweiten 4 Bit wird wieder die Bit-Anzahl für den VLI-codierten Koeffizienten ungleich Null angegeben, der maximal 10 Bit lang sein kann.
Somit ergeben sich maximal 176 Codes (0...f und 0...a $\rightarrow 16 \cdot 11 = 176$). Da eine Null im Koeffizienten über die RLZ codiert wird, tritt eine Null im zweiten Teil nur in den zwei Spezialfällen auf:
Einmal beim „End of Block“(0x00) und bei mehr als 15 Nullen in Folge(0xf0), daher ergeben sich nur 162 Codes.
Aus der Abbildung 3.10 werden jetzt aus den ersten 16 Byte der zweiten Huffman-Tabelle die Anzahl der jeweiligen Bitfolgen ausgelesen und aus den nächsten 162 Byte die zugehörigen Codes. Diese sind nach der Häufigkeit ihres Auftretens sortiert, der häufigste Code zuerst, so dass dieser die kürzeste Bitfolge zugeordnet bekommt. In Tabelle 3.9 sind die Anzahl und die zugeordneten Codes dargestellt. In Tabelle 3.10 sind dann auch die Bitfolgen zugeordnet.

Länge der Bitfolge	Anzahl	Codes
1	0x00	-
2	0x02	0x01, 0x02
3	0x01	0x03
4	0x03	0x00(EOB), 0x04, 0x11
5	0x03	0x05, 0x12, 0x21
6	0x02	0x31, 0x41
7	0x04	0x06, 0x13, 0x51, 0x61
8	0x03	0x07, 0x22, 0x71
9	0x05	0x14, 0x32, 0x81, 0x91, 0xa1
10	0x05	0x08, 0x23, 0x42, 0xb1, 0xc1
11	0x04	0x15, 0x52, 0xd1, 0xf0
12	0x04	0x24, 0x33, 0x62, 0x72
13	0x00	-
14	0x00	-
15	0x01	0x82
16	0x7d(125)	0x09, 0x0a, 0x16 ... 0xfa (In Abb. 3.10 von Adresse 0x10c bis 0x188)

Tabelle 3.9.: Huffman-Luminanz-AC(**DHT 10**) ohne Bitfolge

Länge der Bitfolge	Code	Bitfolge
2	0x01, 0x02	00, 01
3	0x03	100
4	0x00 0x04 0x11	1010 1011 1100
5	0x05 0x12 0x21	11010 11011 11100
6	0x31 0x41	111010 111011
7	0x06 0x13 0x51 0x61	1111000 1111001 1111010 1111011
⋮	⋮	⋮
16	0xfa	1111111111111110

Tabelle 3.10.: Huffman-Luminanz-AC(**DHT 10**) mit Bitfolge

SOS-Marker, 0xff 0xda

Im „Start of Scan“-Abschnitt folgen nach einigen Informationen, die in Abb. 3.11 markiert und erklärt sind, die kodierten Bilddaten (hier ab Adresse 0x26f). Das Ende der Bilddaten und damit das Ende der JPEG-Codierung wird mit dem „End Of Image“-Marker (EOI, 0xff 0xd9) gekennzeichnet. Falls nach diesem Marker noch Daten folgen, werden diese vom JPEG-Decoder nicht gelesen.

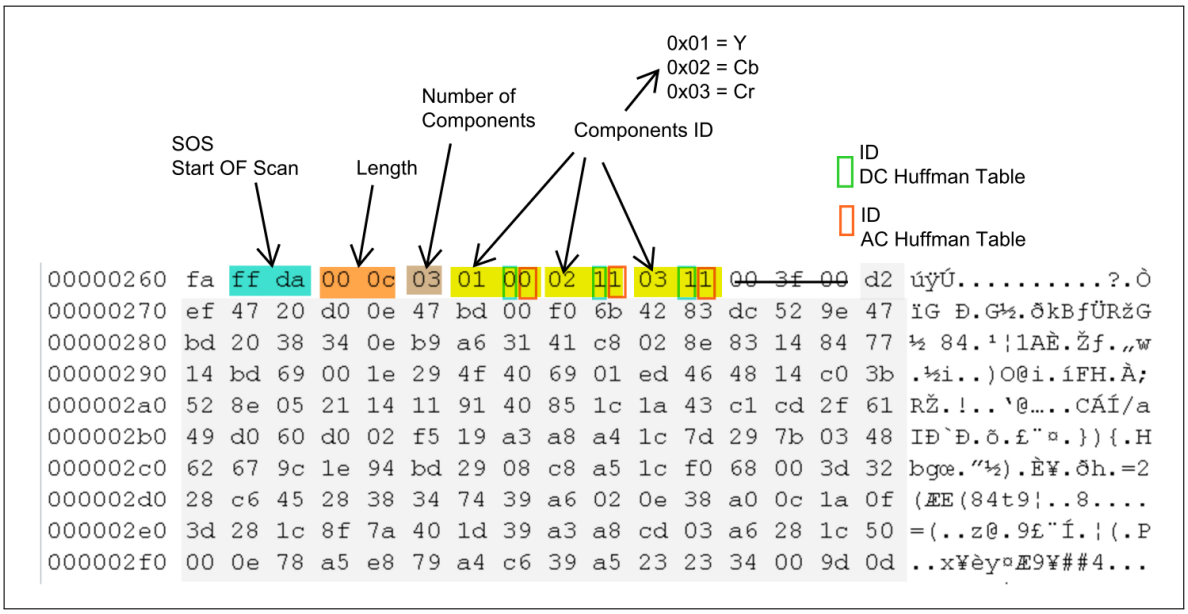


Abbildung 3.11.: SOS-Marker-Abschnitt

Jetzt werden, mit den oben entschlüsselten Informationen und Tabellen, die ersten Koeffizienten decodiert werden.
Aus der Abb. 3.11 werden die Werte ab Adresse 0x26f nach und nach ausgelesen. Nach den oben extrahierten Informationen zum Subsampling(4:2:0), werden für jede MCU sechs 8x8-Koeffizienten-Blöcke dekodiert(vier Luminanz und zwei Chrominanz).

[Hex] d2 ef 47 20 d0 0e 47 bd
11010010 11101111 01000111 00100000 11010000 00001110 01000111 10111101

11010010 11101111 01000111 00100000 11010000 00001110 01000111 10111101
Aus der Huffman-Tabelle 3.8 für DC-Luminanz: $110 \rightarrow Code = 0x05$, d.h. die nächsten 5 Bit stellen den VLI-codierten Wert dar:

11010010 11101111 01000111 00100000 11010000 00001110 01000111 10111101
Aus der Tabelle 3.4: 10010 = 18
→ MCU00, Luminanz, DC quantisiert = 18 → dequantisiert = $18 \cdot 16 = 288$ (16 ist der Quantisierungs-Faktor aus Abb. 3.8)

1101001011101111 01000111 00100000 11010000 00001110 01000111 10111101
Aus der Huffman-Tabelle 3.10 für AC-Luminanz: 111011 → $Code = 0x41$
→ $RLZ = 4$ → AC1...AC4 = 0 und 1 Bit für AC5

1101001011101111 01000111 00100000 11010000 00001110 01000111 10111101
Aus der Tabelle 3.5: 1 = 1
→ MCU00, Luminanz, AC5 quantisiert = 1 → dequantisiert = $1 \cdot 10 = 10$

1101001011101111 01000111 00100000 11010000 00001110 01000111 10111101
Aus der Tabelle 3.10: 1010 → $Code = 0x00$ → **EOB**

In Tabelle 3.11 werden die decodierten Koeffizienten zusammengefasst. Hier ist für die AC-Koeffizienten die korrekte Reihenfolge nach dem Zick-Zack-Muster aus Kapitel 3.5 zu beachten:

288	0	10	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabelle 3.11.: Luminanz Koeffizienten Y00 von MCU00

Anmerkung
Die Blöcke werden in dieser Reihenfolge ausgelesen:

$$\underbrace{Y00, Y10, Y01, Y11, Cb, Cr}_{MCU00}\ \underbrace{Y00, Y10, Y01, Y11, Cb, Cr}_{MCU10}\ \underbrace{Y00, Y10, Y01, Y11, Cb, Cr}_{MCU20}\ldots$$

4. Kryptographie

4.1. Symmetrische Verschlüsselung vs. Asymmetrische Verschlüsselung

Symmetrische Kryptographie, auch Secret-Key-Verfahren gennant, sind die Verfahren, bei denen jeweils zwei Teilnehmer für die Kommunikation untereinander ein und den selben Schlüssel benutzen. Dieser wird sowohl zur Verschlüsselung als auch zur Entschlüsselung der Nachrichten verwendet. Dieser Schlüssel muss geheim gehalten werden und die beiden müssen den Schlüssel über einen sicheren Kanal austauschen. Alle diese Schlüssel, die von einer Gruppe von Teilnehmern gebraucht wird, müssen auf einem Chip gespeichert werden und bei Änderungen an der Gruppe müssen diese aktualisiert werden. Dieser Schlüsselaustausch ist ein Problem beim symmetrischen Verfahren.
Das zweite Problem ist die Anzahl dieser Schlüssel, wenn eine größere Gruppe untereinander verschlüsselt kommunizieren will. Allgemein gilt:

n = Anzahl der Teilnehmer A = Anzahl der Schlüssel

$$A = \frac{n}{2}.(n-1) \tag{4.1}$$

z. B.
2 Teilnehmer$\rightarrow$ 1 Schlüssel
5 Teilnehmer $\rightarrow$ 10 Schlüssel
20 Teilnehmer $\rightarrow$ 190 Schlüssel
50 Teilnehmer $\rightarrow$ 1225 Schlüssel

Bei der asymmetrischen Kryptographie, auch Public-Key-Verfahren genannt, hat jeder Teilnehmer einen öffentlichen Schlüssel, der für alle anderen zur Verfügung steht. Somit entfällt der Schlüsselaustausch. Ein Teilnehmer nutzt den öffentlichen Schlüssel eines zweiten Teilnehmers, um eine Nachricht zu verschlüsseln. Die verschlüsselte Nachricht kann nun nur mit dem privaten Schlüssel des zweiten Teilnehmers entschlüsselt werden.
Jedes dieser Verfahren hat seine Vor- und Nachteile. In in der Tabelle 4.1 werden diese Merkmale verglichen.

Symmetrisches Verfahren	Asymmetrisches Verfahren
AES, DES, 3DES, ...	RSA, Diffe-Hellman, ECC, ...
Schlüsselaustausch notwendig	Kein Schlüsselaustausch
Basieren auf einfachen, Bit-orientierten Funktionen, daher einfacher, neue Verfahren zu entwickeln	Basieren auf mathematisch anspruchsvolleren Funktionen, daher schwieriger ein neues Verfahren zu entwickeln
Erfordert weniger Rechenkraft, daher mehr Geschwindigkeit	Erfordert wesentlich mehr an Rechenkraft, daher langsamer, z. B. RSA etwa um den Faktor 1000 langsamer als AES
–	Komplizierter und daher anfälliger für Implementierungsfehler
Schlüssel, Klartext und Geheimtext als Bit-Folgen	Schlüssel, Klartext und Geheimtext als große Zahlen
Schlüssellänge meist vorgegeben oder aus verschiedenen Vorgaben wählbar	Schlüssellänge völlig variabel

Tabelle 4.1.: Merkmale symmetrischer Kryptographie vs. asymmetrischer Kryptographie

Hier ist zu bemerken, dass alle oben genannten Unterschiede aus der Praxis und Erfahrungstatsachen kommen und kein mathematischer Beweis vorliegt. [Schmeh (2016)]

4.2. Hybridverfahren

Um das Problem beim Schlüsselaustausch zu umgehen und gleichzeitig von der hohen Geschwindigkeit des symmetrischen Verfahrens zu profitieren, wurde das sogenannte Hybridverfahren entwickelt. Dieses verwendet einen geheimen Schlüssel und einen öffentlichen Schlüssel. Es wird durch ein asymmetrisches Verfahren ein geheimer Schlüssel für das nachfolgende symmetrische Verfahren übermittelt.
Wenn ein Teilnehmer eine geheime Nachricht zu einem anderen Teilnehmer schicken möchte, verschlüsselt er seine Nachricht mit einem geheimen Schlüssel. Dann verschlüsselt er diesen geheimen Schlüssel mit dem öffentlichen Schlüssel seines Kommunikations-Partners und schickt diesem den verschlüsselten geheimen Schlüssel, sowie die mit diesem verschlüsselte Nachricht. Der Kommunikationspartner entschlüsselt zuerst den geheimen Schlüssel mit seinem eigenen privaten Schlüssel. Mit dem entschlüsselten geheimem Schlüssel, kann die Nachricht dann mit höherer Geschwindigkeit entschlüsselt werden.

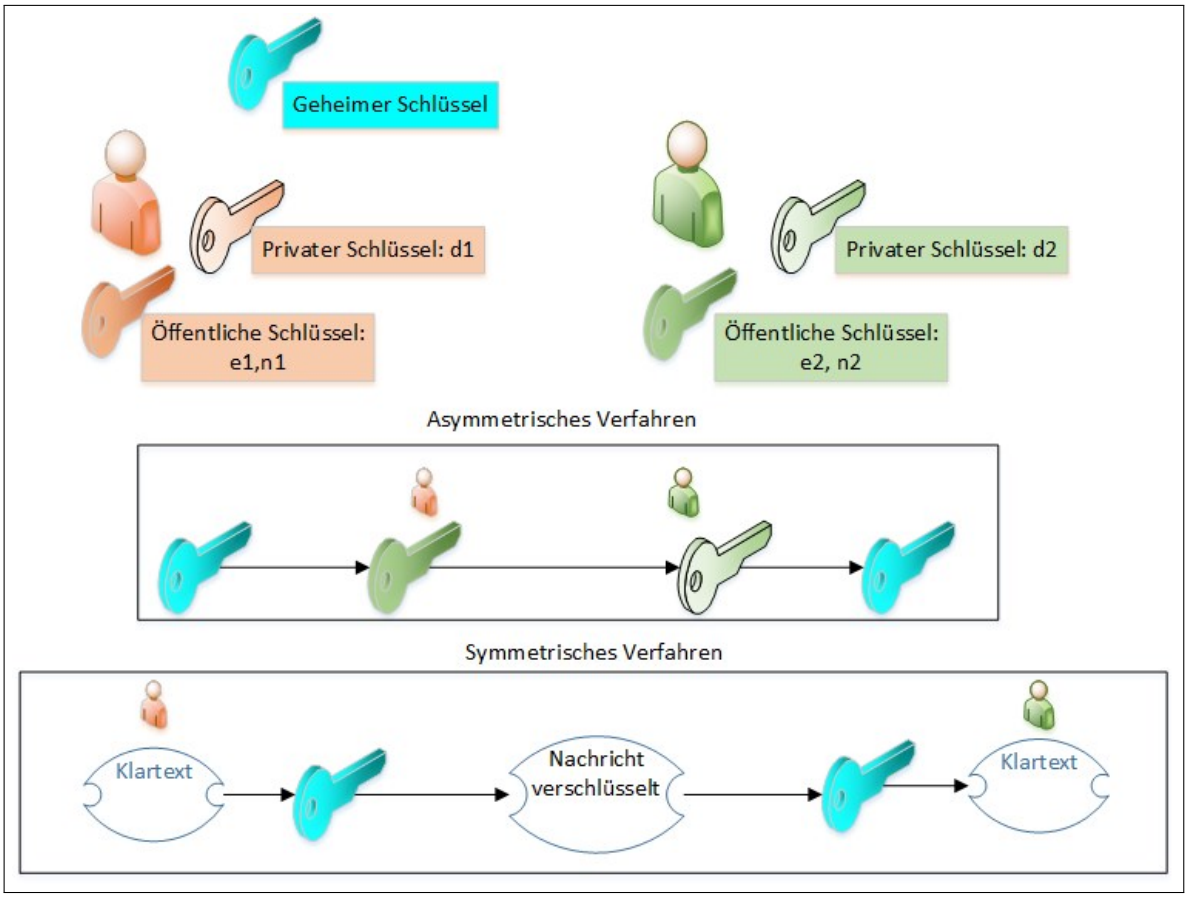


Abbildung 4.1.: Hybridverfahren mit RSA

4.3. RSA

Für diese Arbeit eignet sich am besten das asymmetrische RSA-Verfahren[1][2]. Einerseits braucht man so keinen geheimen Schlüsselaustausch wie bei einem symmetrischen Verfahren. So kann mit der entwickelten Software und dem öffentlichen Schlüssel eines Kommunikations-Partners einfach eine Nachricht verschlüsselt und weitergegeben werden. Oder die Software kann genutzt werden, um den geheimen Schlüssel für ein symmetrisches Verfahren zu verschlüsseln und zu übermitteln. Andererseits werden die verschlüsselten Nachrichten in einer JPEG-Datei versteckt und dieses Bildformat als Container bietet nicht so viel Platz für die Nachrichten. Daher bleiben die Nachrichten klein und somit wirkt sich die geringere Geschwindigkeit des asymmetrischen Verfahrens nicht so stark aus.

Das RSA-Verfahren basiert auf der Berechnung einer diskreten Exponentialfunktion (Gleichung 4.2). Diese Funktion kann in der asymmetrischen Kryptographie eingesetzt werden, weil sie sich als „Einwegfunktion“[3] erwiesen hat. Das heißt, für ihre Umkehrfunktion „diskreter Logarithmus“ existiert kein schneller Algorithmus.

$$a^b \bmod n = c \tag{4.2}$$

wobei c der Rest der Division von a^b durch n ist.

z. B.:
$7^5 \bmod 3 = 1$

In der Kryptographie benutzt man große Zahlen, die z. B. 512, 1024 oder 2048 Bit lang sind und nicht mit einer Integer-Variable darzustellen sind[4]. Damit das funktioniert, muss die diskrete Exponentialfunktion für diese großen Zahlen effizient berechnet werden. Dazu kann der „Satz von Euler“ oder das „Square & Multiply“-Verfahren verwendet werden[5] (Siehe Kapitel 4.3.5).

[1] Ron Rivest, Adi Shamir, Leonard Adleman (RSA)

[2] Liste aller Begriffe, die helfen, dieses Verfahren zu verstehen: **Einwegfunktion, diskrete Exponentialfunktion, diskreter Logarithmus, Primfaktorzerlegung, Eulersche Phi-Funktion, Satz von Euler-Fermat, Erweiterter Euklidischer Algorithmus, Square-and-Multiply-Algorithmus**

[3] Einwegfunktionen sind die mathematischen Funktionen, die leicht zu berechnen sind, aber deren Umkehrfunktion sehr kompliziert und praktisch nicht in angemessener Zeit zu berechnen sind. RSA verwendet zwei Funktionen dieser Art: **Primfaktorzerlegung** und **diskreten Logarithmus**.

[4] Um große Zahlen darstellen zu können, wird in dieser Arbeit die FLINT Bibliothek genutzt.

[5] In dieser Arbeit wird das „Square & Multiply“-Verfahren bei der Entwicklung verwendet.

Verschlüsseln mit RSA

$$c = m^e \bmod n \tag{4.3}$$

wobei m die Nachricht ist, e und n sind die öffentlichen Schlüssel des Empfängers der Nachricht und c ist die verschlüsselte Nachricht.

Entschlüsseln mit RSA

$$m = c^d \bmod n \tag{4.4}$$

wobei m die entschlüsselte Nachricht ist, c die verschlüsselte Nachricht, d der private Schlüssel des Empfängers und n einer seiner öffentlichen Schlüssel.

4.3.1. RSA-Schlüssel generieren

Beim RSA-Verfahren braucht jeder Teilnehmer drei Schlüssel, einen privaten Schlüssel d und zwei öffentliche Schlüssel e und n. Zur Generierung der drei Schlüssel werden die folgenden Schritte durchlaufen:

- Zwei große Primzahlen p und q auswählen.
- $n = p \cdot q$ rechnen.
- $\phi(n)$ berechnen. Nach den Gleichungen 4.6 und 4.7 ist: $\phi(n) = (p-1) \cdot (q-1)$
- Eine Primzahl e auswählen, so dass $1 < e < \phi(n)$ und der größte gemeinsame Teiler $ggT(e, \phi(n)) = 1$ ist.
- d berechnen, so dass $e \cdot d \bmod \phi(n) = 1$ gilt.
  Zur Berechnung wird der „erweiterte euklidische Algorithmus“ verwendet(Siehe Formel 4.9).

Dabei sind p, q und $\phi(n)$ geheime Parameter.

4.3.2. Eulersche Phi-Funktion

Die eulersche ϕ-Funktion gibt für jede natürliche Zahl n an, wie viele zu n teilerfremde[6] natürliche Zahlen es gibt, die kleiner als n sind.

$$\phi(n) =: \{a \in \mathbb{N} | \, (1 \leq a < n) \wedge ggT\,(a, n) = 1\} \tag{4.5}$$

[6] Die natürlichen Zahlen a und b sind teilerfremd, wenn der größte gemeinsame Teiler $ggT(a, b) = 1$ ist.

Ein Beispiel:
$\phi(10) = 4$, nämlich $(1, 3, 7, 9)$
$\phi(7) = 6, (1, 2, 3, 4, 5, 6)$
$\phi(13) = 12, (1, 2, 3, 4, 5, 6, \ldots, 12)$

Das bedeutet, dass für eine Primzahl p die eulersche Phi-Funktion nach Gleichung 4.6 berechnet wird:

$$\phi(p) = p - 1 \tag{4.6}$$

Diese Phi-Funktion hat die Eigenschaft, dass sie eine multiplikative Funktion ist. Das heißt, für teilerfremde natürliche Zahlen m und n gilt:

$$\phi(m \cdot n) = \phi(m) \cdot \phi(n) \tag{4.7}$$

Mit Hilfe dieses Satzes und der Primfaktorzerlegung kann $\phi(n)$ für beliebige $n \in \mathbb{N}$, $n \geq 2$ leicht berechnet werden:

$$n = p_1^{k_1} \cdot p_2^{k_2} \cdot \ldots \cdot p_r^{k_r}$$

$$\Rightarrow \phi(n) = (p_1^{k_1} - p_1^{k_1-1}) \cdot (p_2^{k_2} - p_2^{k_2-1}) \ldots (p_r^{k_r} - p_r^{k_r-1}) \tag{4.8}$$

Wobei $p_1, \ldots p_r$ verschiedene Primzahlen sind und $k_i \geq 1$, $i = 1 \ldots r$.

Ein Beispiel: $\phi(91) = ?$
$\phi(91) = \phi(13 \cdot 7) = \phi(13) \cdot \phi(7) = 12 \cdot 6 = 72$

Ein zweites Beispiel: $\phi(360) = ?$
$360 = 2^3 \cdot 3^2 \cdot 5^1$

$NachGleichung\ 4.8 \Rightarrow \phi(360) = (2^3 - 2^2) \cdot (3^2 - 3^1) \cdot (5^1 - 5^0)$
$\Rightarrow \phi(360) = 4 \cdot 6 \cdot 4 = 96$

4.3.3. Erweiterter euklidischer Algorithmus

Der „euklidische Algorithmus“ bzw. der „moderne euklidische Algorithmus“ berechnet den größten gemeinsamen Teiler(ggT) zweier natürlicher Zahlen.

Ein Beispiel:
$ggT(56, 22) = ?$
$56 = 22 \cdot 2 + 12$
$22 = 12 \cdot 1 + 10$
$12 = 10 \cdot 1 + 2$
$10 = \underbrace{2}_{ggT} \cdot 5 + 0$
Der „Erweiterte euklidische Algorithmus“ berechnet neben dem ggT zweier natürlicher Zahlen zusätzlich noch zwei ganze Zahlen, die die Gleichung 4.9 erfüllen.

$$ggT(a, b) = v \cdot a + u \cdot b \tag{4.9}$$

wobei a und $b \in \mathbb{N}$, v und $u \in \mathbb{Z}$.

z. B. die Gleichung 4.9 für die Zahlen 876 und 98 berechnen:
$876 = 98 \cdot 8 + 92$
$98 = 92 \cdot 1 + 6$
$92 = 6 \cdot 15 + 2$
$6 = \underbrace{2}_{ggT} \cdot 3 + 0 \Rightarrow ggT(876, 98) = 2$

$2 = 92 - 6 \cdot 15$
$= 92 - 15 \cdot (98 - 92 \cdot 1)$
$= 92 - 15 \cdot 98 + 15 \cdot 92 \cdot 1$
$= -15 \cdot 98 + (15 + 1) \cdot 92 = -15 \cdot 98 + 16 \cdot 92$
$= -15 \cdot 98 + 16 \cdot (876 - 98 \cdot 8)$
$= -15 \cdot 98 + 16 \cdot 876 - 16 \cdot 98 \cdot 8 = -15 \cdot 98 + 16 \cdot 876 - 128 \cdot 98 = 16 \cdot 876 - (15 + 128) \cdot 98$
$= \underbrace{16}_{v} \cdot 876 \underbrace{-143}_{u} \cdot 98$
$\Longrightarrow ggT(876, 98) = 2 = \underline{16} \cdot 876 \underline{-143} \cdot 98$

4.3.4. Satz von Euler-Fermat

Mit dem Satz von Euler-Fermat kann die Berechnung großer Exponenten modulo n vereinfacht werden. Der Satz lautet:

Wenn $ggT(a, n) = 1$, dann gilt:

$$a^{\phi(n)} \bmod n = 1 \tag{4.10}$$

Wobei $n \in \mathbb{N}$ und $a \in \mathbb{Z}$

Dies bedeutet konkret:
Sei $ggT(a, n) = 1$ dann gilt:

$$a^s \bmod n = a^f \bmod n \tag{4.11}$$

mit $f = s \bmod \phi(n)$

Für den Spezialfall, dass n eine Primzahl p ist, gilt:
Wenn $ggT(a, p) = 1$, dann:

$$a^{p-1} \bmod p = 1 \quad (SieheGleichung\ 4.6) \tag{4.12}$$

Ein Beispiel: $6^{54321} \bmod 7 = ?$
$ggT(6, 7) = 1$, Bedingung erfüllt, der Satz kann benutzt werden.

7 ist eine Primzahl, also gilt nach Gleichung 4.6: $\phi(7) = 6$

$54321 \bmod 6 = 3$

Nach Gleichung 4.11 $\Rightarrow 6^{54321} \bmod 7 = 6^3 \bmod 7 = 6$

Ein zweites Beispiel: $3^{987654} \bmod 100 = ?$
$ggT(3, 100) = 1$, Bedingung erfüllt, der Satz kann benutzt werden.

$\phi(100) = ?$
$100 = 2^2 \cdot 5^2$
Nach Gleichung 4.8 $\Rightarrow \phi(100) = (2^2 - 2^1) \cdot (5^2 - 5^1) = 2 \cdot 20 = 40$

$987654 \bmod 40 = 14$

Nach Gleichung 4.11 $\Rightarrow 3^{987654} \bmod 100 = 3^{14} \bmod 100 = 69$

4.3.5. Square-and-Multiply-Algorithmus

Dieser Algorithmus[7] berechnet, wie auch der oben beschriebene „Satz von Euler-Fermat“, natürliche Potenzen effizient[8]. Bei diesem Algorithmus wird zuerst die Potenz in Binärform dargestellt. Dann werden genau so viele Schritte benötigt, wie die Anzahl der Stellen der Binärdarstellung der Potenz, um das Ergebnis zu erhalten. Bei der Programmierung wird beim hochwertigsten Bit begonnen. In jedem Schritt wird das Ergebnis vom vorherigen Schritt quadriert und, wenn das Bit gesetzt ist, mit der Basis multipliziert. Dann wird jeweils die Modulo-Operation ausgeführt

Ein Beispiel: $3^{53} \bmod 7 = ?$
$(53)_{10} = (110101)_2 \Rightarrow$ 6 Schritte bis zum Ergebnis.

Beginn: $x_0 = 1$
1.Schritt: Bit5 = 1 $\Rightarrow x_1 = 1^2 \cdot 3^1 \bmod 7 = \ 3 \bmod 7 = 3$
2.Schritt: Bit4 = 1 $\Rightarrow x_2 = 3^2 \cdot 3^1 \bmod 7 = 27 \bmod 7 = 6$
3.Schritt: Bit3 = 0 $\Rightarrow x_3 = 6^2 \cdot 3^0 \bmod 7 = 36 \bmod 7 = 1$
4.Schritt: Bit2 = 1 $\Rightarrow x_4 = 1^2 \cdot 3^1 \bmod 7 = \ 3 \bmod 7 = 3$
5.Schritt: Bit1 = 0 $\Rightarrow x_5 = 3^2 \cdot 3^0 \bmod 7 = \ 9 \bmod 7 = 2$
6.Schritt: Bit0 = 1 $\Rightarrow x_6 = 2^2 \cdot 3^1 \bmod 7 = 12 \bmod 7 = 5$
$\Rightarrow 3^{53} \bmod 7 = 5$

Nun das gleiche Beispiel mit dem „Satz von Euler-Fermat“:
$ggt(3, 7) = 1 \Rightarrow$ Der Satz kann verwendet werden.
$\phi(7) = 6$
$53 \bmod 6 = 5$
$3^{53} \bmod 7 = 3^5 \bmod 7 = 5$

Wie an diesem Beispiel gezeigt, ist die Rechnung nach dem „Satz von Euler-Fermat“ kürzer, aber kann nur verwendet werden, wenn Basis und Divisor zu einander teilerfremd sind. Der „Square-and-Multiply-Algorithmus“ ist sehr einfach zu programmieren und kann für jede Zahlenkombination eingesetzt zu werden. Daher wird in dieser Arbeit der „Square-and-Multiply-Algorithmus“ verwendet.

[7] Auch binäre Exponentiation genannt
[8] Die in dieser Arbeit entwickelte Software verwendet diesen Algorithmus

4.3.6. CRT in RSA

Der chinesische Restsatz, Chinese remainder theorem (CRT), ist ein Theorem der Zahlentheorie, der beim RSA-Verfahren verwendet wird, um die Geschwindigkeit zu erhöhen. Um dieses Theorem nutzen zu können, muss man die geheimen Parameter p und q wissen. Damit ist dieses Verfahren nur bei der Entschlüsselung und beim Signieren[9] einsetzbar.
Bei der Entschlüsselung funktioniert es wie folgt:

- Beim Entschlüsseln ist c der Geheimtext: $m = c^d \mod n$, wobei m der Klartext ist und d der private Schlüssel
- Berechne die „Teilschlüssel“ d_p und d_q (p und q sind die geheimen RSA-Parameter, Siehe Kapital 4.3.1):
  $d_p = d \mod (p-1)$
  $d_q = d \mod (q-1)$
- Berechne die „Teilnachrichten“ m_p und m_q:
  $m_p = c^{d_p} \mod p$
  $m_q = c^{d_q} \mod q$
- Berechne für p und q mit dem „erweiterten euklidischen Algorithmus“ nach Gleichung 4.9 v und u:
  $ggT(p, q) = 1 = v \cdot p + u \cdot q$
- Berechne m aus den „Teilnachrichten“:
  $m = v \cdot p \cdot m_q + u \cdot q \cdot m_p$

Dieser Trick erhöht die Geschwindigkeit der Entschlüsselung. Aber beim Signieren wird das Verfahren anfälliger gegen Angriffe. Das heißt, es reicht für den Angreifer, wenn der Sender beim Signieren durch eine Glitch-Attacke[10] einen Teil der Signatur, p oder q, falsch berechnet und so eine falsche Signatur schickt. Der Empfänger, der die Glitch-Attacke absichtlich verursacht hat, kann daraus die geheimen Parameter p und q berechnen.

[9] Signieren mit RSA: $sig = m^d \mod n$, wobei d der private und n ein öffentlicher Schlüssel des Senders ist
Verifizieren mit RSA: $m = sig^e \mod n$, wobei e und n die öffentlichen Schlüssel des Senders sind

[10] Bei einer Glitch-Attacke wird die Ausführung von Maschinenbefehlen kurz unterbrochen(z. B. durch Spannungsabfall), so dass es zu fehlerhaften Berechnungen kommen kann

5. Huffman

5.1. Huffman-Codierung

Da das JPEG-Format die Bilddaten sehr stark komprimiert, bieten die Bilddateien nur relativ wenig Platz, um darin Informationen zu verstecken. Daher wurde entschieden, optional eine Methode zur Komprimierung der Original-Nachricht anzubieten.
Da der Originaltext selbstverständlich komplett rekonstruiert werden muss, kommen nur verlustfreie Kompressionsmethoden in Frage. Es wurden die Huffman-Codierung und das LZW-Verfahren näher betrachtet.
Da es in dieser Arbeit nicht um die Optimierung von Kompressionsverfahren geht und die Huffman-Codierung ein sehr bewährtes und relativ unkompliziertes Verfahren ist, fiel die Entscheidung auf die Huffman-Codierung. Außerdem nutzt das JPEG-Format ebenfalls die Huffman-Codierung.
In der Originalnachricht wird der erweiterte ASCII-Code genutzt, in dem jedes Zeichen eine feste Länge von 8 Bit hat(Fixed Length Code). Die Huffman-Codierung nutzt die unterschiedliche Häufigkeit von Zeichen in einem Text aus, indem für häufig vorkommende Zeichen kurze Bit-Codes und für selten vorkommende Zeichen lange Bit-Codes genutzt werden(Variable Length Code). Hierdurch werden Redundanzen in der Originalnachricht reduziert.

5.1.1. Huffman-Baum

Um den besten Variable-Length-Code für jeden Text zu finden, nutzt die Huffman-Codierung einen Suchbaum, den Huffman-Baum. Der Huffman-Baum wird von unten, also von den Blättern, nach oben zur Wurzel aufgebaut.
Im ersten Schritt werden die Häufigkeiten der auftretenden Zeichen im Text gezählt und in einer Häufigkeitstabelle abgelegt. Für jedes Zeichen aus der Häufigkeitstabelle wird im nächsten Schritt ein Blattknoten angelegt. Dieser enthält neben dem Zeichen die Häufigkeit als Gewicht. Die einzelnen Blattknoten können auch als Teilbäume angesehen werden. Aus der Menge der Teilbäume werden die beiden mit dem geringsten Gewicht entnommen. Es

wird ein neuer Knoten erzeugt, der die beiden Teilbäume als Kinder bekommt und die Summe der beiden Gewichte als eigenes Gewicht. Als Zeichen wird für diesen neuen Knoten ein Zeichen ausgewählt, dass nicht im Text vorkommt. Dieser neue Teilbaum wird wieder der Menge der Teilbäume hinzugefügt. Dieser Schritt wird wiederholt, bis alle Teilbäume zu einem einzigen Baum zusammengefügt sind.

5.1.2. Huffman-Code

Aus dem erstellten Baum kann jetzt für jedes Blatt ein Code mit variabler Länge erzeugt werden. Hierzu wird von der Wurzel ausgehend jede linke Kante als 0 interpretiert und jede rechte Kante als 1. Dadurch ist sichergestellt, dass kein Codewort dem Anfang eines anderen Codewortes entspricht. Der Code wird als präfixfreier Code bezeichnet und benötigt somit trotz der variablen Länge der Codeworte keine Trennzeichen.
Aus dem Huffman-Baum wird jetzt eine Code-Tabelle erstellt, die zu jedem Zeichen den Bitcode enthält. Mit dieser Tabelle wird der Originaltext jetzt kodiert.
Um den kodierten Text entschlüsseln zu können, muss das verwendete Zeichen für die inneren Knoten, der Huffman-Baum und die Länge der Originalnachricht am Anfang der kodierten Nachricht geschrieben werden. Durch diese zusätzlichen Informationen bringt die Huffman-Codierung erst bei längeren Texten einen Vorteil.

5.2. Huffman-Decodierung

Aus dem Huffman-Baum am Anfang der kodierten Nachricht wird die für die Codierung genutzte Code-Tabelle rekonstruiert. Mit der Code-Tabelle kann die Originalnachricht wieder hergestellt werden.

Teil III.

Entwicklung

6. Huffman-Codierung

Bei der Entwicklung der Huffman-Codierung wurde die Huffman-Codierung aus der Vorlesung „Theoretische Informatik“ an der HAW Hamburg als Vorlage genommen.
Ein kritischer Punkt bei der Huffman-Codierung ist dabei das Sortieren der einzelnen Teilbäume nach ihrem Gewicht. Bei der Erstellung des Huffman-Baums werden in jedem Schritt die beiden Teilbäume mit dem geringsten Gewicht benötigt. Für diesen Zweck bietet sich eine Priority-Queue an. Diese nutzt für die Sortierung einen Binary Heap. Da die Programmiersprache C++ im Gegensatz zu C die Priority-Queue bereits als Datentyp anbietet, wurde die Umsetzung der Huffman-Codierung in C++ vorgenommen[1].
In Abbildung 6.1 wird der Ablauf der Huffman-Codierung, wie er in Kapitel 5 beschrieben wurde, gezeigt.

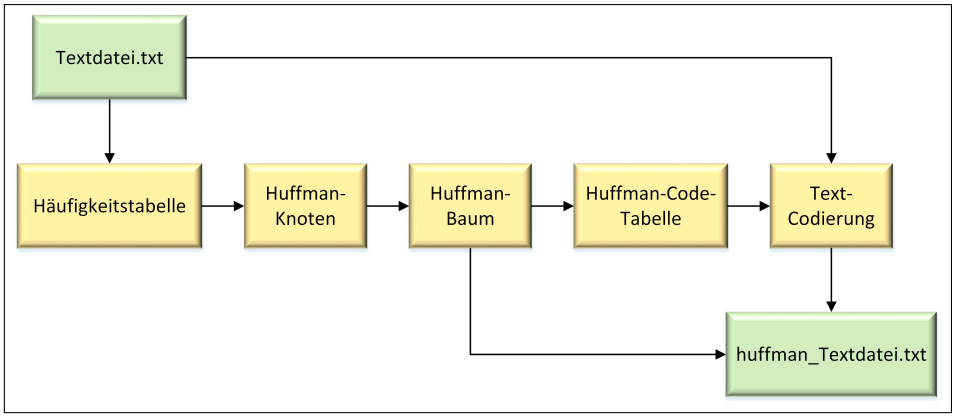


Abbildung 6.1.: Ablauf Huffman-Encodierung

Nach dem Erstellen der Häufigkeitstabelle werden die Huffman-Knoten, für die eine Struktur angelegt wurde, erstellt und in die Priority-Queue eingefügt. Dabei wird aus der Häufigkeitstabelle auch ein Zeichen mit der Häufigkeit 0 als Sonderzeichen für die internen Knoten gewählt. Da der erweiterte ASCII-Code Sonderzeichen aus vielen Bereichen, sowie Umlaute aus vielen Sprachen und zusätzlich Sonderzeichen, die in Texten nicht verwendet werden, enthält, ist die Wahrscheinlichkeit sehr gering, dass alle 256 Zeichen im Text vorkommen.

[1] Die beiden erstellten Programme „Huffman_Decoder.exe“ und „Huffman_Encoder.exe“ werden direkt aus dem C-Programm aufgerufen.

Daher wurde hier auf eine zusätzliche Behandlung vorerst verzichtet.
Dieses Sonderzeichen wird in die erste Zeile der neu erstellten Textdatei geschrieben(Abb. 6.2, rot markiert). In Zeile 2 folgt dann der Huffman-Baum(blau markiert), der mit zwei Zeilenumbrüchen abgeschlossen wird. Dann folgt eine Zeile mit der Anzahl der Zeichen der Originalnachricht(grün markiert) und ab der nächsten Zeile der gesamte codierte Text.

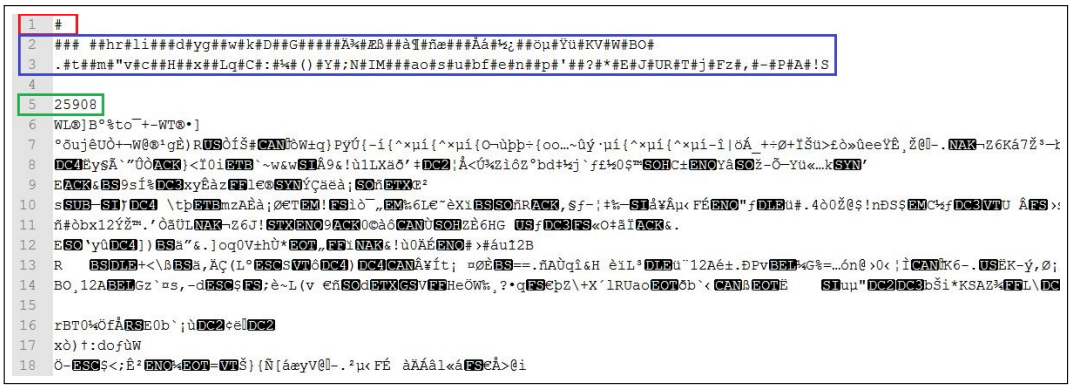


Abbildung 6.2.: Beginn einer Huffman-codierten Textdatei

6.1. Huffman-Decodierung

Bei der Decodierung wird der Prozess umgekehrt(Abb. 6.3). Es wird das Sonderzeichen, der Huffman-Baum und die Anzahl der Zeichen eingelesen. Aus dem Huffman-Baum wird die Code-Tabelle rekonstruiert. Der Text wird eingelesen und mit Hilfe der Code-Tabelle decodiert.

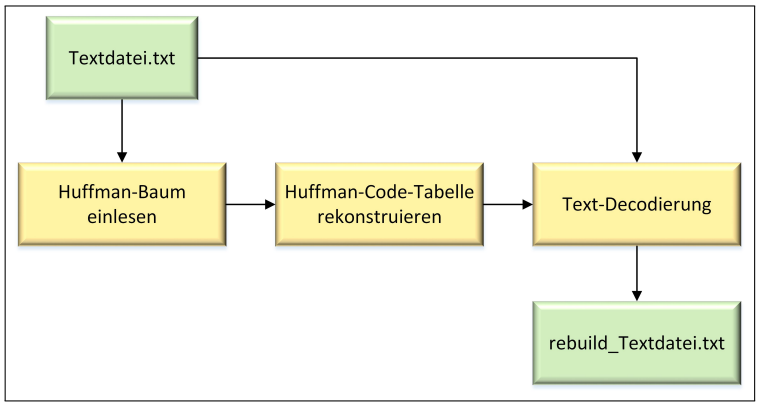


Abbildung 6.3.: Ablauf Huffman-Decodierung

7. Kryptogeraphie

Dieses Kapital beschäftigt sich mit der Implementierung des RSA-Verfahrens, das in Kapital 4.3 analysiert wurde.
Aufgrund des Faktorisierungsproblems erhöht sich die Sicherheit bei der RSA-Verschlüsselung mit der Länge der ausgewählten Primzahlen p und q. Nach dem heutigen Stand der Technik ist eine Schlüssellänge von 1024 oder 2048 Bit üblich. In dieser Arbeit werden p und q mit jeweils 512 Bit ausgewählt, so dass sich damit für einen privaten Schlüssel d eine Schlüssellänge von 1024 Bit.
Um diese großen Zahlen darstellen zu können und mit ihnen Operationen durchführen zu können, wird die FLINT-Bibliothek[1] benutzt.

Es gibt drei Aufgaben, die die Software in diesem Bereich erledigen muss:
- Für neue Benutzer, die keinen Schlüssel besitzen, alle benötigten Schlüssel erzeugen.
- Für Sender die Nachricht verschlüsseln.
- Für Empfänger die erhaltene verschlüsselte Nachricht entschlüsseln.

Die Tabelle 7.1 listet alle geschriebenen Funktionen auf, die die erwähnten Aufgaben übernehmen.

Name der Funktion	Beschreibung
keyGenerator()	Erzeugt zwei geheime Primzahlen p und q, den privaten Schlüssel d und den öffentlichen Schlüssel n.
primZahl()	Erzeugt eine Primzahl mit 512 Bit
squareAndMultiplay()	Verschlüsselungs- bzw. Entschlüsselungs-Funktion durch Berechnung einer diskreten Exponentialfunktion
rsaEncrypt()	Verschlüsselt die Nachricht
rsaDecrypt()	Entschlüsselt die Nachricht

Tabelle 7.1.: Liste der Funktionen im Bereich Kryptographie

[1] Fast Library for Number Theory (FLINT)

7.1. Schlüssel-Generator

Es wird der Blum-Blum-Shub-Generator (BBS-Generator) benutzt, um 512-Bit Zahlen zu generieren. Der Benutzer gibt seine individuelle Zahl im Bereich „unsigned Integer“ ein, um den Generator zu initialisieren. Im Bereich um die generierte Zahl wird dann die nächste Primzahl gesucht. So werden die geheimen Parameter p, q und $\phi(n)$, sowie der geheime Schlüssel d und der öffentliche Schlüssel n berechnet. Für den öffentlichen Schlüssel e wird immer die dritte Fermatsche Primzahl[2], die Zahl 17, genommen. Die erzeugten Schlüssel werden für die spätere Nutzung in Textdateien mit festgelegten Namen gespeichert(Siehe Abb. 7.1).

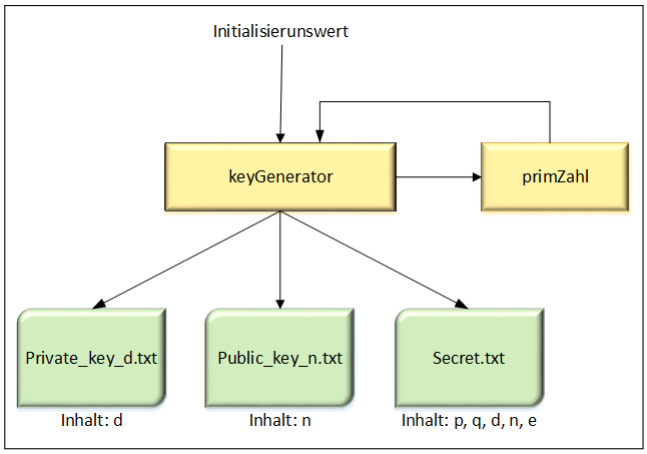


Abbildung 7.1.: Schlüssel-Generator

7.2. Verschlüsseln und Entschlüsseln

Damit die benutzte mathematische Formel im RSA-Verfahren korrekte Ergebnisse liefert, muss der Klartext kleiner als der öffentliche Schlüssel n sein. n ist eine 1024 Bit(128 Byte) große Zahl. Also wird die Nachricht in 127 Byte große Pakete aufgeteilt, die einzelnen Pakete verschlüsselt und dann wieder in einer Textdatei zusammengefügt(Siehe Abb. 7.2). Dabei wird hinter jedem 0xff-Byte ein Stuff-Byte 0x00 eingefügt. Am Ende von jedem Paket wird ein zwei Byte größer „End of Paket“-Marker 0xff 0xff hinzugefügt. Durch die eingefügten Stuff-Bytes ist dieser Marker eindeutig zu identifizieren und die einzelnen Pakete können

[2]Die Fermatschen Primzahlen werden aufgrund ihrer speziellen Darstellung im dualen System genommen. Dadurch vereinfacht sich das Rechnen mit dem Square-and-Multiplay-Verfahren.
z. B. 17 = 1 0001, 257 = 1 0000 0001

beim Entschlüsseln erkannt werden. Die Anzahl der Pakete(1 Byte) sowie die Länge der verschlüsselten Nachricht(4 Byte) wird am Anfang der verschlüsselten Nachricht hinzugefügt(Siehe Abb. 7.3).
Bei der Entschlüsselung werden die Pakete identifiziert, einzeln entschlüsselt und dann wieder zusammengefügt, so dass sich die Original-Nachricht ergibt.

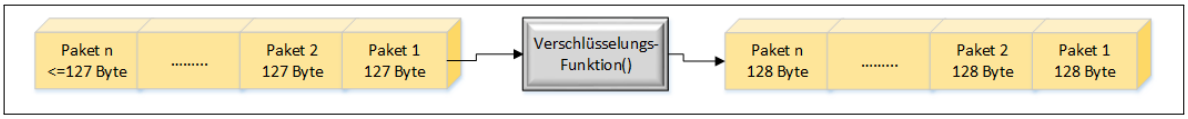


Abbildung 7.2.: Vor der Verschlüsselung sind die Pakete 127 Byte groß, das letzte Paket kann zwischen 1 und 127 Byte groß sein. Nach der Verschlüsselung sind alle Pakete 128 Byte groß

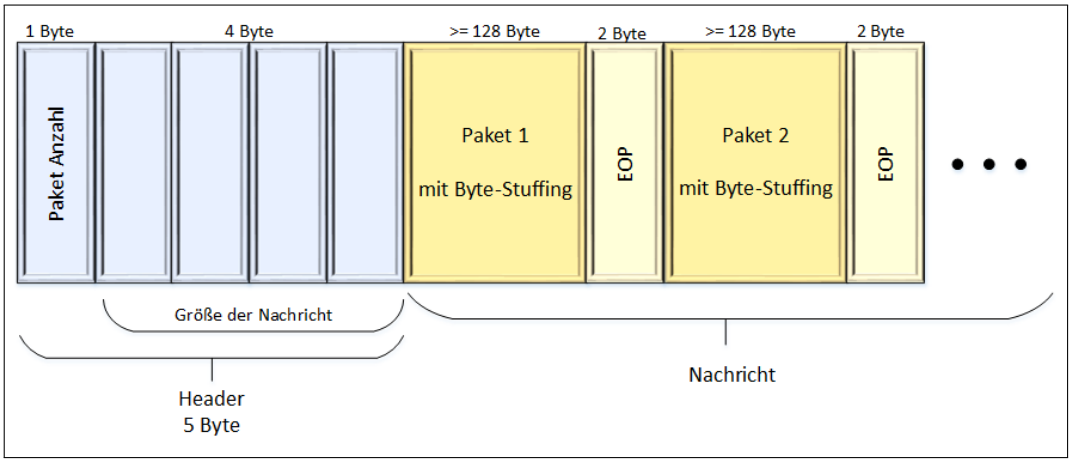


Abbildung 7.3.: Die gesamte Nachricht nach der RSA-Kodierung

Anmerkung
Die Größe der Nachricht muss mitgeschickt werden, da sie beim Extrahieren der Nachricht vom Decoder benötigt wird, weil die Pakete durch das Byte-Stuffing verschiedene Größen haben können. Daher reicht die Anzahl der Pakete alleine dem Decoder nicht, um zu wissen wie viele Koeffizienten manipuliert worden sind. Die Anzahl der Pakete wird für die RSA-Entschlüsselungs-Funktion genutzt.

8. Steganographie

Um ein Bild im JPEG-Format als Container für Steganographie nutzen zu können, muss dafür gesorgt werden, dass die eingebetteten Daten wieder unverändert aus dem Container-Bild extrahiert werden können.
In einem Bildformat wie z. B. PNG werden die Bilddaten mit einer verlustfreien Kompressions-Methode komprimiert. Dies ermöglicht es, die Pixelwerte durch einen Algorithmus zu manipulieren und den gleichen Wert bei der Decodierung zu rekonstruieren. So kann z. B. in jedem Farbkanal das LSB mit einem Nachrichten-Bit überschrieben werden. Wie effizient ausgewählte steganographische Algorithmen sind und wie stark die Merkmale und statistischen Eigenschaften eines Bildes dadurch verändert werden und damit das Stego angreifbar machen, muss für jeden Algorithmus genau untersucht werden.
Durch den wesentlich komplexeren Aufbau und die verlustbehaftete Kompression des JPEG-Formats ist diese steganographische Methode nicht anwendbar. Die Pixelwerte, die aus dem JPEG-Decoder kommen, sind nicht exakt die gleichen Werte, die in den Encoder gegeben werden und daher würde auch eine versteckte Nachricht zerstört werden.
Was sich durch die Codierung und Decodierung der Bilder nicht ändert, sind die Koeffizienten, die aus den MCU-Blöcken gerechnet werden. Diese Koeffizienten können nach der Quantisierung manipuliert werden.

Option1: .png / .bmp to .jpg
Ein Möglichkeit wäre, einen JPEG-Encoder zu implementieren. So könnte ein Bild aus einem anderen Format, wie z. B. „.png“ oder „.bmp“, durch diesen Encoder in das JPEG-Format umgewandelt werden und dabei die geheimen Daten eingebettet werden(Abb. 8.1).

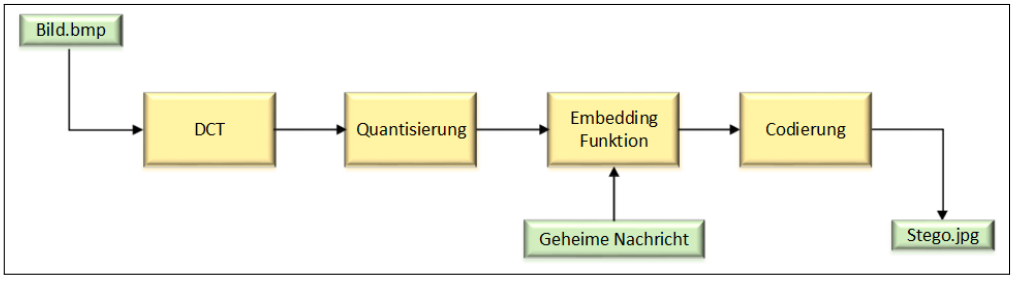


Abbildung 8.1.: JPEG-Encoder mit Manipulationsfunktion

Vorteile
Es gibt viele Bibliotheken wie z. B. OpenCV oder Software wie z. B. Matlab, die viele Funktionen im Bereich der Bildverarbeitung anbieten. Hier können sehr einfach Farbkonvertierungen oder auch die diskrete Kosinus-Transformation berechnet werden.

Nachteile
Einen kompletten JPEG-Encoder zu implementieren ist sehr aufwändig und selbst wenn es einem gelingt, werden die kodierten Bilder nicht die gleichen Merkmale besitzen, wie die aus einem Standard JPEG-Encoder. Das könnte es einem Angreifer leichter machen, das Stego zu identifizieren. Außerdem wären große Kenntnisse über die Wahrnehmungsfähigkeiten des Menschen notwendig und bis zu welchem Grad diese ausgenutzt werden können.

Option2: .jpg to .jpg
Die zweite Möglichkeit wäre, ein Bild zu nutzen, dass bereits im JPEG-Format vorliegt. Hier müssen zuerst die enthaltenen Koeffizienten extrahiert und decodiert werden. So können die Koeffizienten durch einen geeigneten Algorithmus mit Nachrichten-Bits manipuliert werden(Abb. 8.2).

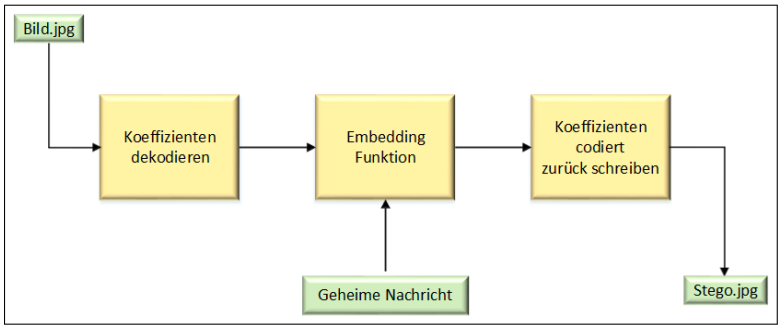


Abbildung 8.2.: JPEG-Decoder mit Manipulationsfunktion

Vorteile
Dieses Verfahren bringt zwei Vorteile mit sich. Zum einen ist es wesentlich weniger fehleranfällig, die Daten aus einer komprimierten Datei zu extrahieren, als einen kompletten JPEG-Encoder zu implementieren.
Zum anderen bleibt das Bild, das aus einem Standard-Encoder stammt, an vielen Stellen unverändert und somit ist der Verdacht auf Manipulation geringer.

Nachteile
Die oben genannten Hilfsmittel[1] können nicht zum Herstellen des Stegos verwendet, da

[1] Software und Bibliotheken im Bereich Bildverarbeitung

diese keine Funktionen anbieten, die die Koeffizienten aus dem Bild extrahieren können.

In dieser Arbeit wird die zweite Methode verwendet. So werden Funktionen implementiert, um die Koeffizienten aus dem Bild zu extrahieren, zu decodieren, zu manipulieren und wieder an dieselbe Stelle zurückzuschreiben(Tabelle 8.1). Diese Aufgaben übernimmt der **Encoder** und wandelt so ein Bild in ein Steganogramm um.
Als zweites wird ein **Decoder** implementiert, um die Nachricht wieder aus dem Steganogramm holen zu können.

Es muss überlegt werden, welche Koeffizienten überschrieben werden sollen und wie dies geschehen soll. Die Koeffizienten sind in die zwei Kategorien DC und AC geteilt. Einerseits haben diese beiden Kategorien unterschiedlich gewichteten Einfluss auf das Bild. Andererseits können diese Koeffizienten sehr verschiedene Werte annehmen. Ein Bild mit sehr vielen Farbmustern und schnellen Farbwechseln besitzt viele AC Koeffizienten, die nicht die Werte 0, 1 und -1 aufweisen. Dagegen haben eintönige Bilder, wie z. B. das Bild eines Apfels vor einem einfarbigen Hintergrund, sehr viele Koeffizienten mit den Werten 0, 1 und -1.
Ein anderer Punkt ist, dass die Koeffizienten VLI(Variable Length Integer) kodiert sind. Daher müssen die Koeffizienten ihre Bitlänge beibehalten, damit die Huffman-Tabellen unverändert bleiben können. Dies verschafft Vorteile, was die Erkennung der Original-Koeffizienten im Stego angeht.

Diese Arbeit bietet vier verschiedene Auswahlmöglichkeiten, bei welchen der Koeffizienten das LSB überschrieben werden soll:

- **NN** Alle Koeffizienten, die nicht den Wert 0 haben. Diese Methode bietet die größte Kapazität an
- **N3** Alle Koeffizienten, die nicht den Wert -1, 0 und 1 haben. Diese Methode sortiert zusätzlich zur ersten Methode die Werte 1 und -1 aus. Diese Werte haben in der VLI-Kodierung eine Länge von nur einem Bit und so ändert sich bei einer Änderung des LSB(einziges Bit) der Wert von 1 auf -1 bzw. von -1 auf 1, also ein Vorzeichenwechsel und eine Wertänderung um 2.[2]
- **NN-AC** Wie erste Methode(NN), zusätzlich werden die DC-Koeffizienten nicht verändert
- **N3-AC** wie zweite Methode(N3), zusätzlich werden die DC-Koeffizienten nicht verändert

[2] Bei allen anderen Bitlängen ändert sich der Wert bei Änderung des LSB nur um 1:
Länge 2: $-3(00) \rightleftarrows -2(01)$ und $3(11) \rightleftarrows 2(10)$
Länge 3: $-7(000) \rightleftarrows -6(001)$, $-5(010) \rightleftarrows -4(011)$ und $7(111) \rightleftarrows 6(110)$, $5(101) \rightleftarrows 4(100)$

Der Encoder und auch der Decoder bestehen aus vielen Funktionen, die jeweils mit einer kurzen Beschreibung in der Tabelle 8.1 am Ende des Kapitels aufgelistet sind.

8.1. Realisierung Encoder

Die Funktionen des Encoders werden alle Informationen, die in der JPEG-Datei enthalten sind (siehe Kapitel 3), zur Decodierung der komprimierten Koeffizienten, extrahieren. Für einen Überblick über die Reihenfolge der Funktions-Aufrufe sorgen die Abbildungen 8.3, 8.4 und 8.5.

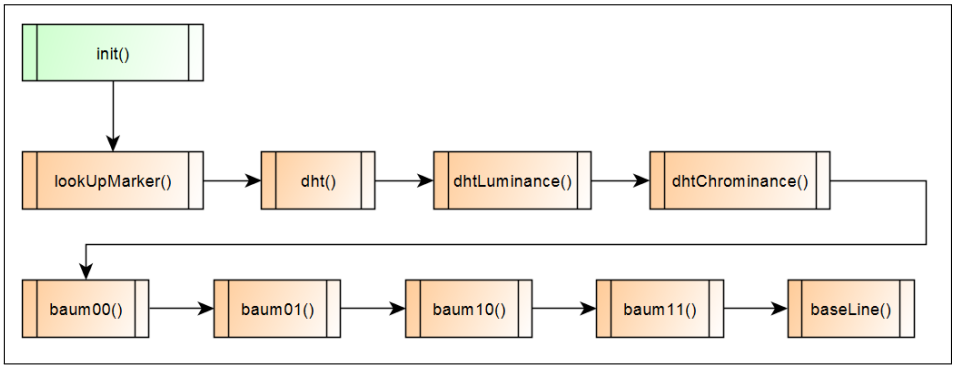


Abbildung 8.3.: Beim Aufruf der init()-Funktion werden weitere neun Funktionen durchgeführt.(Ihre jeweilige Aufgabe ist in Tabelle 8.1 zu finden)

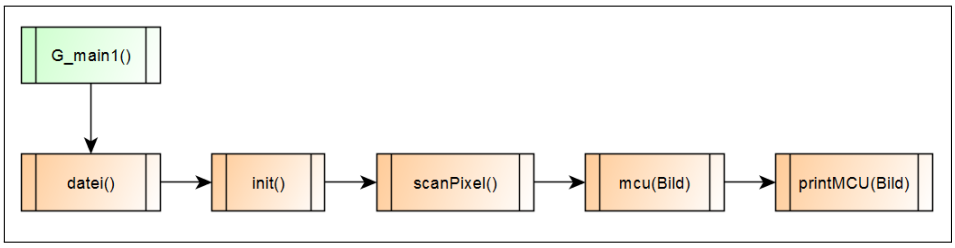


Abbildung 8.4.: G_main1() ruft weitere fünf Funktionen in der richtigen Reihenfolge auf, um die Koeffizienten in zwei Kategorien, DC und AC, zu speichern

Bis hierher wurden alle Koeffizienten mit ihren Eigenschaften, Werten und der Position ihres LSB aus der Bilddatei extrahiert und zur späteren Verwendung gespeichert. Dadurch ist das Einbetten der geheimen Nachricht in der Funktion stego(), die aus der GUI gestartet wird, dann relativ einfach geworden. Nachdem das Stego erstellt wurde, werden die Funktionen in Abb. 8.5 aufgerufen, um Informationen und Vergleichsdaten zu Originalbild und Stego in Textdateien zu schreiben.

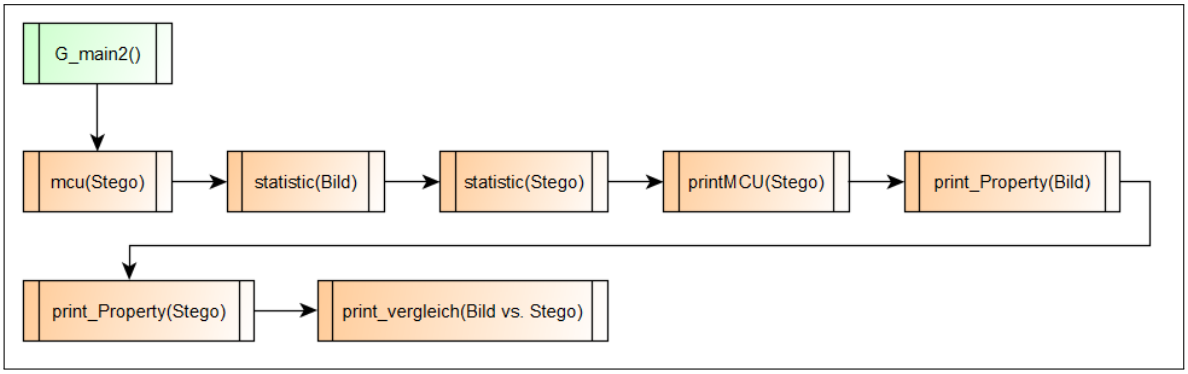


Abbildung 8.5.: G_main2()-Funktion und die aufgerufenen Funktionen

In Abb. 8.6 ist ein Ablauf-Diagramm zur Herstellung eines Steganogramms dargestellt.

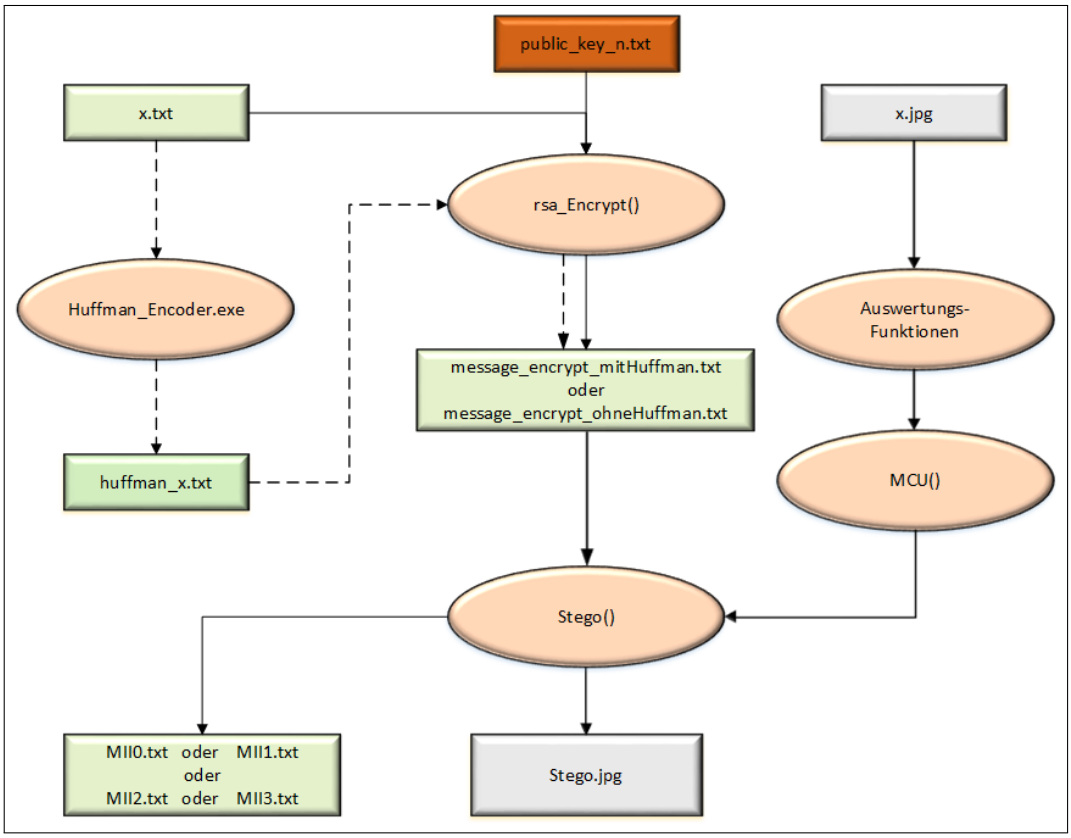


Abbildung 8.6.: Ablaufplan zur Herstellung eines Steganogramms

8.2. Realisierung Decoder

Damit der Decoder erkennen kann, welche der vier oben genannten Methoden verwendet wurde, wird der Header der Nachricht um zwei Byte erweitert (Abb. 8.7, zum Vergleich siehe Abb. 7.3). Das erste Byte kennzeichnet, welche der vier Methoden benutzt wurde. Das zweite Byte kennzeichnet, ob die Nachricht zusätzlich Huffman-codiert ist oder nicht.

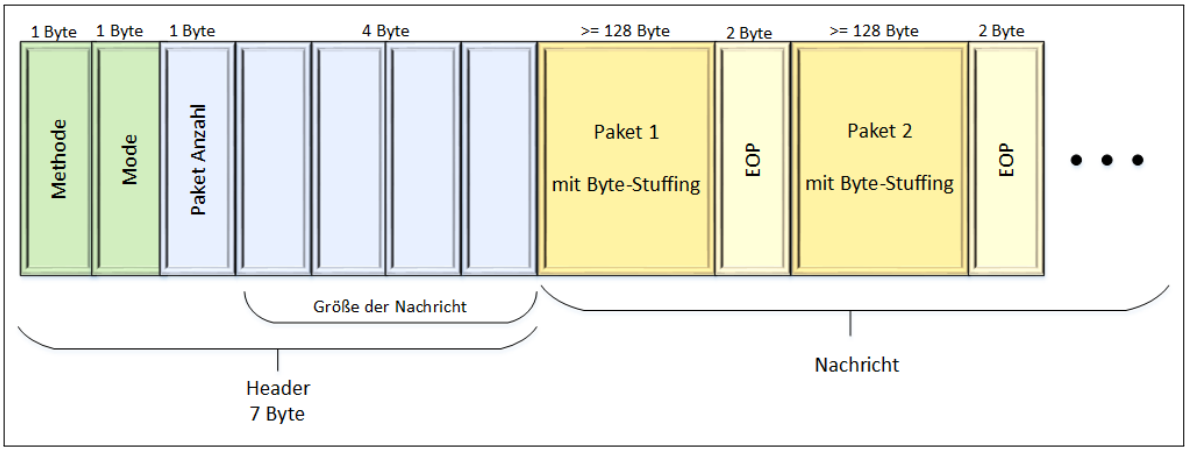


Abbildung 8.7.: Die gesamte Datei (Header und verschlüsselte Nachricht), die im Bild eingebettet wird

Das Extrahieren der eventuellen Nachricht aus dem Steganogramm durch den Decoder läuft nach Abb. 8.8. Dabei werden nacheinander für die vier Kategorien jeweils 56 Bit aus den entsprechenden ersten 56 Koeffizienten geholt und geprüft, ob der sieben Byte Header erkannt wird.
Ist dies der Fall wird die Größe der Nachricht ausgelesen und die entsprechende Anzahl Bytes aus den Koeffizienten ausgelesen. Die Stuff-Bytes werden entfernt und die einzelnen Pakete entschlüsselt und zusammengesetzt. Zum Abschluss wird die Nachricht ggf. noch Huffman-decodiert.
In Abb. 8.9 ist ein Ablauf-Diagramm zum Extrahieren der Nachricht aus dem Steganogramm dargestellt.

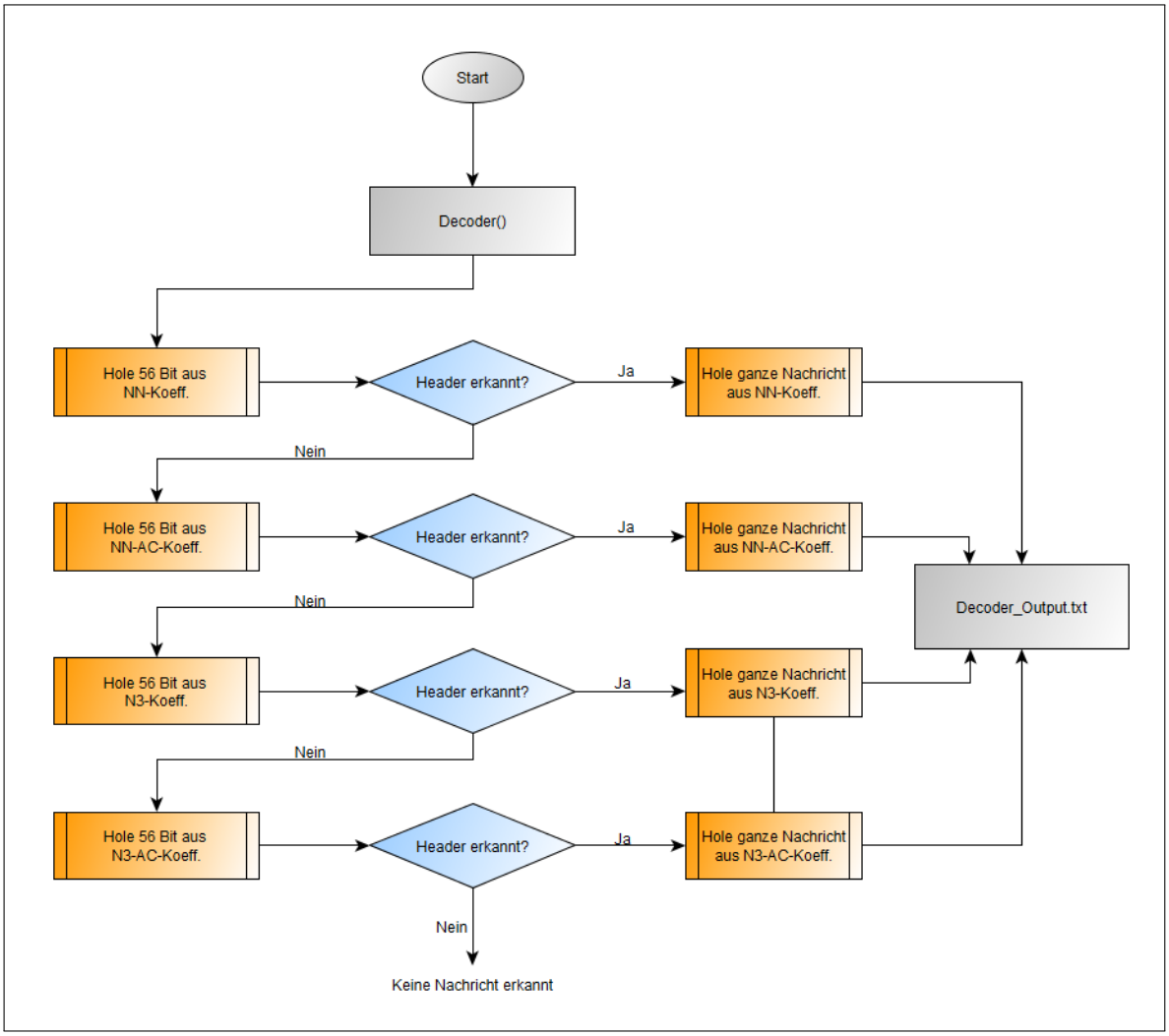


Abbildung 8.8.: Steganogramm-Decoder: Header-Erkennung

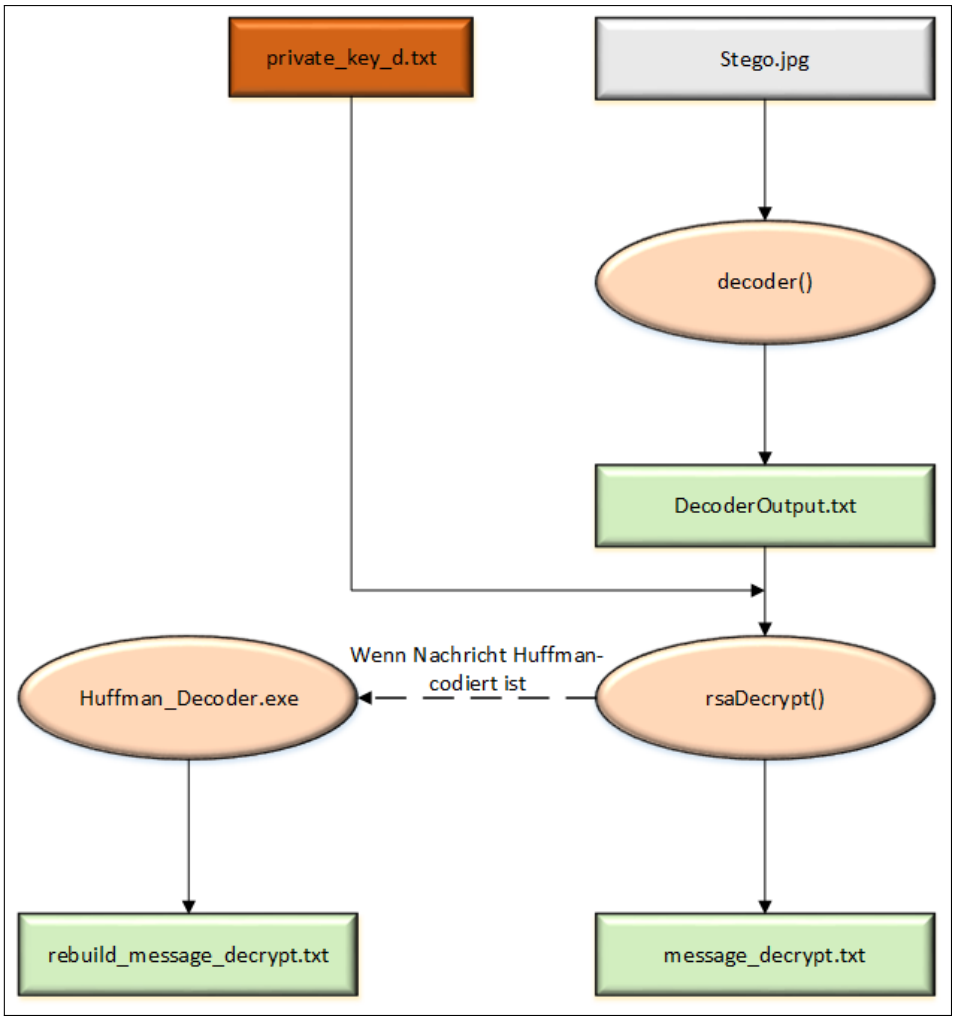


Abbildung 8.9.: Ablaufplan zum Extrahieren einer eventuellen Nachricht aus dem Steganogramm

Die Liste aller Funktionen:

Name der Funktion	Beschreibung
lookUpMarker()	Sucht alle Marker raus, legt diese mit Namen und Adressen in einer Tabelle ab.
dht()	Liest die Huffman-Tabellen aus, speichert ihre Adresse, Header-Größe, Anzahl der gesamten Codes, Anzahl der Codes pro Bitlänge und die Codes.
baum00() baum01() baum10() baum11()	Rekonstruiert Huffman-Bäume und speichert die Bitmuster für jeden Code. Insgesamt vier Lookup-Tabellen mit Bitlänge, Bitmuster und Code werden gespeichert.
dqtLuminance() dqtChrominance()	Die zwei Quantisierungs-Tabellen werden in zwei 8x8 Matrizen abgelegt.
scanPixel()	Filtert die Stuff-Bytes raus und speichert die Bilddaten(aus dem SOS-Abschnitt) ohne Stuff-Bytes neu. Ermittelt die Größe der Bild-Daten in Byte.
baseLine()	Ermittelt das Bildformat(BxH), die Art des Subsamplings und berechnet damit die Anzahl der MCU, sowie der DC- und AC-Koeffizienten.
wertDecoder()	Rechnet eine Bitfolge nach VLI-Codierung in einen signed Integer-Wert um.
header()	Ermittelt für jeden Marker die Größe des Abschnitts.
adresse()	Holt die Marker-Adresse aus der Marker-Lookup-Tabelle.
datei()	Liest das Bild in den Speicher ein und gibt den Zeiger zurück
mcu()	- Gehört zu den Haupt-Funktionen. - Dekodiert die Koeffizienten aus dem Bitstrom - Sortiert diese nach ihren Werten in verschiedene Arrays - Speichert die Bit-Positionen bzw. Byte-Position für jeden Koeffizienten für späteren Zugriff - Ermittelt die Anzahl der NN-, N3-, ACNN-, ACN3-Koeffizienten

Tabelle 8.1.: Funktionsliste(1.Teil) der Steganographie

Name der Funktion	Beschreibung
histogramm()	Berechnet Histogramm-Werte und schreibt diese in eine csv-Datei.
statistic()	Berechnet die Anzahl der 1- und -1-Koeffizienten, sowie Minimum, Maximum, Durchschnitt, Summe und Anzahl der Koeffizienten, die nicht -1, 0 und 1 sind.
mp0() mp1() mp2() mp3()	- Gehört zu den Haupt-Funktionen - mp0(): Überschreibt die LSB der **NN-Koeffizienten** mit den Nachrichten-Bits - mp1(): Überschreibt die LSB der **N3-Koeffizienten** mit den Nachrichten-Bits - mp2(): Überschreibt die LSB der **NN-AC-Koeffizienten** mit den Nachrichten-Bits - mp3(): Überschreibt die LSB der **N3-AC-Koeffizienten** mit den Nachrichten-Bits
stego()	Haupt-Funktion. Tauscht die Bilddaten gegen die manipulierten Daten und erstellt das Stego-Bild.
Decoder()	Haupt-Funktion, inverse Funktion von stego(). Header auslesen, ja nach Inhalt des Headers, werden die betroffenen Koeffizienten aus dem Stego extrahiert und zur Decodierung bereit gestellt.

Tabelle 8.2.: Funktionsliste(2.Teil) der Steganographie

Name der Funktion	Beschreibung
printMCU()	Ausgabe-Funktion, schreibt Informationen über Koeffizienten in Textdatei
print_property()	Ausgabe-Funktion, schreibt verschiedene Statistik-Zahlen in Textdatei
print_vergleich()	Ausgabe-Funktion, schreibt Vergleich der Statistik-Werte von Bild und Stego in Textdatei
printMatrixShortDC() printMatrixShortAC() printMatrixChar() printMatrixInt() printCoeffizient()	Ausgabe-Funktionen: Schreiben verschiedene Matrizen in Textdateien

Tabelle 8.3.: Liste der Ausgabe-Funktionen

9. GUI - Graphical User Interface

Um die Untersuchung von JPEG-Bildern bzw. die Durchführung von kryptographischen und steganographischen Verfahren für die Benutzer zu vereinfachen, wird eine grafische Benutzeroberfläche implementiert. Dies ermöglicht es dem Benutzer, mit dieser Sofware mit wenigen Klicks Steganogramme zu erstellen. Zudem können empfangene Nachrichten aus Stegos dekodiert werden.

Es wird eine Windows API[1] zur Entwicklung der GUI benutzt, da die meisten Programmteile in der Sprache C geschrieben worden sind. Die Nutzung von Bibliotheken wie z. B. QT, die mit objektorientierten Sprachen wie C++ zu nutzen sind, ist daher nur schwer möglich.

Die GUI besteht aus fünf Fenstern(Abbildungen und weitere Informationen folgen in der Anleitung in Kapitel 14):

- **jpgStego**(Hauptfenster)
  Hier wird je nach Vorhaben das entsprechende Fenster ausgewählt und geöffnet.
- **Encrypt**
  In diesem Fenster werden das gewünschte Bild und der Text zur Herstellung des Steganogramms ausgewählt. Das Fenster gibt viele Informationen über das Bild und seine Koeffizienten aus.
- **Decrypt**
  Es wird ein Steganogramm zum decodieren und extrahieren der Nachricht ausgewählt.
- **Key generator**
  Ermöglicht dem Benutzer, eigene kryptographische Schlüssel zu erzeugen.
- **JPG-Info**
  Hier können Bild-Informationen ausgewählt werden, die während des Encrypt-Verfahrens in Dateien gespeichert werden.

[1] Windows application programming interfaces (WinAPI)

In der Tabelle 9.1 sind alle Funktionen zur Realisierung der GUI jeweils mit einer kurzen Beschreibung aufgelistet.

Name der Funktion	Beschreibung
InitApplication()	Initialisiert und registriert die Fenster-Klasse
InitInstance()	Erzeugt das Hauptfenster und zeigt dieses an
WinMain()	Ablaufsteuerung des Hauptfensters
WndProc()	Kontrolliert das Hauptfenster
DlgProcEncrypt()	Ablaufsteuerung zur Erstellung des Steganogramms
DlgProcDecrypt()	Ablaufsteuerung zum Extrahieren der Nachricht aus dem Stego
DlgProcJPG()	Ablaufsteuerung zur Erstellung von .txt und .csv Dateien mit Informationen über Bild bzw. Steganogramm
DlgProcKey()	Ablaufsteuerung zur Erstellung der kryptographischen Schlüssel
Check()	Gibt die im JPG-Info-Fenster ausgewählten Informationen zurück z. B. verschiedene Koeffizienten, Histogramme, Huffman-Bäume, Huffman-Tabellen, . . .
CheckMethode()	Erkennung, welche der vier Methoden ausgewählt wurde
CheckMode()	Erkennung, ob Text vorab Huffman-Kodiert werden soll
MainFunktion()	Prüft, ob das Bild mit der ausgewählten Methode genug Kapazität für den Text bietet. Wenn ja, erfolgt der Funktionsaufruf zur Herstellung des Steganogramms

Tabelle 9.1.: Liste der Funktionen der GUI

10. Webserver auf Raspberry Pi zur Übermittlung der Bilddateien

Da in der heutigen Zeit alle Daten, die über das Internet übertragen werden oder auf fremden Servern hinterlegt werden, langfristig gespeichert werden, sollen die Bilder mit den Nachrichten ausschließlich auf eigenen Medien gespeichert werden. Um die Bilder dennoch übertragen zu können, soll ein eigener Webserver auf einem Raspberry Pi eingerichtet werden. Somit können die Bilder kurzfristig auf einer Webseite unter einer eigenen IP-Adresse zum Download angeboten werden.

10.1. Raspberry Pi

Der Raspberry Pi ist ein Microcontroller mit einem ARM-Mikroprozessor. Bei dem in dieser Arbeit eingesetztem Model handelt es sich um den Raspberry Pi 2 Model B mit einer ARM Cortex-A7-CPU mit 4 Kernen und einem Takt von 900MHz. Dieses Model verfügt über einen Arbeitsspeicher von 1024 MB und einen microSD-Kartenleser. Es wird das Betriebssystem Raspian, das auf dem Linux-Betriebssystem Debian basiert, genutzt. Das Betriebssystem befindet sich auf einer micro-SD-Karte.

10.2. Programmiersprache Python

Bei Python handelt es sich um eine höhere Programmiersprache, die meist über einen Interpreter ausgeführt wird. Während ein Compiler das komplette Programm in Maschinensprache übersetzt und erst anschließend ausführt, liest der Interpreter ein Programm zeilenweise ein, übersetzt die einzelnen Befehle und führt diese direkt aus. Der Interpreter kann entweder im Kommandozeilen-Modus direkt einzelne Befehle ausführen oder im Script-Modus den Inhalt einer Datei. So lassen sich kleine Programme schnell programmieren und testen.

Ein Nachteil der Interpreter-Programmiersprachen ist aber die niedrigere Ausführungsgeschwindigkeit der Programme, da die selben Programmteile immer wieder neu übersetzt werden. Der Compiler hingegen übersetzt das ganze Programm auf einmal und kann dabei Optimierungen vornehmen. Dies ist aber erst bei größeren Programmen ein Vorteil und wirkt sich für das kleine Webserver-Programm in dieser Arbeit nicht negativ aus

10.3. Webframework Flask

Flask ist ein Webframework, das in Python geschrieben ist. Mit Hilfe von Flask kann ein Webserver auf dem Rapsberry Pi gestartet werden und über Python-Programme können Daten dynamisch an HTML-Templates übergeben werden.
So kann eine dynamische HTML-Webseite auf dem Webserver erstellt werden, die alle Bilder mit Bildinformationen aus einem vorgegebenen Verzeichnis darstellt.

10.4. Webserver

Um den Webserver zu erstellen, wird die in Abb. 10.1 dargestellte Ordnerstruktur auf der micro-SD-Karte des Raspberry Pi erstellt.

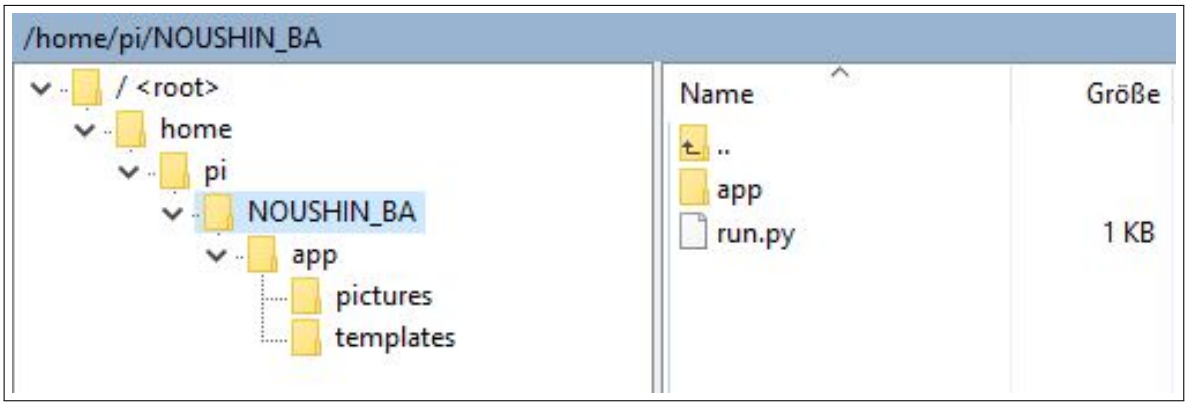


Abbildung 10.1.: Verzeichnisstruktur auf dem Raspberry Pi

Das Python-Skript *run.py* enthält nur 2 Zeilen:

```
from app import app
app.run(host='0.0.0.0', port=5000, debug=False)
```

Hier wird das *app*-Package importiert und die Web-Applikation auf dem Port 5000 gestartet. Der host-Eintrag '0.0.0.0' bedeutet, dass alle Geräte auf die Web-Applikation zugreifen dürfen. Mit dem debug-Modus ist es während der Entwicklung möglich, Fehlermeldungen des Programms beim Aufruf direkt auf der aufgerufenen Webseite anzeigen zu lassen.

Im Unterordner *app* befinden sich die Python-Skripte *__init__.py* und *views.py*.
In der *__init__.py* wird die Applikation initialisiert:

```
from flask import Flask
app = Flask(__name__, static_folder='pictures', static_url_path='/pictures')
from app import views
```

Hier wird das Flask-Package geladen und die Applikation als Flask-App mit dem Ordner für statische Objekte, wie z. B. Bilder, initialisiert. Zudem wird das views-Package importiert, so dass die Ansichten der verschiedenen Unterseiten dann in der Datei *views.py* definiert werden können.
Hierzu wird in der Datei *views.py* der folgende Befehl genutzt:

```
@app.route('/')
@app.route('/mohammadi/noushin')
def index():
```

Hier wird definiert, dass bei Aufruf der Hauptseite('/') und bei Aufruf der Unterseite('/mohammadi/noushin') die Funktion *index()* aufgerufen wird.
In der Funktion *index()* wird der Unterordner *pictures* nach JPEG-Dateien durchsucht und die gefundenen Dateinamen und Dateiinformationen in Listen gespeichert. Mit dem Befehl *render_template* wird das HTML-Template *index.html* aus dem Ordner *templates* geladen und diesem die Listen als Parameter übergeben:

```
return render_template('index.html',
                       title ='Bachelor Thesis Noushin Mohammadi',
                       time = timeString,
                       pictureList = dirList,
                       widthList = widthList,
                       heightList = heightList,
                       wList = wList,
                       imageSizeList = imageSizeList)
```

In der HTML-Datei ist es jetzt möglich, auf die übergebenen Variablen zuzugreifen und

die Listen mit for-Schleifen zu durchlaufen, wobei der $loop.index$ automatisch zur Verfügung gestellt wird:

```
{% for pic in pictureList %}
    <tr>
        <td>Format: {{ widthList[loop.index - 1] }} x {{ heightList[loop.index - 1] }}, </td>
    </tr>
{% endfor %}
```

Das Ergebnis wird dann von der Funktion zurückgegeben und die erstellte Seite im Browser angezeigt.
So ist es möglich die Bilddatei mit der versteckten und verschlüsselten Nachricht zwischen vielen anderen Bildern auf einer temporären Webseite anzuzeigen, ohne das die Bilddatei jemals auf fremden Medien gespeichert werden muss.
Alle JPEG-Bilddateien, die auf der Webseite angezeigt werde sollen, müssen im Ordner „pictures“ liegen. Diese können am sichersten mit einem USB-Stick oder mit den Programmen „Putty“ oder „WinSCP“ übertragen werden.

10.5. Raspberry Pi im www

Der Raspberry Pi wird per LAN-Kabel an den Router im Heimnetzwerk angeschlossen und bekommt dort eine feste IP-Adresse zugewiesen(Abb. 10.2). Am Router kann jetzt „Port Forwarding“ für den genutzten Port 5000 des Raspberry Pi eingestellt werden, so dass auch Geräte außerhalb des Heimnetzwerks auf die erstellte Webseite zugreifen können. Zusätzlich kann hier auch der Port 22 freigegeben werden, um über das SSH-Protokoll[1] von außen auf den Raspberry Pi zugreifen zu können(Abb. 10.3). Zusätzlich wird die externe IP-Adresse des Routers benötigt, über die der Router mit dem Internetserviceanbieter verbunden ist(Abb. 10.4).
Diese ändert sich bei jeder neuen Einwahl des Routers, also in der Regel alle 24 Stunden. Der Aufruf der Webseite erfolgt über die externe IP-Adresse und den Port 5000, also für den Zeitpunkt als diese Abbildungen erstellt wurden:

78.54.8.232:5000

Somit ist die Webseite nur temporär erreichbar und kann durch das Deaktivieren des „Port Forwarding“ schnell wieder deaktiviert werden.

[1] Secure Shell: Netzwerkprotokoll für verschlüsselten Zugriff auf entfernte Geräte

Abbildung 10.2.: Feste IP-Adresse für Raspberry Pi

Abbildung 10.3.: Port Forwarding für Raspberry Pi

Abbildung 10.4.: Externe IP-Adresse des Routers

Teil IV.

Auswertung

11. Testfälle

In diesem Kapitel werden Steganogramme, die durch die vier verschiedenen Methoden erzeugt wurden, verglichen, um die Qualität der Steganogramme zu überprüfen. Ein gutes Steganogramm muss sowohl visuelle Angriffe als auch statistische Angriffe überstehen können.
Bei visuellen Angriffen werden die Bilder mit verschiedenen Filtern untersucht. Bei statistischen Angriffen werden statistische Merkmale eines Bildes untersucht, wie z. B. Symmetrie.

Nach den folgenden Kriterien wird untersucht:

- Wie sehen die Histogramme von Bild und Steganogramm im Vergleich aus?
- Wie viele Änderungen werden durch die verschiedenen Methoden im Histogramm verursacht?
- Wie unterscheidet sich das Aussehen von Bild und Steganogramm nach Filterung?[1]
- Wie ändert sich die Symmetrie?
- Vergleich der Durchschnittswerte

11.1. Fall 1: Vergleich der Histogramme

Für diesen Test wird das Bild aus Abb. 11.1 genutzt. Von den vier Steganogrammen ist nur eins dargestellt, da sie optisch nicht zu unterscheiden sind. Bei allen vier Methoden wird die Kapazität voll ausgenutzt. In Tabelle 11.2 sind die Histogramme für die vier Methoden jeweils mit dem Histogramm des Originalbildes dargestellt. Die jeweils dunkleren Farbtöne stellen das Histogramm des Originalbildes dar. Unter den Histogrammen sind die Histogramm-Vergleiche abgebildet.

[1] Die genutzte Filter-Funktion:
Wenn das LSB des Pixels gleich 0 ist, wird der Pixelwert mit 0 überschrieben,
wenn das LSB des Pixels gleich 1 ist, wird der Pixelwert mit 255 überschrieben(bei 8 Bit Farbtiefe),
Ergebnis: Wenn das Bild nicht manipuliert ist, muss sein Muster immer noch erkennbar sein. [Master Arbeit]

Bild	Steganogramm
	

Tabelle 11.1.: Bild vs. Steganogramm

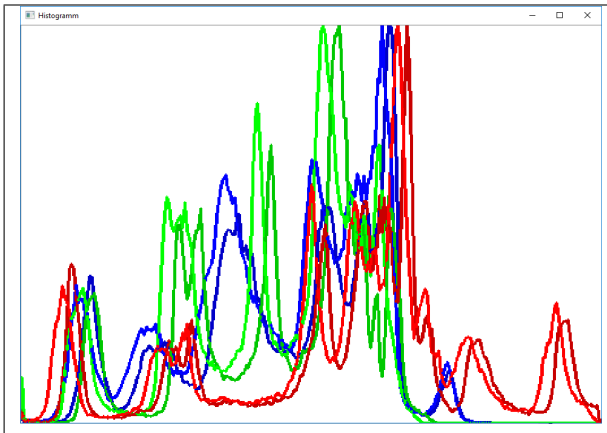
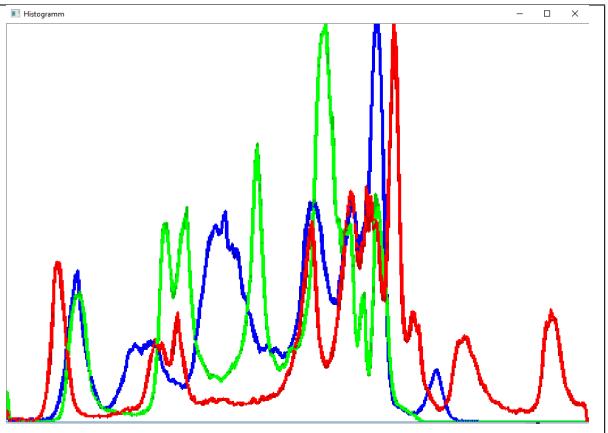
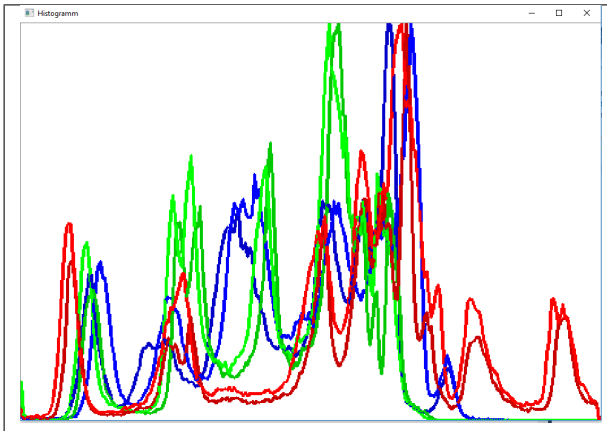
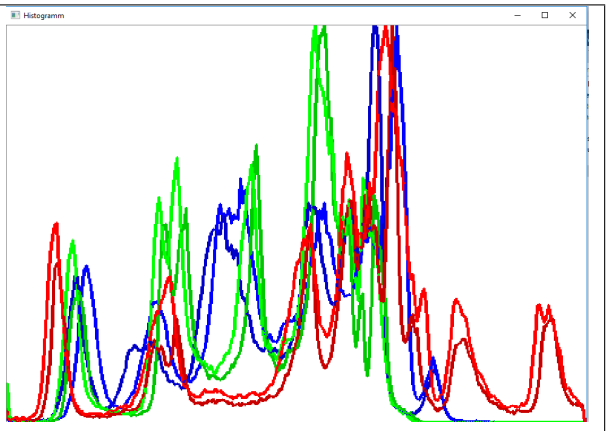
Methode NN	Methode AC-NN
	
Blauer Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.869932 Gruener Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.691538 Roter Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.700200 z.B. blauer Kanal mit sich selbst: 1.000000	Blauer Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.999368 Gruener Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.999424 Roter Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.999284 z.B. blauer Kanal mit sich selbst: 1.000000
Methode N3	**Methode AC-N3**
	
Blauer Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.618221 Gruener Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.883317 Roter Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.887446	Blauer Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.618855 Gruener Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.883965 Roter Kanal: Histogramm-Vergleich zwichen Bild und Stego: 0.887607

Tabelle 11.2.: Bild-Histogramm vs. Steganogramm-Histogramm

In der Tabelle 11.2 ist zu sehen, dass die Unterschiede zwischen den vier Methoden recht groß sind und nicht in allen Fällen mit den Erwartungen übereinstimmen. So ist das Ergebnis bei der Nutzung der AC-NN-Koeffizienten nahezu perfekt. Dagegen ist das Ergebnis bei den AC-N3-Koeffizienten, bei denen ein besseres Ergebnis zu erwarten war, deutlich schlechter. Werden zusätzlich auch die DC-Koeffizienten genutzt(linke Seite) ist das Ergebnis mit N3-Koeffizienten besser als mit NN-Koeffizienten. Zu beachten ist, dass keine der Methoden ein schlechtes Stenogramm liefert, da die Form der Kurven immer beibehalten wird und es nur zu leichten Verschiebungen kommt.
Es wurden viele Bilder mit unterschiedlichen Texten untersucht und es ergibt sich kein eindeutiges Ergebnis, bei Nutzung welcher Koeffizienten die Übereinstimmung am besten ist. Es ist immer sehr abhängig von der Kombination aus Bild und Nachricht, welche Methode das beste Ergebnis liefert. So können mit der Software mehrere Kombinationen erstellt und untersucht werden und die beste ausgewählt werden.

11.2. Fall 2: Filter auf Bild und Stego

Im zweiten Fall wird der oben beschriebene Filter auf das Bild und das Stego angewendet. Wie in der Tabelle 11.3 zu erkennen ist, ergeben sich kaum sichtbare Unterschiede. Nur bei der Uhrzeit unten rechts im Bild sind Unterschiede zu erkennen. Hier darf nicht vergessen werden, dass der Angreifer keinen Zugriff auf das Originalbild hat und kein direkter Vergleich möglich ist.

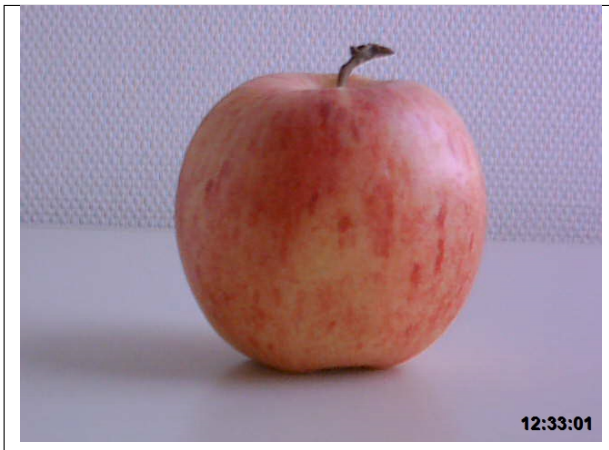
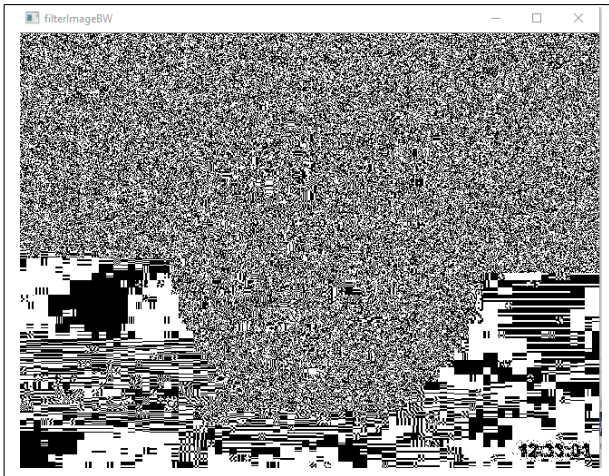
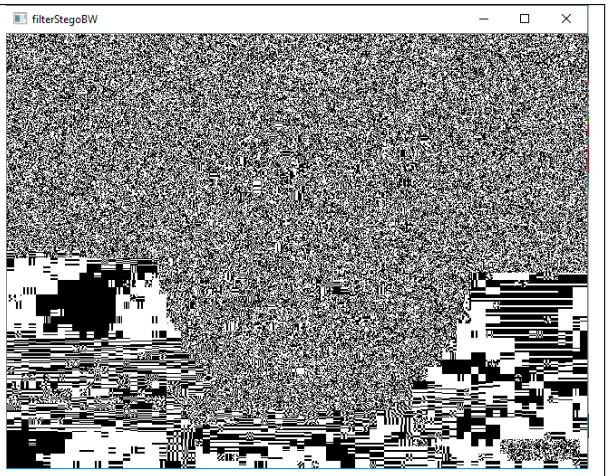
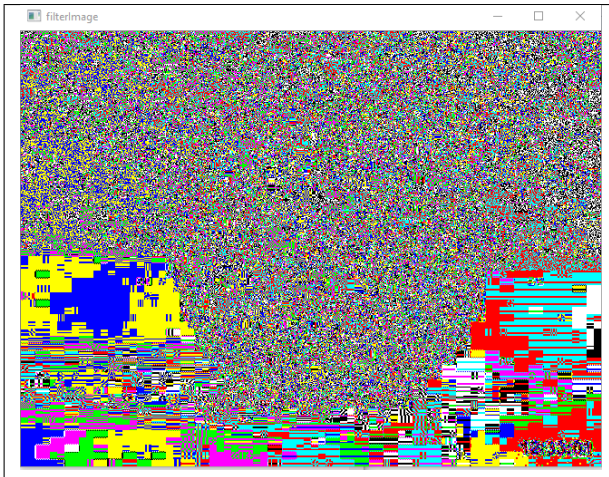
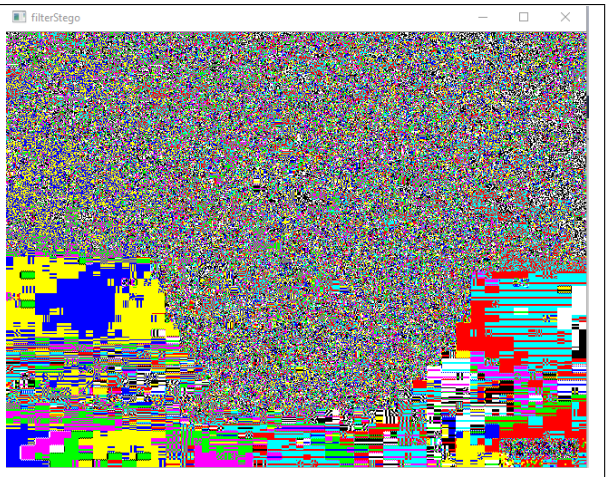
Bild	Steganogramm
	
Gefiltert/Bild(Schwarz-Weiß)	Gefiltert/Steganogramm(Schwarz-Weiß)
	
Gefiltert/Bild(RGB)	Gefiltert/Steganogramm(RGB)
	

Tabelle 11.3.: Nachricht: 5100 Byte, Bild: 103 KB, N3-AC-Koeffizienten: 5130 Byte(4,96% vom Bild), 4,93% vom Bild wurden überschrieben

11.3. Fall 3: Symmetrie-Vergleich

Im dritten Fall wird die Änderung der Symmetrie zwischen Originalbild(Abb. 11.1) und Steganogramm betrachtet. Die Verteilung der Koeffizienten-Werte ohne Null-Werte in ist in Abb. 11.2 dargestellt[2]. Es haben sich nur sehr geringfügige Änderungen bei der Verteilung der Koeffizienten ergeben, die ohne direkten Vergleich mit dem Original nicht auffallen würden.

Abbildung 11.1.: Auswertung der Symmetrie-Änderung

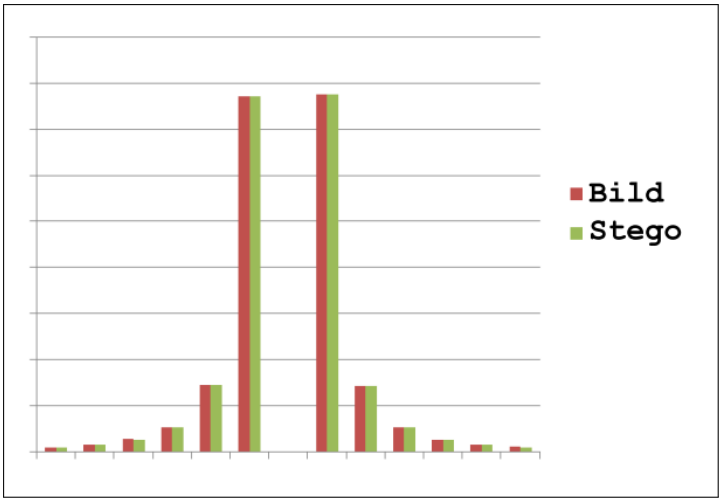


Abbildung 11.2.: Auswertung der Symmetrie-Änderung. Jeweils 20 Koeffizienten in einem Intervall.

[2] Es wird die Anzahl von 20 Koeffizienten-Werten pro Intervall dargestellt. Auch bei genauerer Auswertung der Symmetrie(z. B. 1 und -1 Vergleich jeweils in der Info.txt) ergaben sich keine großen Abweichungen.

12. Fazit

Ziel dieser Arbeit war es, eine Software zu erstellen, die ein kryptographisches Verfahren mit einem steganographischen Verfahren kombiniert. Durch die Kryptographie wird die Sicherheit erhöht, falls das Steganogramm erkannt und entschlüsselt wird. Zusätzlich wird durch das kryptographische Verfahren die Struktur eines Textes zerstört.

Dabei wurde im Bereich der Steganographie das JPEG-Bildformat als Container-Medium genutzt. Da das JPEG-Format ein kompliziertes Kompressionsverfahren einsetzt, musste dieses Verfahren im ersten Schritt bis ins Detail analysiert werden. Dieser detaillierte Einblick in das JPEG-Format war sehr interessant und aufschlussreich.
Daraufhin konnte ein Verfahren entwickelt werden, dass die geheime Nachricht in den Koeffizienten des JPEG-Bildes versteckt. Hierbei werden verschiedene Methoden angeboten und viele Informationen zum Ausgangsbild und dem erstellten Stego gespeichert, so das geprüft werden kann, ob das Stego sicher ist bzw. welche Methode für die Kombination aus Bild und Text am besten geeignet ist. So lässt sich für jede Nachricht ein geeignetes Bild finden, um ein gutes Steganogramm zu erstellen.
Für die Übertragung der erstellten Steganogramme wurde ein Webserver mit Python und Flask auf dem Raspberry Pi realisiert, so dass das Stego zwischen vielen anderen Bildern versteckt werden kann. Da dieser Server auch nur kurzfristig und unter wechselnden IP-Adressen erreichbar ist, erhöht diese Methode der Übertragung die Sicherheit. In Abb. 12.1 ist ein Bild der Webseite dargestellt.
In Kombination mit dem eingebauten RSA-Verfahren ergibt sich ein sicherer Übertragungsweg für geheime Nachrichten.
Zusätzlich wurde für die Software eine graphische Benutzeroberfläche entwickelt, so dass die Software komfortabel und schnell eingesetzt werden kann.

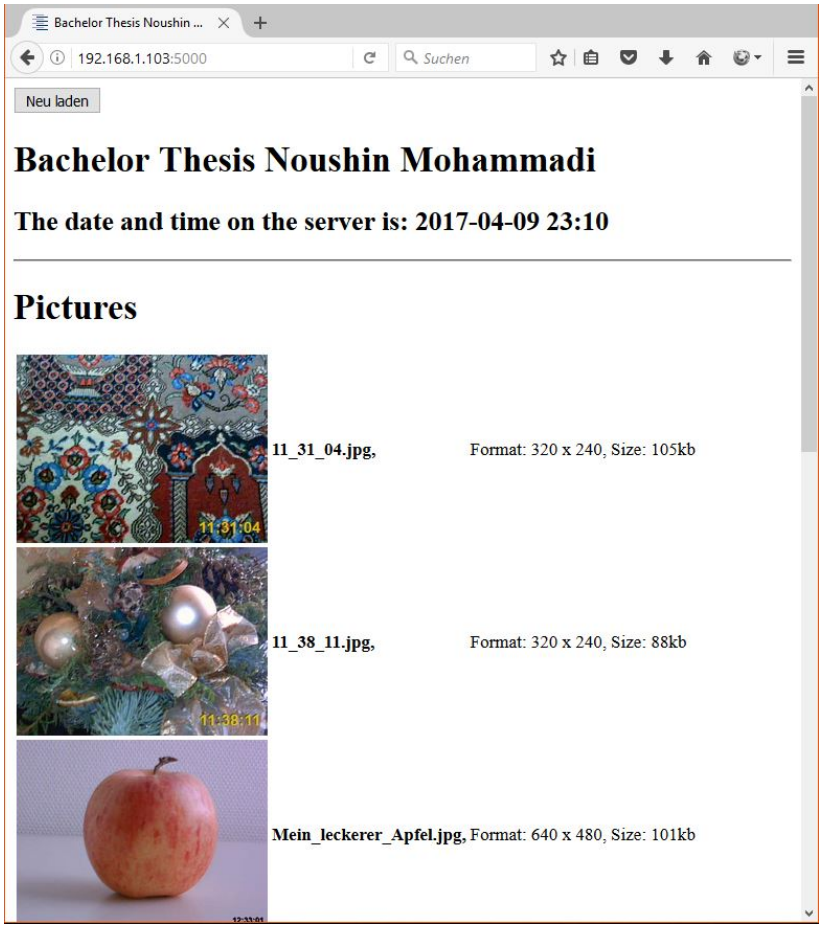


Abbildung 12.1.: Erstellte Webseite mit Testbildern

13. Ausblick

Während der Tests und Auswertungen sind immer wieder neue Ideen zur Erweiterung der Software entstanden.

So könnte die Auswahlmöglichkeit der genutzten Koeffizienten noch erweitert werden, um noch mehr Möglichkeiten zu bieten, ein möglichst sicheres Stego zu erstellen.

Bei Nachrichten, die deutlich kleiner sind, als die Kapazität des Bildes, könnte ein Permutations-Verfahren eingesetzt werden, um die Daten gleichmäßig über das Bild zu verteilen.

Um die Qualität der erstellten Stegos genauer untersuchen zu können, könnte die Analyse um weitere Aspekte erweitert werden.
Die bisherigen Analyse-Methoden könnten in die GUI integriert werden.

Die aktuelle Software ist auf JPEG-Bilder der genutzten Systemkamera beschränkt. Ein Ziel wäre es, die Software so zu erweitern, dass sie für alle JPEG-Bilder genutzt werden kann.

Teil V.

Software Anleitung

14. Anleitung

Da die erstellte Software viele Funktionen bietet und eine große Menge an Informationen zu den JPEG-Bildern und Stegos zur Verfügung stellt, wird in diesem Kapitel eine Anleitung zur Nutzung der Software zur Verfügung gestellt.

Die Software mit dem Namen „GUI.exe“ befindet sich im Ordner „JPEG_Workspace“ auf der DVD. In diesem Ordner befinden sich zusätzlich die „Huffman_Encoder.exe“ und „Huffman_Decoder.exe“. Dieser Ordner kann auf ein beliebiges Laufwerk auf dem Computer kopiert werden. Alle Bilder, Texte und Schlüssel-Dateien müssen in diesem Ordner liegen, damit alle Prozesse korrekt laufen.
Der Aufruf erfolgt dann über die „GUI.exe“ und es öffnet sich das Hauptfenster(Abb. 14.1). Über den Menüpunkt „Options“ sind die drei Hauptfunktionen auswählbar.

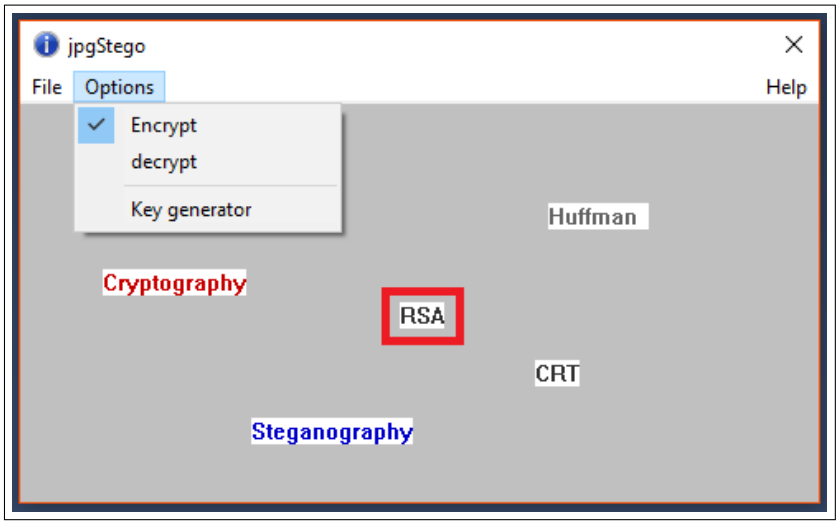


Abbildung 14.1.: Hauptfenster mit Options-Menü

Als erstes werden die Schlüssel für die RSA-Verschlüsselung benötigt. Hierfür wird der Key-Generator(Abb. 14.2) genutzt. Hier muss der Benutzer einen initialen Wert im Bereich „unsigned Integer“ eingeben und mit „Enter“ bestätigen. Der Key-Generator erzeugt daraufhin den privaten Schlüssel d in der Datei „private_key_d.txt“ und den öffentlichen Schlüssel n in der

Datei „public_key_n.txt“. Diese beiden Dateien werden unbedingt in diesem Ordner benötigt, um eine Nachricht ver- oder entschlüsseln zu können. Zusätzlich wird die Datei „secret.txt“ erzeugt, in die noch die benutzen 512-Bit-Primzahlen p und q, sowie der feste öffentliche Schlüssel $e = 0x11$ geschrieben wurden. p, q und d müssen im realen Einsatz unbedingt geheim gehalten werden und werden hier nur zur Kontrolle ausgegeben.

Abbildung 14.2.: Key-Generator

Im Hauptfenster können die beiden anderen Optionen „Encrypt“ und „Decrypt“ ausgewählt werden und so mit einem Haken versehen werden. Die entsprechenden Fenster öffnen sich aber erst nach einem Rechtsklick auf einen versteckten Button, der sich auf dem RSA-Label befindet(In Abb. 14.1 rot markiert). So kann nicht jeder unberechtigte Nutzer die Software ohne weiteres nutzen.

Das Decrypt-Fenster ist in Abb. 14.3 dargestellt. Hier können die Nachrichten aus erhaltenen Stegos extrahiert werden. Dazu muss das Stego und die Datei mit dem privaten Schlüssel im Ordner „JPEG_Workspace“ liegen. Das Stego kann ausgewählt und der Prozess über den „Process“-Button gestartet werden.
Sollte in der Datei eine Nachricht versteckt sein, wird diese in die Datei „message_decrypt.txt“ geschrieben. Wurde die versteckte Nachricht mit dem Huffman-Verfahren kodiert, wird diese nach der Entschlüsselung in die Datei „rebuild_message_decrypt.txt“ geschrieben.

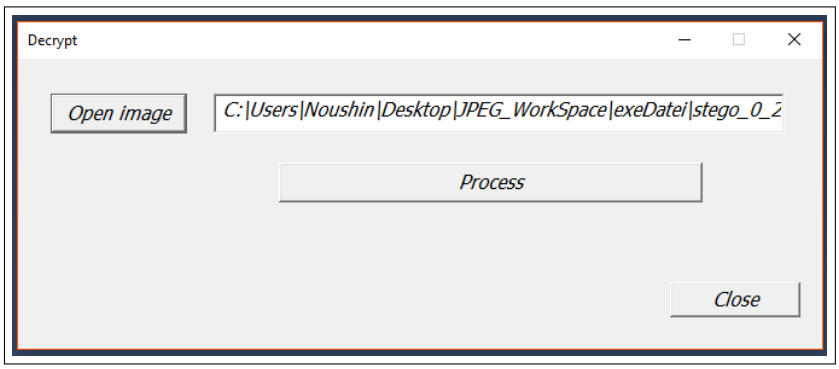


Abbildung 14.3.: Decrypt-Fenster zur Entschlüsselung von Stegos

Das Encrypt-Fenster ist in Abb. 14.4 dargestellt. Das Fenster ist in drei Bereiche aufgeteilt. Im ersten Bereich kann ein Bild ausgewählt werden. Dieses wird analysiert und die Bildinformationen dargestellt. Neben der Dateigröße und dem Bildformat, wird hier die Gesamtzahl der (AC-)Koeffizienten(Count) und die entsprechende Kapazität in Byte angezeigt.
Im zweiten Abschnitt kann die Textdatei ausgewählt werden. Diese wird mit dem Huffman-Verfahren codiert und mit dem RSA-Verfahren verschlüsselt. Hierzu muss die Datei mit dem öffentlichen Schlüssel n des Empfängers im Ordner liegen. Es werden die Größen der Nachrichten und die mögliche Einsparung durch die Huffman-Codierung angezeigt. Hier muss über eine Checkbox ausgewählt werden, ob die Huffman-Codierung zum Komprimieren der Nachricht verwendet werden soll.
Im dritten Teil werden genauere Informationen zu den Koeffizienten des Bildes angezeigt. Diese sind in vier mögliche Optionen aufgeteilt:
- NN-Coeff. sind alle Koeffizienten, die nicht den Wert 0 haben
- N3-Coeff. sind alle Koeffizienten, die nicht den Wert 0, 1 oder -1 haben
- NN-AC-Coeff. sind alle AC-Koeffizienten, die nicht den Wert 0 haben
- N3-AC-Coeff. sind alle AC-Koeffizienten, die nicht den Wert 0, 1 oder -1 haben

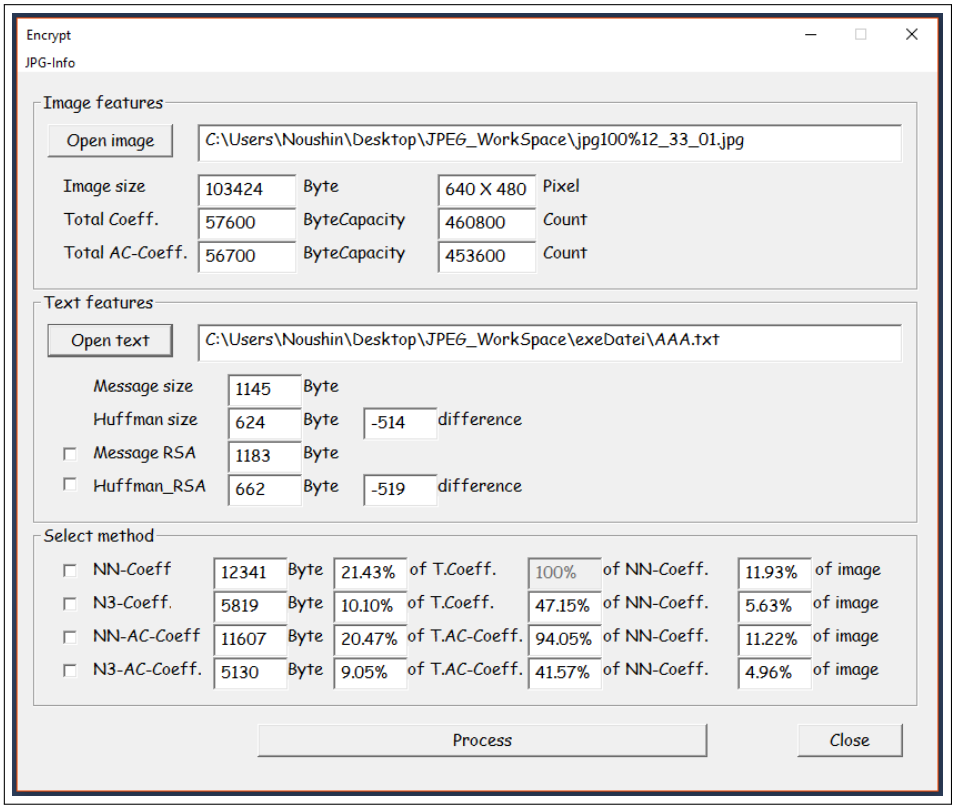


Abbildung 14.4.: Encrypt-Fenster

Im Beispiel in der Abb. 14.4 sind insgesamt 12.341 Byte oder 21.43% der gesamten Byte-Kapazität nicht Null. In der dritten Spalte wird die Anzahl der Koeffizienten im Bezug auf alle Koeffizienten ungleich Null angezeigt und in der vierten Spalte die Kapazität in Bezug auf die Gesamtgröße der Bilddatei.
Hier kann jetzt, entsprechend der Nachrichtengröße aus dem zweiten Abschnitt, ausgewählt werden, in welchen Koeffizienten die Nachricht versteckt werden soll. Dabei bietet die beste Option, die N3-AC-Koeffizienten, am wenigsten Platz.
Mit dem „Process“-Button wird die Nachricht in dem Bild versteckt. Das Ergebnis ist ein JPEG-Bild mit dem Namen „Stego“ gefolgt von der aktuellen Uhrzeit.
Encrypt- und Decrypt-Fenster können mit dem „Close“-Button unten rechts geschlossen werden und das Programm kehrt zum Hauptfenster zurück. Werden die Fenster über das Close-Symbol(x) oben rechts geschlossen, wird das Programm komplett beendet.

Oben links im Encrypt-Fenster gibt es die Menü-Option „JPG-Info“. Es öffnet sich das Fenster aus Abb. 14.5. Hier kann ausgewählt werden, welche zusätzlichen Informationen in weiteren Dateien zur Analyse gespeichert werden sollen. Hier ist unbedingt zu beachten, dass das Fenster nach Auswahl der Optionen im Hintergrund geöffnet bleiben muss, damit die ausgewählten Informationen während der Codierung erkannt werden.
Alle erzeugten Dateien sind nach Abschluss im Ordner „JPEG-Workspace“ zu finden.

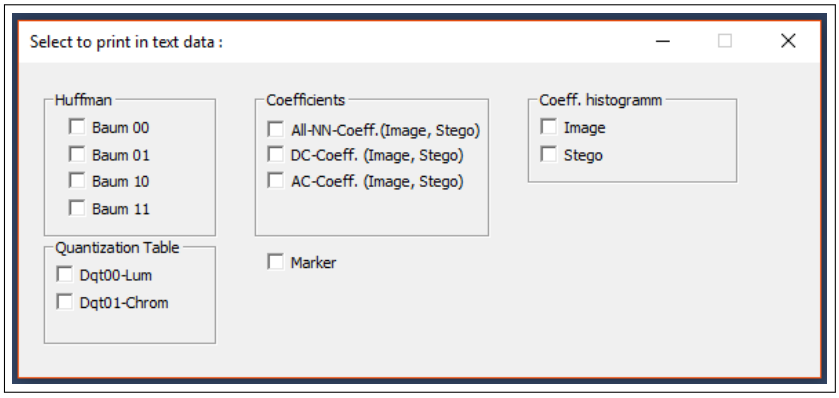


Abbildung 14.5.: JPG-Info-Fenster

14.1. Fehler-Meldungen

Leider ist es nicht in allen Teilen des Programms möglich, auftretende Fehler durch eine Message-Box mitzuteilen.
In den Fällen, wo dies nicht möglich ist, wird der Grund für den Abbruch des Programms in die Datei „Info_$Uhrzeit$.txt“[1] geschrieben. Dies ist z. B. der Fall, wenn die Dateien mit den Schlüsseln d oder n im Workspace-Ordner fehlen.

Die Ausführung des Programms direkt auf der CD ist **nicht** möglich.

[1] In diese Datei werden auch laufend die wichtigsten Informationen zu Bild und Stego geschrieben, z. B. Bild-Size vs. Stego-Size

Literaturverzeichnis

[Bradski und Kaebler] BRADSKI, Gray ; KAEBLER, Adrian: *OpenCV*. O'REILLY. – ISBN 978-0-596-51613-0

[Compression] WIKIPEDIA: *Lossless compression*. – URL https://en.wikipedia.org/wiki/Lossless_compression. – Letzter Zugriff: 13.02.2017

[CRT a] WIKIPEDIA: *Chinese remainder theorem*. – URL https://en.wikipedia.org/wiki/Chinese_remainder_theorem. – Letzter Zugriff: 23.01.2017

[CRT b] SHINDE, G.N. ; FADEWAR, H.S.: *Faster RSA Algorithm for Decryption Using Chinese Remainder Theorem*. – URL https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwipusbW-OzSAhXI_ywKHZuSC7wQFgguMAE&url=http%3A%2F%2Fwww.techscience.com%2Fdoi%2F10.3970%2Ficces.2008.005.255.pdf&usg=AFQjCNFUWS5Ud1_GCPST1QxlcZKsiDEC5w. – Zugriff: 11.01.2017

[Flask a] WIKIPEDIA: *"Flask"*

[Flask b] RICHARDSON, Matt: *Serving Raspberry Pi with Flask*. – URL http://mattrichardson.com/Raspberry-Pi-Flask/. – Letzter Zugriff: 05.04.2017

[Huffman] WIKIPEDIA: *"Huffman Codierung"*. – URL https://www.google.de/search?q=g%C3%A4nsef%C3%BCssschen+in+latex&ie=utf-8&oe=utf-8&client=firefox-b&gfe_rd=cr&ei=Uce_WMm-L-Sl8wfd4byICg. – Letzter Zugriff: 08.12.2016

[JPEG a] PROGRAMMFABRIK: *BILDFORMATE*. – URL https://www.programmfabrik.de/wissen/bildformate-gif-png-jpg-tiff/. – Letzter Zugriff: 08.03.2017

[JPEG b] WIKIPEDIA: *Huffman codierung*. – URL https://en.wikipedia.org/wiki/Huffman_coding. – Letzter Zugriff: 09.12.2016

[JPEG c] INTERNET: *JPEG*. – URL https://en.wikipedia.org/wiki/JPEG). – Letzter Zugriff: 01.04.2017

[JPEG d] FILEFORMAT.INFO: *JPEG Compression*. – URL http://www.fileformat.info/mirror/egff/ch09_06.htm. – Letzter Zugriff: 19.01.2017

[JPEG e] INTERNETSEITE: *JPEG File Layout Format*. – URL http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20Layout%20and%20Format.htm. – Letzter Zugriff: 7.03.2017

[JPEG f] WICKENBURG, Sebastian ; ROOCH, Aeneas ; GROSS, Johannes: *Die JPEG-Kompression*. – URL http://www.mathematik.de/spudema/spudema_beitraege/beitraege/rooch/einleit.html. – Letzter Zugriff: 27.03.2017

[JPEG g] ROMAN10: *JPEG Standard*. – URL http://www.roman10.net/2011/08/13/jpeg-standarda-tutorial-based-on-analysis-of-sample-picturepart – Zugriff: 5.10.2016

[JPEG h] PROGRAMMFABRIK: *JPEG Standard-A Tutorial Based on Analysis of Sample Picture-Part 1. Coding of a 8x8 Block*. – URL http://www.roman10.net/2011/08/09/jpeg-standarda-tutorial-based-on-analysis-of-sample-picturepart-1-coc – Letzter Zugriff: 22.03.2017

[Jsteg] INTERNETSEITE: *Extracting data embedded with JSteg*. – URL http://www.guillermito2.net/stegano/jsteg/. – Letzter Zugriff: 23.11.2016

[Laganiere] LAGANIERE, Robert: *OpenCV Computer Vision Application Programming Cookbook*. Packet Publishing Ltd. – ISBN 978-1-78216-148-6

[Master Arbeit] DAUCH, Dominique I.: *Implementierung eines Steganographie- und Kryptographie-Verfahrens auf einem FPGA mit Betriebssystemanbindung*

[PNG a] GROUP: *PNG Specification, Version 1.2*. – URL http://www.libpng.org/pub/png/spec/1.2/PNG-Contents.html. – Letzter Zugriff: 19.11.2016

[PNG b] RECOMMENDATION, W3C: *PNG Specification(Second Edition)*. – URL https://www.w3.org/TR/2003/REC-PNG-20031110/. – Letzter Zugriff: 15.03.2017

[PNG c] WIKIPEDIA: *Portable Network Graphics*. – URL https://en.wikipedia.org/wiki/Portable_Network_Graphics. – Letzter Zugriff: 13.08.2016

[PNG d] RECOMMENDATION, W3C: *Portable Network Graphics(PNG) Specification*. – URL https://www.w3.org/TR/PNG/#4Concepts.Encoding. – Letzter Zugriff: 13.02.2017

[Prinz und Kirch] PRINZ, Peter ; KIRCH, Ulla: *C Lernen und professionell anwenden*. mitp. – ISBN 978-3-8266-9504-9

[Python a] RESOURCES, Raspberry Pi L.: *Build a Python Web Server with Flask*. – URL https://www.raspberrypi.org/learning/python-web-server-with-flask/. – Letzter Zugriff: 10.02.2016

[Python b] WIKIPEDIA: *Python*. – URL https://de.wikipedia.org/wiki/Python_(Programmiersprache). – Letzter Zugriff: 19.03.2017

[Redundanz] WIKIPEDIA: *Redundanz*. – URL https://de.wikipedia.org/wiki/Redundanz_(Informationstheorie)

[RSA] WIKIPEDIA: *RSA(cryptosystem)*. – URL https://en.wikipedia.org/wiki/RSA_(cryptosystem). – Zugriff: 22.11.2016

[Schmeh 2016] SCHMEH, Klaus: *kryptografie*. dpunkt.verlag, 2016. – ISBN 978-3-86490-356-4

[TU] TU, Freiberg: *Fermatsche Primzahlen*. – URL http://www.mathe.tu-freiberg.de/~hebisch/cafe/fermatprim.html. – Zugriff: 20.10.2016

[Welschenbach] WELSCHENBACH, Michael: *Kryptographie in C und C++*. Springer-Verlag. – ISBN 3-540-64404-0

Anhang

Alle Anhänge befinden sich auf der mitgelieferten DVD. Hier eine Übersicht über den Inhalt der DVD.

SourceCodes

- GUI
- Huffman_Encoder_Decoder
- OpenCV
- Raspberry_Webserver

JPEG_Workspace

- GUI.exe
- Huffman_Encoder.exe
- Huffman_Decoder.exe
- Testbilder
- Testtext

Auswertung

- Excel-Tabelle zur Symmetrie-Auswertung

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 10. April 2017	
Ort, Datum	Unterschrift