HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Thomas Riedl

Glaubhafte Abstreitbarkeit durch Blockgeräteverschlüsselung in Kombination mit Steganografie

Ein „proof-of-concept“ aus dem Themenkomplex der Anti-Forensik

2020

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Glaubhafte Abstreitbarkeit durch Blockgeräteverschlüsselung in Kombination mit Steganografie

Ein „proof-of-concept“ aus dem Themenkomplex der Anti-Forensik

Autor:
Thomas Riedl

Studiengang:
IT-Forensik/Cybercrime

Seminargruppe:
CC16w1-B

Erstprüfer:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

Zweitprüfer:
Stefan Schildbach, M. Sc.

Mittweida, Juli 2020

Faculty **Applied Computer Sciences and Biosciences**

BACHELORTHESIS

Plausible deniability through block device encryption in combination with steganography

A „proof of concept“ in the field of anti-forensics

Author:
Thomas Riedl

Study Programme:
IT-Forensik/Cybercrime

Seminar Group:
CC16w1-B

First Referee:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

Second Referee:
Stefan Schildbach, M. Sc.

Mittweida, July 2020

Bibliografische Angaben

Riedl, Thomas: Glaubhafte Abstreitbarkeit durch Blockgeräteverschlüsselung in Kombination mit Steganografie, Ein „proof-of-concept“ aus dem Themenkomplex der Anti-Forensik, 61 Seiten, 10 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Bachelorarbeit, 2020

Satz: LaTeX

Referat

Das Ziel der vorliegenden Bachelorarbeit war es, eine Software zu entwickeln, die plausible Abstreitbarkeit auf Grundlage einer Trennung der Metadaten von den Nutzdaten eines verschlüsselten Datenträgers erreicht. Beide Teile wurden auf unterschiedliche Weise mittels bildsteganografischer bzw. datenträgersteganografischer Methoden dahingehend verändert, um letztendlich die Existenz verschlüsselter Daten verbergen zu können. Die als “Proof-of-Concept“ entwickelte Software für Linux-Betriebssysteme soll dabei für den Nutzer ein möglichst transparentes Arbeiten gewährleisten aber durch die steganografischen Komponenten einen möglichst niedrigen Mehraufwand aufweisen. Sie greift weitestgehend auf Standardwerkzeuge zurück, wurde in der Programmiersprache Python 3 implementiert und steht hinsichtlich des Open-Source-Gedankens unter der GNU General Public License.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
BMP	Windows Bitmap Format
CLI	Command-Line Interface
CRC	Cyclic Redundancy Check
DCO	Device Configuration Overlay
DCT	Diskrete Kosinustransformation
GIF	Graphics Interchange Format
GUI	Graphical User Interface
HPA	Host Protected Area
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
LSB	Least Significant Bit
LUKS	Linux Unified Key Setup
LVM	Logical Volume Manager
MBR	Master Boot Record
PIL	Python Image Library
PNG	Portable Network Graphics
RAID	Redundant Array of Independent Disks
RAM	Random-Access Memory
UUID	Universally Unique Identifier
XOR	Exklusiv-Oder

1 Einleitung

Seit vielen Jahren kursieren immer wieder Schlagzeilen durch die Medien, Verschlüsselung sei eine Gefahr und sie müsste unbedingt gebannt werden. Erst im Januar diesen Jahres entbrannte wieder einmal der Verschlüsselungsstreit, in dem US-Präsident Donald Trump das Unternehmen Apple wegen gesperrter iPhones verbal attackierte. Auch wenn hierzulande Dinge wie das Internet und die damit verbundenen Technologien immer noch “Neuland“ zu sein scheinen, befinden wir uns längst in einem hochdigitalisierten Zeitalter, in dem auch Kryptografie nicht nur allein für das Militär relevant ist, sondern für jeden Einzelnen von uns, um die eigene Privatsphäre zu schützen.

Es gibt derzeit einige Länder in denen Bestrebungen geführt wurden Kryptografie zu verbieten bzw. deren legale Nutzung auszuhebeln. Mag man hierbei zunächst an absolutistisch regierte Nationen denken, wird man recht bald feststellen, dass auch einige europäische Nationen - wie Großbritannien oder Frankreich - bereits entsprechende “Key Disclosure Laws“ erlassen haben. Damit werden Behörden die rechtliche Grundlage gegeben, Passwörter “höflich zu erfragen“ oder man landet eben im Gefängnis.

Letztendlich geben alle Anti-Verschlüsselungsgesetze vor, Kriminalität einzudämmen, indem sie die Bürger daran hindern wollen, ihre Online-Kommunikation durch Verschlüsselung zu schützen. Doch genau hier liegt auch das Problem. Welche Arten der Verschlüsselung sollen denn nun genau verboten werden? Bleibt es bei der Ende-zu-Ende-Verschlüsselung von Messenger-Diensten oder sollen diese Gesetze immer weiter ausgedehnt werden? Macht man sich irgendwann schon verdächtig, nur weil man Verschlüsselung zum Schutz seiner privaten Daten nutzt? Und wer entscheidet darüber? Vermutlich diejenigen Menschen, aus deren Reihen jüngst die Idee kam, man könne doch einmal darüber nachdenken, Desinfektionsmittel zur Bekämpfung einer viralen Infektion zu verabreichen. Auch wenn diese Aussage später als Sarkasmus abgetan wurde, wird deutlich, dass es Regierungen oftmals an entsprechendem Weitblick fehlt. Werden durch Gesetzgebung bestimmte Probleme möglicherweise gelöst oder eingedämmt, so kann dies sicherlich auch weitreichende Folgen in anderen Belangen haben. Ein Verschlüsselungsverbot schafft letztendlich nur wieder neue Möglichkeiten, wodurch Cyberkriminellen beispielsweise der Identitätsdiebstahl erleichtert wird.

Am Ende haben all diese Anti-Verschlüsselungsbestrebungen zwei Punkte gemeinsam: Sie versuchen, beobachtete Fakten zu reglementieren und die grundlegenden Herangehensweisen sind vollkommen falsch!

1.1 Problemstellung

Das Problem der Anti-Verschlüsselungspolitik liegt natürlich auf der Hand. Es wird immer Daten oder Informationen gegen, die aus diversen Gründen nicht in die Hände der falschen Leute fallen sollen oder dürfen. Dies können private Dinge sein, mit denen sich Kriminelle zu bereichern suchen oder aber auch Betriebsgeheimnisse von denen der Erfolg eines Unternehmens abhängen kann.

Begibt man sich - privat wie auch geschäftlich - auf Reisen, trägt man mit seinem Notebook oder Smartphone eigentlich immer potentiell sensible Daten bei sich. Um die Kreditkarten-PIN oder Zugangsdaten zum Onlinebanking vor unrechtmäßigem Gebrauch im Falle des Verlusts oder Diebstahls zu sichern, sollten diese niemals irgendwo im Klartext abgelegt sein. Die Gefahren des Identitätsdiebstahls und der damit verbundenen Folgen sind größer denn je, sodass eine Nutzung kryptografischer Verfahren in der heutigen Zeit schon zur Selbstverständlichkeit gehören sollte. Auch Betriebsgeheimnisse aller Art haben in den Händen Dritter sicherlich nichts verloren und haben in der Regel auch keine strafrechtliche Relevanz.

Wird man nun beispielsweise am Flughafen kontrolliert und durch entsprechende “Key Disclosure Laws“ um die Preisgabe seiner Passwörter gebeten, ist es sicherlich kein Problem, die PIN der Kreditkarte oder die Zugangsdaten zum Onlinebanking zu offenbaren. Doch es gibt auch Daten, die man nicht unbedingt preisgeben möchte. Hierbei handelt es sich selbstverständlich stets um erlaubte Inhalte; schließlich wäre alles andere moralisch höchst verwerflich!

1.2 Motivation

Man muss sich schließlich fragen, wie man mit entsprechenden Restriktionen umgehen soll und wie diese in ein Paar Jahren aussehen könnten. Steht das Risiko eines möglichen Verlusts oder die Gefahr einer Veröffentlichung betrieblicher Internas im Verhältnis zu umstrittenen Vorgaben, denen man sich beugen soll? Oder besteht auch die Möglichkeit, Verschlüsselung unerkannt zu nutzen? Grundsätzlich kann nur sanktioniert werden, was offensichtlich untersagt ist. Wäre in diesem Fall die Tatsache der Verschlüsselung gar nicht erst offensichtlich, so können Maßnahmen aufgrund eines Verbots gar nicht erst greifen.

Am Beispiel der Grenzkontrolle kann man davon ausgehen, dass sich diese - zumindest in halbwegs demokratischen Ländern - nicht grundlos ins Unendliche ziehen lässt. Denn wenn erst einmal eine richterliche Entscheidung benötigt wird, aktuell aber keine handfesten Beweise vorliegen, dann dürften etwaige Maßnahmen ein Ende haben und man hat letztendlich gewonnen.

Um solchen Umständen von vornherein aus dem Weg zu gehen wird ein Tool benötigt, dass nicht nur Verschlüsselung bietet, sondern dessen Existenz auch vor kritischeren Blicken zu verbergen vermag. Natürlich ist kein Verfahren 100% sicher, aber man sollte es seinen “Kontrahenten“ möglichst schwer machen und sie dadurch bei der Suche - um es sportlich zu formulieren - zur vorzeitigen Aufgabe zu bewegen.

Bei der Kryptografie werden Daten grundsätzlich auf eine sichtbare Art und Weise verschlüsselt. Zwar sind die geheimen Informationen (Betriebsgeheimnisse, Forschungsergebnisse, journalistische Recherchen, etc.) nicht lesbar, doch das bloße Vorhandensein verschlüsselter Daten lässt sich nur schwer leugnen. Selbst wenn man sich keine strafbaren Handlungen vorzuwerfen hat, könnte diese Tatsache alleine einen schon zur Zielscheibe werden lassen. Dies gilt es möglichst zu vermeiden, um potentiellem Ärger vornherein aus dem Weg gehen zu können.

An dieser Stelle kommt die Steganografie ins Spiel; sie kann definiert werden als der Gebrauch eines Verfahrens, mit dessen Hilfe eine Botschaft verborgen wird. Werden nun geheime Informationen in ein *Trägermedium* eingebettet, so entsteht daraus ein *Steganogramm*, dem man im Idealfall die Manipulation nicht anmerkt. Kombiniert man schließlich die Steganografie mit der Kryptografie auf geeignete Weise, so bleibt im Gegensatz zu anderen Techniken neben den eigentlichen Daten auch die Tatsache des Verschlüsselns selbst geheim.

Bereits im Jahre 1883 formulierte Auguste Kerckhoffs den Grundsatz der modernen Kryptografie. Dieser besagt, dass die Sicherheit eines kryptografischen Verfahrens einzig und allein auf der Geheimhaltung des Schlüssels beruht und nicht auf der Geheimhaltung des Algorithmus. Dies wäre bei reiner Steganografie gerade umgekehrt, da die Sicherheit auf der Geheimhaltung des Algorithmus beruht; man spricht auch von “Security Through Obscurity“. Um dem Kerckhoffs’schen Prinzip auch bei Anwendung der Steganografie gerecht zu werden, sollten die einzubettenden Daten grundsätzlich chiffriert werden. Dies bringt aber auch einen weiteren Vorteil mit sich, da moderne Verschlüsselungsalgorithmen zu einer Gleichverteilung der Daten neigen und dieser Umstand die Steganalyse erschwert.

1.3 Ziel der Arbeit

Diese Arbeit setzt sich zum Ziel, glaubhafte Abstreitbarkeit (plausible deniability) in einem “Proof-of-Concept“ durch geschickte Kombination von Kryptographie und Steganographie zu realisieren. Bei einer solchen Software sollte eine Umgebung geschaffen werden, in der ein Nutzer transparent arbeiten kann, d. h. als wären Kryptografie bzw. Steganografie gar nicht vorhanden. Im Hintergrund werden die schützenswerten Informationen durch Verschlüsselung gesichert und diese selbst sodann durch Steganografie unsichtbar gemacht. Die Software wird für das Betriebssystem Linux erstellt.

Bei Recherchen im Internet findet man tonnenweise Software, die diverse steganografische Verfahren umsetzen und damit in der Lage sind, die gewünschten Ergebnisse zu erzielen. Meist arbeitet eine reine Steganografiesoftware allerdings auf Dateiebene, was für den angestrebten Zweck nicht optimal ist.

Ziel dieser Software ist es, dem Nutzer einen in der Größe bestimmbaren zusammenhängenden Speicherbereich zu bieten, auf dem nach Belieben Dateien erstellt, geöffnet, verändert oder gelöscht werden können, wie dies auch bei einem regulärem Dateisystem in transparenter Form möglich ist. Wurde ein entsprechender Container, der die Daten letztendlich beinhaltet einmal erstellt, so soll er immer wieder geöffnet, benutzt und sodann wieder geschlossen werden können.

Da die kryptografischen Implementierungen der Linux-Welt oftmals zu den Standardtools vieler Derivate gehören, liegt das Hauptaugenmerk dieser Arbeit auf der Steganografiekomponente. Neben diversen Möglichkeiten des dateiweisen Chiffrierens bietet der Linux-Kernel eine ausgereifte Implementierung zur Verschlüsselung von Blockgeräten, was sich die Software zu nutze machen wird.

Ein solches Blockgerät soll inklusive aller Metadaten mittels diverser steganografischer Verfahren insoweit verborgen werden, um letztlich das Ziel “Glaubhafte Abstreitbarkeit“ zu erreichen. Damit kann auch die Existenz großer Datenmengen vor den Blicken Dritter geschützt werden, um beispielsweise Zensur zu umgehen.

2 Steganografische Verfahren

Im Gegensatz zur Kryptografie, wo Nachrichten oder Daten offensichtlich, d. h. durch die erkennbare Nutzung eines Kryptosystems chiffriert werden, liegt das Hauptaugenmerk der Steganografie auf dem Verstecken von Informationen in einem Trägermedium. Während in der Kryptografie bekannt ist, dass chiffrierte Informationen vorliegen, soll die Steganografie erreichen, dass einem potenziellen Angreifer die Existenz schützenswerter Daten gänzlich verborgen bleibt. Um die Vertraulichkeit der Daten als klassisches Schutzziel der Informationssicherheit zu gewährleisten, sollte Steganografie stets mit Verschlüsselung kombiniert werden. [1]

Zur Einbettung von Informationen in ein Trägermedium gibt es zahlreiche Methoden. Damit die Steganografie allerdings effektiv umgesetzt werden kann, müssen eine Reihe von Voraussetzungen erfüllt sein. Die wichtigsten Grundvoraussetzungen sind in der nachfolgenden Auflistung aufgeführt: [2]

- Bei der Überführung der Informationen in das Steganogramm darf deren Integrität nicht gefährdet werden. Während des steganografischen Prozesses darf die geheime Nachricht deshalb nicht durch Hinzufügen, Ändern oder Löschen von Informationen verändert werden. Das eigentliche Ziel der Steganografie wäre verfehlt, wenn sich die ursprünglichen Informationen nicht wiederherstellen ließen.
- Das Steganogramm darf für das bloße Auge kaum oder bestenfalls gar nicht vom Original unterschieden werden können. Wird das Trägermedium zu stark verändert könnte die Existenz verborgener Informationen von einem Außenstehenden festgestellt werden, der seinerseits nunmehr versuchen könnte, diese zu extrahieren oder zu zerstören.
- Idealerweise sollten nachträgliche Manipulationen am Steganogramm bis zu einem gewissen Maße nicht die eingebetteten Informationen verändern oder zerstören. Dies könnten beispielsweise Veränderungen der Größe, sowie Drehung oder Bescheidung des Bildes sein. Diese Voraussetzung ist insbesondere für das “Watermarking“ von Bedeutung, wobei steganografische Verfahren zum Einbetten von Informationen über den Urheber des Werkes genutzt werden. Werden einfache Verfahren genutzt, lässt sich das Copyright recht leicht entfernen.
- Schlussendlich sollte man immer davon ausgehen, dass ein potentieller Angreifer die Existenz versteckter Informationen herausfinden könnte.

Im heutigen digitalen Zeitalter wächst die Vielfalt potentieller Trägermedien stetig an, die sich durch das Einbetten von Informationen zu Steganogrammen machen lassen. Heutzutage wird oftmals auf Bild-, Audio-, oder Videodateien zurückgegriffen, während in den Anfangszeiten oftmals unverfängliche Textpassagen als Trägermedien dienten. Auf den folgenden Seiten werden diverse Methoden näher betrachtet und bewertet.

2.1 Allgemeines

Der eigentliche Terminus Steganographie lässt sich auf die beiden griechischen Wörter $\sigma\tau\varepsilon\gamma\alpha\nu\text{o}\varsigma$ (steganós) und $\gamma\rho\alpha\phi\varepsilon\iota\nu$ (gráphein) zurückführen, die zusammengesetzt sinngemäß “verdeckt schreiben“ bedeuten. So ist es auch nicht verwunderlich, dass textsteganografische Verfahren bis ins 20. Jahrhundert hinein vorherrschend waren, da erst mit dem Informationszeitalter zahlreiche neue Möglichkeiten entstanden.

In Anlehung an die Kryptografie werden Formen der Textsteganografie mitunter auch als *“null cipher“* bezeichnet, da faktisch keine Chiffrierung im kryptografischen Sinne gegeben ist. Die geheime Nachricht ist lediglich in einem unverfänglichen, oftmals sinnlos anmutenden Text untergebracht, dessen Länge um ein vielfaches länger ist. Um die geheime Nachricht wieder sichtbar zu machen, muss dem Empfänger das verwendete Verfahren bekannt sein, wie *jeder n-te Buchstabe* oder *jedes n-te Wort*.

Das nachfolgende Beispiel von Textsteganografie in Form zweier Telegramme eines deutschen Spions stammt aus dem ersten Weltkrieg:

1. Telegram: *“president’s embargo ruling should have immediate notice. grave situation affecting international law. statement foreshadows ruin of many neutrals. yellow journals unifying national excitement immensely.“*
2. Telegram: *“apparently neutral’s protest is thoroughly discounted and ignored. isman hard hit. blockade issue affects pretext for embargo on byproducts, ejecting suets and vegetable oils.“*

Betrachtet man in der ersten Botschaft nur die jeweils ersten Buchstaben eines Wortes, ergibt sich die geheime Nachricht: ***Pershing sails from N.Y. June 1***. Die zweite Botschaft diente als Überprüfung der ersten und enthüllte dieselbe geheime Nachricht; man musste allerdings den jeweils zweiten Buchstaben betrachten. [3]

Bei den modernen, computergestützten steganografischen Verfahren werden geheime Nachrichten vor allem in Bild-, Audio- und Video-Daten eingebettet. Sie sind zahlreich vorhanden und es wurden zahlreiche verschiedene Verfahren entworfen. Darüber hinaus existieren noch weitere steganografische Formen, die allerdings aufgrund des angestrebten Zwecks kategorisch ausgeschlossen werden können. Ein Beispiel wäre die Netzwerksteganografie, bei der Informationen in die Protokolle bzw. die Datenübertragung selbst eingebettet werden. Für die Umsetzung des Projekts hat sich allerdings herausgestellt, dass sich Bild- und Datenträgersteganografie hierfür am besten eignen und implementieren lassen dürften.

Abschließend sei noch angemerkt, dass sich Steganografie auch alternativ nutzen lässt um geschützte Werke mit individuellen digitalen Wasserzeichen zu versehen. Das sogenannte *“watermarking“* ist für diese Arbeit allerdings nicht von Relevanz.

2.2 Bildsteganografie

Bilddateien sind auf Grund ihrer weiten Verbreitung und der damit verbundenen großen Verfügbarkeit sehr beliebte Trägerobjekte für die Steganografie. Da es für die unterschiedlichsten Anwendungsbereiche zahlreiche Bilddateiformate gibt, existieren auch diverse Ansätze steganografischer Algorithmen, um geheime Informationen in Bildern einzubetten.

Im Gegensatz zu Audio- oder Videodaten werden Bilder am ehesten selbst hergestellt und sind damit individueller als andere potentielle Träger. Dies können beispielsweise Urlaubsbilder oder selbst erstellte Desktophintergründe sein. Obgleich auf den Rechnern dieser Welt sicherlich auch zahlreiche Audio-/Videodateien gespeichert sein mögen, so liegt es dennoch nahe, dass es sich dabei oftmals auch um Raubkopien handeln könnte, denen genau dieses individuelle Merkmal fehlt. Trifft man auf ein mögliches Steganogramm, das eine solche Datei als Träger nutzt, bestünde eine erhöhte Chance der Detektierbarkeit. Durch einen Vergleich mit dem Original, das sich über das World Wide Web beschaffen ließe, könnte man zumindest die Tatsache einer Manipulation feststellen und würde ein potentielles Steganogramm sodann näher unter die Lupe nehmen.

2.2.1 Least Significant Bit

Durch Veränderung des Least Significant Bit (LSB) lassen sich auf eine sehr einfache Weise Informationen in eine Bilddatei einbetten. Die Methode eignet sich besonders für Rastergrafiken, wie Windows Bitmaps (BMP) oder Portable Network Graphics (PNG), indem das am wenigsten bedeutsame Bit entsprechend der geheimen Nachricht oder des Datenstroms angepasst wird.

Bei einem Bild mit einer Farbtiefe von 24 Bit werden für alle drei Farbkanäle (rot, grün und blau) je 8 Bit zur Darstellung der jeweiligen Farbkanäle genutzt. In einem einzigen Pixel lassen sich somit bereits 3 Bit an Informationen verstecken; die Veränderung zum Original ist dabei jedoch ausgesprochen gering. [4]

Die nachfolgende Abbildung zeigt eine 8-Bit-Sequenz, die beispielsweise den blauen Farbkanal eines Pixels mit einem Helligkeitswert von 149 darstellen könnte. Wird nun das LSB von 1 auf 0 geändert, wird der Blauanteil des Pixels nur geringfügig dunkler.

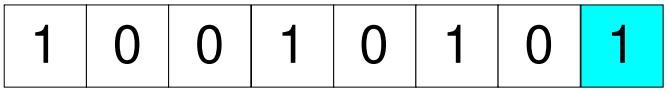

Abbildung 2.1: Ein Byte (Dezimalwert 149) mit hervorgehobenem Least Significant Bit (Quelle: `https://commons.wikimedia.org/wiki/File:Least_significant_bit.svg`)

Es besteht auch die Möglichkeit zwei oder mehrere der weniger bedeutsamen Bits zu manipulieren um eine größere Kapazität für den Payload zu erreichen; allerdings steigt damit auch die optische Wahrnehmbarkeit, die sich durch Artefaktbildungen bemerkbar macht. Beschränkt sich die Änderung auf lediglich ein Bit pro Pixelwert, hat das “Ziel-Bit“ statistisch gesehen mit einer 50%-igen Wahrscheinlichkeit bereits den richtigen Wert; bei zwei Bits liegt die Wahrscheinlichkeit immer noch bei 25%. Folgerichtig muss sich durch die Manipulation nicht zwangsläufig jeder Originalwert verändern. [4]

Am Beispiel eines Desktop-Hintergrundbildes, das in einer Auflösung von 2560 x 1440 Pixeln und einer Farbtiefe von 24 Bit vorliegt, gibt es rechnerisch 11.059.200 dieser “Least Significant Bits“, womit sich sich ca. 1,3 MB an Daten verstecken lassen. Für jeden der drei Farbkanäle stehen 256 mögliche Farbintensitäten zur Verfügung. Da mit dem letzten Bit die Farbintensität lediglich um eins erhöht bzw. verringert wird, beträgt die Veränderung gegenüber dem Originalbild weniger als 1 % und ist für das menschliche Auge nicht sichtbar. Werden zwei Bits zur Einbettung der Daten angepasst, so beträgt die Veränderung weniger als 2 % und ist auch bei genauerer Betrachtung kaum wahrnehmbar.

Nachteilig an diesem Verfahren ist die Tatsache, dass sich grundsätzlich jede Rastergrafik hinsichtlich der LSB-Technik untersuchen lässt, indem über die einzelnen Pixelwerte iteriert wird. Versteckte Daten ließen sich auf diese Weise also recht einfach entdecken. Aus diesem Grund sollte die Daten zunächst kodiert oder verschlüsselt und erst danach in das Steganogramm überführt werden. Darüber hinaus wird man auf diese Weise auch dem Kerckhoffs’schen Prinzip gerecht.

2.2.2 Diskrete Kosinustransformation

Die Modifikation des “Least Significant Bit“ eignet sich grundsätzlich nur für Formate, die durchgehend mit verlustfreier Kompression arbeiten. Um eine größere Kompressionsrate zu erreichen, werden gegenüber dem unkomprimierten Original auch geringfügige optische Veränderungen in Kauf genommen. Dies spart letztlich Speicherplatz und ggf. auch Übertragungsbandbreite. Bei verlustbehafteter Kompression fallen selbiger also gerade die am wenigsten bedeutsamen Bits zum Opfer in denen Informationen versteckt werden könnten.

Die Joint Photographic Experts Group stellte 1992 die *Norm ISO/IEC 10918-1* vor und beschreibt darin verschiedene Methoden der Bildkompression. Aus dieser Norm ging auch das weit verbreitete Bildformat “JPEG“ hervor. Es wird vor allem von Digitalkameras bzw. Fotohandys oder auf Internetauftritten mit verlustbehafteter Kompression genutzt, da dieser Modus den geringeren Speicherbedarf benötigt. Ein verlustfreier Modus ist zwar ebenfalls in der Norm definiert, aber vergleichsweise selten anzutreffen.

Der in der *Norm ISO/IEC 10918-1* definierte JPEG-Kompressionsalgorithmus durchläuft die nachfolgenden Stufen: [7]

- Zunächst werden die Daten vom RGB-Farbraum in das YCbCr-Farbmodell umgerechnet. Hintergrund ist die Tatsache, dass das menschliche Auge Helligkeitsunterschiede viel eher wahrnimmt, als Farbunterschiede.
- Eine erste Kompression wird durch eine Unterabtastung hinsichtlich Farbabweichung der Farbkanäle Cb (Blue-Yellow Chrominance) und Cr (Red-Green Chrominance) von der Grundhelligkeit Y (Luminance) erreicht, wodurch lediglich Differenzwerte gespeichert werden müssen.
- Im Anschluss folgt eine Aufteilung des Bildes in Blöcke mit einer Größe von 8x8 Pixeln. Auf diese Blöcke wird sodann eine zweidimensionale diskrete Kosinustransformation angewandt, was der Aufbereitung der Daten für den Folgeschritt dient.
- In der Quantisierungsphase werden die Koeffizienten aus der DCT durch eine Quantisierungsmatrix geteilt, wobei die Ergebnisse auf die nächstliegende Ganzzahl gerundet werden. Diese Maßnahme nimmt zu Gunsten des eingesparten Speicherplatzes geringfügige Rundungsfehler in Kauf.
- Da die 64 Werte des 8x8 Musters von oben links nach unten rechts abnehmen, werden sie durch einen sogenannten “zig zag scan“ umsortiert, um eine optimale Anordnung für die abschließende Entropiekodierung zu erreichen.

Auch wenn JPEG hauptsächlich mit verlustbehafteter Kompression arbeitet, ließe sich Bildsteganografie auch mit derartigen Trägermedien durchführen. Die Kompression ist nämlich nur bis einschließlich der diskreten Kosinustransformation durch Rundungsfehler verlustbehaftet. Dadurch eignet sie sich auch nicht direkt für das Verstecken einer Nachricht, da die - wenn auch nur geringen - Abweichungen sie zerstören könnten. Die nachfolgende Entropiekodierung, für die meist die Huffman-Kodierung verwendet wird, arbeitet verlustfrei. Zwischen Quantisierungsphase und Entropiekodierung könnte auf die Technik des “Least Significant Bit“ zurückgegriffen werden, um die geheimen Informationen zu verstecken. [4]

Auch wenn heutzutage Speicherplatz und Bandbreite in diesem Maße keine Rolle mehr spielen dürfte, so war dies vor knapp 30 Jahren durchaus noch von Belang und starke JPEG-Kompression sehr willkommen. Der grundlegende Vorteil kleinerer Dateien ist für die Steganografie allerings eher ein Nachteil, da dies auch zu einer wesentlich geringeren Payload-Kapazität führt.

Bei JPEG ist die Steganografie im Allgemeinen schwerer detektierbar, da nicht einzelne Pixelwerte gespeichert sind, die dann verändert werden, sondern das Bild erst beim Öffnen aus den gespeicherten Daten berechnet wird. Die Steganografie findet bei JPEG also im Transformationsraum *(transform domain)* statt und nicht direkt im Bildraum *(spatial domain)*, wie bei Rastergrafiken.

2.2.3 Modifikation der Farbpalette

Im Gegensatz zu den bereits angesprochenen Methoden verfolgt das Graphics Interchange Format (GIF) abermals einen anderen Ansatz. Die Darstellung von Bildinhalten erfolgt hierbei durch Referenzierung über eine Farbpalette. Per Definition steht eine maximale Farbtiefe von 8 Bit zu Verfügung, womit max. 256 unterschiedliche Farben (inklusive einer “Transparenzfarbe“) dargestellt werden können. Ein Pixel wird, dem jeweiligen Wert in der Farbpalette entsprechend, nunmehr über ein einzelnes Byte referenziert. Um trotz der reduzierten Farben eine realistischere Darstellung zu ermöglichen, lässt sich die Standardfarbtabelle auch auf den jeweiligen Bildinhalt anpassen, wenn dieser beispielsweise vermehrt Rottöne enthält. Insgesamt stehen dafür ca. 16,7 Millionen mögliche Farbwerte zur Verfügung. [5]

Typischerweise werden die Farbwerte innerhalb der Tabelle absteigend nach ihrer Häufigkeit sortiert, um die zum Nachschlagen benötigte Zeit so gering wie möglich zu halten. Dies hat allerdings zur Folge, dass benachbarte Werte in der Farbtabelle nicht zwangsläufig auch Nachbarn im Farbspektrum sein müssen. Folgt beispielsweise die Farbe Rot unmittelbar auf die Farbe Blau, wird dies bei Anwendung der LSB-Technik zu gravierenden Änderungen im Bild führen. Die LSB-Technik kann also bei GIFs nicht pauschal angewandt werden. Damit der Farbunterschied nicht auffällt, dürften benachbarte Farbeinträge nur minimal voneinander abweichen und müssten demnach nicht nach Häufigkeit, sondern nach Ähnlichkeit angeordnet werden.

Als Alternative könnte man somit auf eine benutzerdefinierte Farbtabelle zurückgreifen. Befinden sich in der Standard-Farbpalette ungenutzte Farbwerte, ließen sich diese durch ähnliche Werte verwendeter Farben ersetzen; der eigentliche Bildinhalt würde dadurch nicht verändert. Diese Paare ähnlicher oder gar gleichen Farben könnten sodann für die Steganografie genutzt werden. Dies setzt allerdings voraus, dass das Bild ohnehin weniger als die 256 zur Verfügung stehenden Farbwerte benutzt. Zudem müsste vorab geprüft werden, ob sich das Bild überhaupt als Trägerdatei eignet. Wenn nicht auf die Standard-Farbpalette zurückgegriffen wird, lassen sich Manipulationen zudem verhältnismäßig leicht detektieren. [6]

Bei Verwendung eines Graustufenbildes fällt das “Farbpaletten-Problem“ nicht so sehr ins Gewicht, da das menschliche Auge zwischen 256 verschiedenen Graustufen weitaus schwieriger unterscheiden kann, wie bei einem Farbbild. [4].

Da bei GIFs ein Pixel grundsätzlich durch ein einziges Byte referenziert wird und bei Rastergrafiken mit RGB-Farbraum für jede der drei Farben ein Byte zur Verfügung steht, ist die maximale Kapazität für den Payload bei GIFs folglich deutlich geringer.

2.3 Datenträgersteganografie

Ähnlich wie es bei Bilddateien Bits gibt, deren Informationsgehalt für die Darstellung der Inhalte als eher gering anzusehen ist, finden sich auch auf Datenträgern Bereiche, die selten oder gar nicht genutzt werden. Bei der Datenträgersteganografie gibt es eine ganze Menge von Möglichkeiten Informationen zu verbergen. Hierzu sollten jedoch zunächst einige allgemeine Begebenheiten hinsichtlich der Datenspeicherung bei modernen Computersystemen betrachtet werden.

Würde der gesamte Speicher auf Betriebssystemebene bitweise adressiert, bestünde keine Möglichkeit geheime Informationen auf Datenträgern zu verstecken. Datenträger werden aus Effizienzgründen allerdings über abstrakte Ebenen adressiert, wodurch erst Möglichkeiten geschaffen werden, bestimmte Bereiche als Datenverstecke nutzen zu können. Einige dieser Bereiche sind dabei nicht ohne weiteres zugänglich. Im Grunde genommen ist die Datenträger- bzw. Dateisystemsteganografie vielmehr nur ein Nebenprodukt der Systemarchitektur.

Obwohl es neben Festplatten auch diverse weitere Massenspeicher wie Flashmedien oder optische Datenträger gibt, soll im Folgenden nur auf klassische Magnetfestplatten eingegangen werden, da sie sich grundlegend hinsichtlich der Speicherung von Daten ähneln.
Zunächst weist jede Festplatte eine grundlegende geometrische Struktur auf. Innerhalb dieser Struktur befinden sich eine oder mehrere Partitionen, die ihrerseits wiederum ein Dateisystem beinhalten. Die eigentlichen Dateien werden dort abgelegt und der Nutzer kann auf deren Inhalt letztendlich zugreifen. Neben dem eigentlichen Inhalt besitzen Dateien weitere Parameter, die sie genau spezifizieren; sie werden Metadaten genannt. Grundsätzlich lassen sich auf all diesen Ebenen Informationen verstecken. [8]

2.3.1 Physische Datenträgerstruktur

Bei einigen Datenträgern existieren reservierte Bereiche, die als **Host Protected Area** (HPA) und **Device Configuration Overlay** (DCO) bezeichnet werden. In diesen reservierten Bereichen können Computerhersteller speziele Daten ablegen, wie beispielsweise Images zur Systemwiederherstellung in Form eines “Werksreset“ oder allgemeine Diagnosetools. Des Weiteren können Datenträger unterschiedlicher Hersteller dadurch auf eine Einheitsgröße genormt werden, wenn diese produktionsbedingt unterschiedliche Kapazitäten aufweisen. Das Betriebssystem kann nicht unmittelbar auf diese Daten zugreifen, womit ein versehentliches Löschen oder Verändern durch den Nutzer verhindert wird. Mit speziellen Tools, die die entsprechenden Befehle beherrschen müssen, ist ein Zugriff dennoch möglich. Durch Verringerung der Gesamtkapazität eines geeigneten Datenträgers lässt sich mithilfe eines solchen Tools Speicherplatz für ein Datenversteck schaffen. [9]

2.3.2 Virtuelle Datenträgerstruktur

Wie bereits erwähnt, wird Festplattenspeicher von modernen Betriebssystemen nicht bitweise adressiert; er wird vorab in virtuelle Bereiche eingeteilt - die Partitionen. Je nach verwendetem Betriebssystem gibt es ein oder mehrere solcher virtuellen Datenträgerstrukturen. Partitionen sind eine Sequenz von Sektoren, die später mit einem Dateisystem initialisiert werden.

Exemplarisch wird im Folgenden auf das MBR-Partitionsschema eingegangen. Am Anfang eines Datenträgers befindet sich einen Bereich, der als Master Boot Record (MBR) bezeichnet wird. Neben dem zum Starten des Betriebssystems nötigen initialen Boot-Code befindet sich hier auch die Partitionstabelle, wo sich Informationen zu bis zu vier primären Partitionen wiederfinden. Der MBR ist genau einen Sektor (512 Byte) groß. Da Partitionen aus Performancegründen allerdings an Zylindergrenzen ausgerichtet werden sollen, entsteht ein ungenutzter **Bereich nach dem Master Boot Record**, wo sich Daten verstecken lassen. Die Größe dieses Bereiches ist zwar variabel, er umfasst jedoch meist 2047 Sektoren. [8]

Beim Anlegen oder Löschen von Partitionen kann es vorkommen, dass letztendlich nicht der gesamte zur Verfügung stehende Speicherplatz eines Datenträgers beansprucht wird. Ein als **Volume Slack** bezeichneter ungenutzter Bereich kann sich vor der ersten, hinter der letzten oder auch zwischen Partitionen befinden. Da das Betriebssystem ein Dateisystem benötigt, kann es auf diesen Bereichen keine Dateien ablegen. Es kann also im herkömmlichen Sinne nicht auf diese Volume Slacks zugegriffen werden.
Erstellt man beispielsweise eine zweite Partition, schreibt seine Daten in das dortige Dateisystem hinein und löscht die Partition im Anschluss wieder, wären die einzelnen Dateien zwar noch vorhanden, sind aber vom Betriebssystem nicht mehr lesbar. [8]

2.3.3 Partitionsstruktur

Um dem Betriebssystem das Abspeichern von Dateien zu ermöglichen, müssen Partitionen mit einem Dateisystem initialisiert werden. Dabei werden mehrere Sektoren zu Blöcken zusammengefasst, die ihrerseits durch das Dateisystem adressiert werden können; dies geschieht abermals aus Gründen der Effizienz. Ist die Anzahl der Sektoren nicht durch die Blockgröße teilbar, so bleiben am Ende Sektoren übrig, die nicht mehr zur Bildung eines Blocks ausreichen. In einem solchen **Partition Slack** lassen sich ebenfalls Daten verstecken. [8]

Grundsätzlich beginnt jede Partition mit einen eigenen **Bootsektor**. Ist bei einer Partition das “bootable“ Flag zur Markierung der Startfähigkeit nicht gesetzt, wird sie vermutlich auch keinen Bootcode enthalten; ein weiteres kleines Datenversteck. [8]

2.3.4 Dateisystemstruktur

Enthält ein Dateisystem weniger Daten, als Speicherplatz zur Verfügung steht, gibt es logischerweise Bereiche, wo sich keine Dateien befinden. Diese **nicht allokierten Bereiche** werden vom Betriebssystem hinsichtlich des Vorhandenseins von Daten ignoriert. Solange das Betriebssystem an ebendieser Stelle keine neuen Dateien ablegt, ließen sich auch dort Daten verstecken. [8]

Um dieser Gefahr des Überschreibens aus dem Weg zu gehen, können durch Manipulation am Dateisystem **"fake bad sectors"** imitiert werden. Dabei wird die Fähigkeit vieler Dateisysteme ausgenutzt, defekte Blöcke zu markieren, wenn beispielsweise ein darunter liegender Sektor beschädigt ist. Diese Blöcke werden fortan vom Betriebssystem ignoriert und können so zum Datenversteck umfunktioniert werden, da die darunterliegenden Sektoren in Wirklichkeit nicht beschädigt sind. [8]

Wie bereits geschildert, fassen viele Dateisysteme mehrere Sektoren zu Blöcken zusammen. Ist eine Datei kleiner als der Speicherplatz, den ein Block zu Verfügung stellt, muss das Betriebssystem den letzten genutzten Sektor mit Daten auffüllen. Bei modernen Betriebssystemen geschieht dies durch ein Padding mit Nullen über den Rest des Sektors. Solange die abgelegte Datei nicht verändert wird, bleiben die übrigen Sektoren des Blocks ungenutzt und werden als **File Slack** bezeichnet. Da früher noch Zufallsdaten aus dem Arbeitsspeicher für das Padding verwendet wurden, ist auch die historische Bezeichnung *RAM-Slack* gebräuchlich. Durch geeignetes Anlegen von sehr kleinen Dateien lassen sich File Slacks auch absichtlich erzeugen, die dann zur Ablage geheimer Informationen genutzt werden können. [8]

2.4 Gegenüberstellung der steganografischen Verfahren

Für das anvisierte Ziel der Plausiblen Abstreitbarkeit mittels Steganografie und Verschlüsselung, wurden diverse steganografische Methoden ausgewählt und genauer betrachtet, um geeignete Verfahren zu finden, die letztendlich in den Prototyp der Software Einzug erhalten werden.

Von Grund auf haben alle steganografischen Verfahren ein Kosten-Nutzen-Problem. Je mehr geheime Informationen versteckt werden sollen, desto mehr Speicherplatz muss auch von den Trägermedien zur Verfügung gestellt werden. Die meisten Verfahren lassen sich hinsichtlich ihrer "steganografischen Intensität" justieren. Am Beispiel der LSB-Technik kann man statt einem einzelnen Bit auch zwei oder mehr als Datenversteck verwenden. Dadurch kann der gewünschte Speicherplatz zwar relativ einfach erhöht werden, es steigt jedoch auch das Risiko einer Detektion.

Die Detektierbarkeit ist jedoch ebenfalls ein Aspekt der nicht vernachlässigt werden darf. Um eine möglichst geringe Entdeckungsrate zu gewährleisten aber dennoch möglichst viele Informationen einzubetten, werden entsprechend große Trägerdateien benötigt.

Die Wahrnehmbarkeit einer Manipulation wächst grundsätzlich mit der zunehmenden Veränderung der Originaldaten. In einer der Arbeit vorausgegangenen Untersuchung wurden audiosteganografische Verfahren miteinander verglichen. Im Gegensatz zu Bildern kommt die Wahrnehmung eines “Verrauschens“ gerade bei Audiosamples sehr viel schneller zum Tragen, da die kognitiven Eigenschaften des menschlichen Gehörs weitaus empfindlicher sind; die visuelle Wahrnehmung ist hier weitaus toleranter. Ferner kann angenommen werden, dass der durchschnittliche Computernutzer weitaus weniger Audiofiles als Bilddateien besitzt. Aus diesen Gründen wurde die Bildsteganografie der Audiosteganografie vorgezogen und auf letztere nicht gesondert eingegangen.

Ein weiterer Aspekt ist die Individualität der Trägerdaten, da jegliche Form der Steganografie bei Vorhandensein des Originals sehr leicht detektiert werden kann. Wurde die Steganografie an einem über das Internet heruntergeladenen Bild durchgeführt, so wird bereits der Vergleich des Hashwertes mit dem des Originals eine Manipulation ans Tageslicht bringen; eine steganalytische Untersuchung könnte die Folge sein. Die Gefahr, dass persönliche Urlaubsfotos reihenweise über das Internet eingesehen werden können - und damit die Originale für einen direkten Vergleich greifbar sind - ist doch recht gering; es sei denn man teilt sein Leben mit der ganzen Welt auf Instagram.

Abschließend wurden die drei ausgewählten bildsteganografischen Verfahren hinsichtlich der Wahrnehmbarkeit von Manipulationen, zu erwartender Kapazität für den payload, allgemeiner Detektierbarkeit, Robustheit gegenüber Veränderungen und Komplexität in der Umsetztung gegenübergestellt, wie die nachstehende Tabelle zeigt.

Tabelle 2.1: Vergleich ausgewählter bildsteganografischer Verfahren

Träger	Sichtbarkeit	Kapazität	Detektierbarkeit	Robustheit	Komplexität
PNG	niedrig	hoch	hoch	niedrig	niedrig
JPEG	niedrig	mittel	niedrig	mittel	mittel
GIF	niedrig	mittel	mittel	niedrig	niedrig

Die LSB-Technik lässt sich recht leicht bei Rastergrafiken wie PNG umsetzen. Sie benötigen zwar auch grundsätzlich mehr Speicherplatz als ein JPEG oder GIF, bieten im Gegenzug aber auch die höchste Payload-Kapazität. Da die Daten ohnehin lokal vorgehalten werden, spiele der benötigte Speicherplatz auch eine untergeordnete Rolle. Ebenso verhält es sich hinsichtlich der Robustheit, weshalb sie vernachlässigt werden kann. Da sich die LSB-Technik am leichtesten implementieren lässt und den größten Speicher zur Verfügung stellt, wurde sie letztendlich für das Projekt ausgewählt.

Da auch größere Datenmengen versteckt werden sollen, reicht reine Bildsteganografie allein für den angestrebten Zweck nicht aus. Wegen des Speicherplatzbedarfs muss deshalb auch auf diverse Formen der Datenträgersteganografie zurückgegriffen werden. Hierzu sind die beschriebenen Verfahren nochmals tabellarisch gegenübergestellt.

Tabelle 2.2: Vergleich ausgewählter datenträgersteganografischer Verfahren

Träger	Sichtbarkeit	Kapazität	Detektierbarkeit	Robustheit	Komplexität
HPA/DCO	niedrig	hoch	niedrig	hoch	hoch
MBR	mittel	niedrig	hoch	niedrig	niedrig
Volume Slack	niedrig	hoch	hoch	niedrig	niedrig
Partition Slack	niedrig	niedrig	niedrig	hoch	niedrig
Bootsector	mittel	niedrig	hoch	niedrig	niedrig
freier Speicher	niedrig	hoch	niedrig	niedrig	mittel
fake bad sectors	niedrig	hoch	niedrig	hoch	hoch
File Slack	niedrig	niedrig	niedrig	niedrig	mittel

Sowohl Host Protected Area (HPA) als auch Device Configuration Overlay (DCO) eignen sich zweifelsohne zum Verbergen großer Datenmengen, es besteht dagegen allerdings auch ein hohes Risiko, den Datenträger bei unsachgemäßer Bedienung unbrauchbar zu machen. Auch wenn die geheimen Informationen dadurch vermutlich auch nicht mehr detektierbar sind, ist dies sicherlich nicht im Sinne des Erfinders. Letztlich sind auch nicht alle Datenträger dafür geeignet.

Partition Slacks oder aber auch die Bootsektoren der Partitionen eignen sich hervorragend zum verstecken von geheimen Informationen. Da auf diese Bereiche in der Regel niemals zugegriffen wird, ist die Gefahr eines Überschreibens quasi gleich Null. Leider bieten sie nur begrenzten Speicherplatz, eigen sich aber bestens zur Ablage von kryptografischen Schlüsseln.

Zur Ablage großer Datenmengen wären Volume Slacks (einschließlich dem Bereich nach dem Master Boot Record) dagegen die bessere Wahl. Ein solcher Bereich muss nicht zwangsweise negativ auffallen, solange sich dort keine Klartextdaten wiederfinden. Dies lässt sich durch vorherige Verschlüsselung der Daten recht einfach umsetzen.

Innerhalb bestehender Dateisysteme wird es dagegen schon recht viel schwieriger die Steganografie erfolgreich umzusetzen. Die Verwendung nicht allokierter Bereiche birgt immer die Gefahr, dass dort abgelegte Informationen überschrieben werden. Gleiches gilt für File Slacks, die darüber hinaus eher geringen Speicherplatz bieten. “Fake Bad Sectors“ haben diese Probleme zwar nicht, die sind aber je nach Dateisystem auch weitaus aufwändiger zu implementieren. Um schwer detektierbar zu bleiben, dürften sie dabei schließlich auch keinem regelmäßigen Muster folgen.

Dem aufmerksamen Leser wird nicht entgangen sein, dass bereits zu Anfang dieser Arbeit von der Manipulation des Swaps die Rede war. Aufgrund der Beschaffenheit des Swap-“Dateisystems“ wird hier strenggenommen eine Kombination aus Partition Slack und nicht allokiertem Speicher umgesetzt. Durch Manipulation des Swap-Headers kann die Größe des zur Verfügung stehenden Speicherbereiches verringert werden, ohne dass die darunterliegende Partition verändert wird. Es entsteht dadurch sozusagen ein “provozierter“ Partition Slack. Die Sektoren sind dabei nicht als defekt markiert, es kann beim Swapping einfach nicht auf sie zugegriffen werden, da sie außerhalb des definierten Bereiches liegen; aus Sicht des Swap existieren sie gar nicht.

Durch diese Methode können die in der Tabelle ersichtlichen Vorteile von Partition Slacks (hohe Robustheit, niedrige Komplexität) mit denen des nicht allokierten Speichers (hohe Kapazität) kombiniert werden. Hinsichtlich Sichtbarkeit und Detektierbarkeit können beide Verfahren als hinreichend gut betrachtet werden, da ohnehin nur verschlüsselte Daten gespeichert werden, allerdings ohne Hinweise auf den dabei verwendeten Algorithmus. Diese Metadaten werden auf geeignete Weise anderweitig abgelegt.

Da bei der Installation gängiger Linuxderivate grundsätzlich das Anlegen einer Swap-Partition empfohlen wird, ist deren Vorhandensein ebenfalls unverdächtig. Der Swap dient grundlegend zur Auslagerung von Daten aus dem Arbeitsspeicher, in dem sich hauptsächlich Binärdaten wiederfinden. Es wäre folglich kein zwingendes Indiz für ein Datenversteck, nur weil dort auch “kryptisch wirkende“ Daten zu finden sind. Ob Swap heutzutage allerdings noch so relevant ist wie in den 1990ern - als das Betriebssystem Linux noch in den Kinderschuhen stand und Arbeitsspeicher noch nicht in rauen Mengen verfügbar war - darüber streiten sich die Geister. Die Größe des Swap ist damit eher zweitrangig, sollte man auf die Funktion des “Ruhezustand“ bzw. “Suspend-to-Disk“ verzichten können. Einer Verkleinerung des Swaps zur Erstellung eines Datenverstecks steht damit nichts im Wege.

Zusammenfassend besteht der steganografische Teil der Software also letztlich aus einer Kombination von Bildsteganografie durch Nutzung der LSB-Technik, die sich mit adäquatem Aufwand implementieren lässt und vor allem mit einem guten Kosten-/Nutzen-Verhältnis punktet und einer hybriden Form der Datenträgersteganografie, um möglichst große Datenmengen verarbeiten zu können. Es kann an dieser Stelle bereits vorweggenommen werden, dass auch eine rein datenträgersteganografische Variante implementiert wird.

3 Verschlüsselungsverfahren

Um das Kerckhoffs'sche Prinzip zu wahren reicht Steganografie allein noch nicht aus, da deren Sicherheit hauptsächlich auf der Geheimhaltung des genutzten Algorithmus aufbaut. Sollte es einem Angreifer gelingen sämtliche steganografische Verfahren auszuhebeln, wären die Daten zugänglich. Sie bedürfen daher einer vorherigen Verschlüsselung. Auf mögliche zur Anwendung kommende Verfahren wird im diesem Kapitel eingegangen.

3.1 Allgemeines

Hinsichtlich der Verschlüsselung selbst ist grundlegend zwischen symmetrischen und asymmetrischen Verfahren zu unterscheiden. [10]

Symmetrische Verfahren nutzen für die Ver- und Entschlüsselung der Daten den selben Schlüssel. Dies setzt voraus, dass der Schlüssel allen an der Konversation beteiligten Personen gekannt sein muss. Möchte man beispielsweise Nachrichten über unsichere Kanäle austauschen und greift dabei auf symmetrische Verschlüsselung zurück, so muss der Schlüssel vorab über einen sicheren Kanal ausgetauscht werden. 3DES (Triple Data Encryption Standard) , AES (Advanced Encryption Standard) oder Twofish sind Vertreter moderner symmetrischer Verschlüsselungsverfahren.

Wie die nachfolgende Abbildung zeigt, chiffriert der Sender die Nachricht mit dem gemeinsamen Schlüssel und versendet sodann den Geheimtext. Der Empfänger dechiffriert mit dem selben Schlüssel und erhält die Nachricht sodann wieder im Klartext.

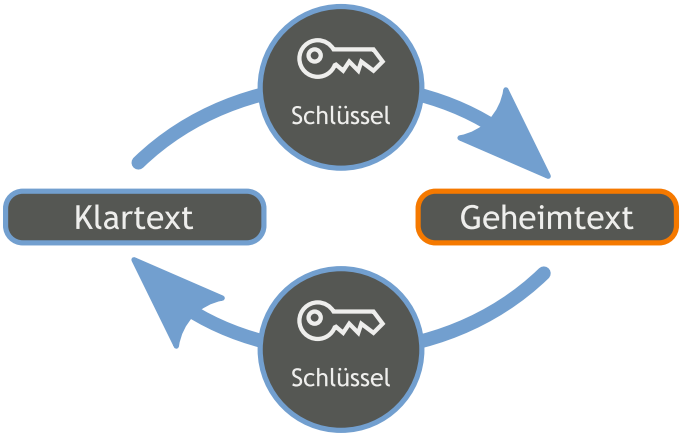

Abbildung 3.1: Symetrische Verschlüsselung
(Quelle: `https://commons.wikimedia.org/wiki/Category:Orange_blue_cryptography_diagrams`)

Asymmetrische Verfahren oder auch Public-Key-Verfahren haben das Problem eines vorherigen Schlüsseltausches nicht. Möchte man dieses Verfahren zum Versenden von Nachrichten über unsichere Kanäle nutzen, benötigt der Empfänger ein Schlüsselpaar, bestehend aus einem öffentlichen und einem privaten Schüssel. Als moderne asymmetrische Verschlüsselungsverfahren seien beispielsweise Rabin, Elgamal oder das RSA-Kryptosystem zu erwähnen.

Der *öffentliche Schlüssel* (public key) wird dabei zum Verschlüsseln einer Nachricht genutzt; er kann bzw. sollte - wie der Name es schon vermuten lässt - öffentlich zugänglich sein. Der *private Schlüssel* (private key) wird hingegen dazu verwendet, eine mit dem dazugehörigen öffentlichen Schlüssel chiffrierte Nachricht wieder lesbar zu machen; er ist unbedingt geheim zu halten. Diese Vorgehensweise ist in der folgenden Abbildung nochmals visuell dargestellt.

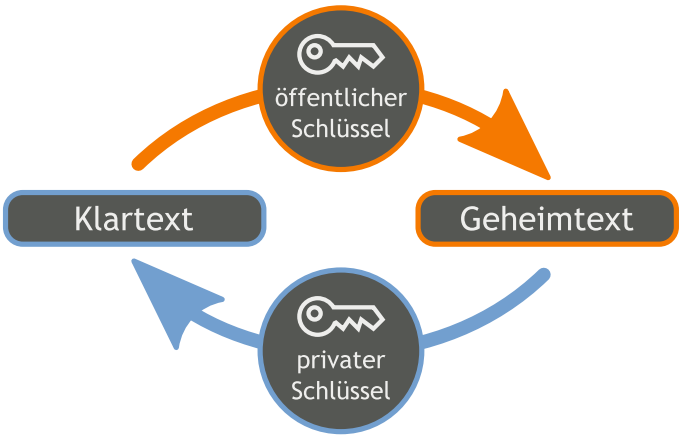

Abbildung 3.2: Asymetrische Verschlüsselung
(Quelle: `https://commons.wikimedia.org/wiki/Category:Orange_blue_cryptography_diagrams`)

Für beide Verfahren gilt hinsichtlich der Verschlüsselung, dass die Sicherheit nicht auf der Geheimhaltung des verwendeten Algorithmus beruht, sondern auf der Geheimhaltung des Schlüssels. Da auf ihm die gesamte Sicherheit aufbaut, muss er folgerichtig hinreichende Komplexität besitzen, d. h. er muss eine entsprechende Schlüssellänge ausweisen. Würde die Schlüssellänge zu kurz gewählt, wäre es in überschaubarer Zeit möglich, alle Kombinationen durchzuprobieren und damit den richtigen Schlüssel zu finden. Vom Bundesamt für Sicherheit in der Informationstechnik werden derzeit Schlüssellängen von mindestens 128 Bit für symmetrische bzw. 2048 Bit für asymmetrische Verfahren empfohlen, um einen solchen Brute-Force-Angriff zu verhindern.
Grundsätzlich lässt sich jeder beliebige Datenstrom verschlüsseln, da er stets aus einer Aneinanderreihung einzelner Bits besteht. Bezogen auf die Datenverschlüsselung moderner Betriebssysteme gibt es hierbei diverse Ansätze. Chiffrierbar sind einzelne Dateien bishin zu ganzen Blockgeräten, wie Festplatten bzw. deren Partitionen.

3.2 Verschlüsselung auf Dateiebene

Die Verschlüsselung einzelner Dateien ist die einfachste Variante, Daten zu chiffrieren; hierbei wird oftmals auch der englische Terminus **"Single File Encryption"** verwendet. Im Folgenden wird die Verwendung symmetrischer Verschlüsselung anhand der beiden weit verbreiteten Verfahren GnuPG bzw. OpenSSL an einem Beispiel veranschaulicht. Beide Varianten können zum Ver- bzw. Entschlüsseln der Daten auf ein symmetrisches Verfahren zurückgreifen, da dies im Gegensatz zu asymmetrischen wesentlich performanter arbeitet. Sowohl bei der Ver- als auch bei der Entschlüsselung wird der Nutzer interaktiv zur Eingabe des Kennwortes aufgefordert.
Der Vorteil von Single File Encryption ist die sehr einfache Handhabung, wobei ein Datenstrom ohne Zwischenschritte direkt in sein "chiffriertes Pendant" überführt werden kann. Zu beachten ist allerdings, dass neben dem Chiffrat auch die Originaldatei im Dateisystem erhalten bleibt, die sodann gesondert gelöscht werden muss. Man muss sich diesbezüglich auch im Klaren sein, dass damit auch deren Metadaten in Form von Zeitstempeln verloren gehen. Bei Entschlüsselung des Chiffrats wird stets eine neue Datei erstellt - und damit auch neue Metadaten. Das Original wird also nicht wiederhergestellt; es entsteht lediglich eine inhaltsgleiche Datei.

3.2.1 GnuPG

Mit GNU Privacy Guard wurde der OpenPGP-Standard vollständig und kostenlos implementiert; es lassen sich Daten und Mitteilungen sowohl verschlüsseln als auch signieren. Damit kann jeder elektronische Daten unter der Verwendung kryptografischer Methoden mittels GnuPG vertraulich übermitteln. Obgleich GnuPG in erster Linie als Public-Key-Verschlüsselungsverfahren genutzt wird, lassen sich Daten auch symmetrisch verschlüsseln. Aufgrund ihrer höheren Performanz eignen sich symmetrische Verfahren sehr gut für große Datenmengen. Zudem müssen hierbei in der Regel vorab auch keine Schlüssel bzw. Passwörter zwischen verschiedenen Kommunikationspartnern ausgetauscht werden, wie dies in der E-Mail-Kommunikation üblich wäre. GnuPG leitet den verwendeten Schlüssel von einer Passphrase ab, die beim Verschlüsseln des Dokuments vom Benutzer eingegeben wird. [11]

Verwendung von GnuPG:
Verschlüsselung: `$ gpg --output ENCRYPTED --symmetric FILE`
Entschlüsselung: `$ gpg --output DECRYPTED --decrypt ENCRYPTED`

Um ein Dokument symmetrisch zu verschlüsseln, wird die Option `--symmetric` bzw. `-c` verwendet. Über die Option `--output` kann eine Zieldatei bestimmt werden; andernfalls erstellt GnuPG eine Datei mit dem selben Namen, ergänzt durch die Dateiendung `.gpg`. Für die Entschlüsselung wird respektive die Option `--decrypt` bzw. `-d` genutzt. Die Ausgabe erfolgt hier standardmäßig auf der Kommandozeile; über die Option `--output` kann jedoch ebenso eine Zieldatei bestimmt werden.

3.2.2 OpenSSL

Bei OpenSSL handelt es sich um ein umfassendes Toolkit für die Protokolle *Transport Layer Security (TLS)* und *Secure Sockets Layer (SSL)*, womit es hauptsächlich zum verschlüsselten Übertragen von Daten in Netzwerken verwendet wird. OpenSSL ist aber auch eine allgemeine Kryptografiebibliothek, die zahlreiche symmetrische Verschlüsselungsverfahren anbietet. Ebenso wie GnuPG ist OpenSSL in der Grundkonfiguration vieler Linux-Distributionen bereits mit an Bord.

Verwendung von OpenSSL:
Verschlüsselung: `$ openssl enc -e -aes256 -in FILE -out ENCRYPTED`
Entschlüsselung: `$ openssl enc -d -aes256 -in ENCRYPTED -out DECRYPTED`

Durch den Parameter `enc` wird openssl zur Verwendung des Kryptografiemodus auf der Kommandozeile angewiesen. Zur Verschlüsselung wird die Option `-e` gesetz, zur Entschlüsselung dagegen die Option `-d`. Zusätzlich muss das zu verwendende Verfahren angegeben werden. In diesem Fall wird OpenSSL mittels `-aes256` zur Verwendung des Advanced Encryption Standard (AES) mit einer Schlüssellänge von 256 Bit angewiesen. Mittels `-in` bzw. `-out` übergibt man dem Programm schließlich noch die entsprechenden Daten für die Ein- bzw. Ausgabe.

3.3 Verschlüsselung von Blockgeräten

Blockgeräteverschlüsselungsverfahren arbeiten unterhalb der Dateisystemschicht. Sie stellen sicher, dass nur verschlüsselte Daten auf das entsprechende Blockgerät geschrieben werden. Dabei kann es sich um eine ganze Festplatte oder eine einzelne Partition handeln oder aber auch eine Container-Datei, die als Schleifengerät fungiert. Ist ein Blockgerät also nicht eingebunden und damit offline, sieht dessen Inhalt wie zufällige Daten aus. Es kann damit auch nicht festgestellt werden, ob dort ein Dateisystem vorhanden ist, geschweige denn welches. Ebenso verhält es sich natürlich mit den Daten selbst, die dort geschrieben wurden. Der Zugriff auf die Daten erfolgt durch Bereitstellung des Blockgeräts/Containers an einem beliebigen Ort der vorhandenen Verzeichnisstruktur. Im Folgenden wird auf die unter Linux gängigen Blockgeräteverschlüsselungsverfahren eingegangen.

3.3.1 dm-crypt

Der Linux Kernel enthält mit dem “device mapper“ ein Framework um physische Blockgeräte auf virtuelle Blockgeräte abzubilden. Dadurch entsteht eine zusätzliche Abstraktionsschicht für diverse Anwendungsmöglichkeiten; darunter zählen Funktionalität für Software-RAID oder der Logical Volume Manager (LVM) aber auch die Verschlüsselungsfunktionen von **“dm-crypt“**.

Am Beispiel der Laufwerksverschlüsselung werden die Daten von einem von “device mapper“ bereitgestellten virtuellen Blockgerät an ein anderes Blockgerät weitergereicht. Letzteres kann dabei physisch oder aber ein weiteres virtuelles Blockgerät sein, was dann auch eine Kombination der bereits genannten Möglichkeiten erlaubt. Somit lässt sich beispielsweise ein RAID-Verbund mit einer zusätzlichen Verschlüsselungsschicht kombinieren. Während die Daten von einem Blockgerät an das nächste “übergeben“ werden, lassen sich selbige entsprechend modifizieren, wie dies im Falle von dm-crypt in Form von Verschlüsselung geschieht. Es wird eine Vielzahl von Verschlüsselungsalgorithmen unterstützt, da dm-crypt hierbei auf die Crypto API des Linux-Kernels zurückgreift. [14]

Die Software Cryptsetup steuert die Funktionalitäten von dm-crypt im user-space. Sie zielt primär auf das Linux Unified Key Setup (LUKS) sowie die ältere Variante (ohne Header) “plain dm-crypt“ ab, bietet aber auch Basiskompatibilität zu loopAES- und TrueCrypt/VeraCrypt-Geräten. Seit der im Februar 2020 freigegebenen Version 2.3.0 unterstützt Cryptsetup erstmalig auch die BitLocker-Verschlüsselung, die bei Microsoft Windows verwendet wird. [15]

Mit dem Release des Linux Kernels in der Version 2.6.4 wurde dm-crypt erstmals integriert und damit offiziell freigegeben. Damals (März 2004) gab es zunächst lediglich einen “plain mode“, bei dem allerdings stets zusätzliche Parameter, wie beispielsweise der Verschlüsselungsalgorithmus übergeben werden mussten bzw. sollten. Dies machte die Nutzung zu einen etwas unhandlich, zum anderen konnten bei Angabe falscher Parameter zum Überschreiben bereits bestehenden Daten kommen.

Ein mittels (plain) dm-crypt initialisiertes Gerät kann mit folgendem Kurzbefehl eingerichtet werden. Satt <device> ist das zugrunde liegende physische/logische Blockgerät bzw. statt <target> das neu anzulegende Blockgerät anzugeben.

```
# cryptsetup open -type plain <device> <target>
```

Diese Eingabe enthält aktuell die nachfolgenden Standardwerte, welche in älteren Versionen allerdings abweichend sein können.

```
# cryptsetup --type plain --cipher aes-cbc-essiv:sha256 \
  --key-size 256 --hash ripemd160 <device> <target>
```

Wurde also in der Vergangenheit ein Gerät im `plain`-Modus mit Standardwerten erstellt, könnten Daten ungewollt überschrieben werden, wenn es mit einer aktuellen Version geöffnet wird. Seitens Cryptsetup findet hier keinerlei Plausibilitätsprüfung statt.

Die im Folgejahr veröffentlichte Erweiterung *LUKS* sollte diese Gefahr des *plain*-Modus durch Verwendung eines zusätzlichen Headers, der sämtliche Verschlüsselungsparameter enthält, allerdings beseitigen.

3.3.2 LUKS (Linux Unified Key Setup)

Das Linux Unified Key Setup (LUKS) ist eine Erweiterung für dm-crypt, deren Version 1.0 im Januar 2005 von Clemens Fruhwirth veröffentlicht wurde. Gegenüber der einfachen (plain) Variante bietet LUKS einen zusätzlichen Komfort. Sämtliche Informationen zur Initialisierung des verschlüsselten Blockgerätes werden in Form eines Headers gespeichert. Damit wurde die Übergabe weiterer Parameter - wie Angaben zum verwendeten Verschlüsselungsverfahren - beim Öffnen des Gerätes obsolet. Anfangs musste der Header noch vor den eigentlichen verschlüsselten Daten, auf dem selben Blockgerät liegen. Seit Version 1.4 ist es allerdings möglich, ihn abzutrennen und an einem beliebigen Ort zu speichern; man spricht hier sodann von einem “detached header“. [16]

Ein LUKS-Gerät wird mit folgendem Kurzbefehl initialisiert, wobei anstelle von <device> das zu verwendete physische/logische Blockgerät anzugeben ist.

```
# cryptsetup luksFormat <device>
```

Die Parameter können natürlich auch den eigenen Wüschen angepasst werden. Im folgenden Beispiel wurden die jeweiligen Standardwerte übergeben.

```
# cryptsetup -v --type luks --cipher aes-xts-plain64 \
  --key-size 256 --hash sha256 --iter-time 2000 --use-urandom \
  --verify-passphrase luksFormat <device>
```

Zeitgleich mit der 2. Version von Cryptsetup wurde auch die 2. Version von LUKS eingeführt - wenn auch nicht direkt als Standard. Bis Cryptsetup 2.0.0 wurde noch auf LUKS1, seit Version 2.0.6 wird auf LUKS2 zurückgegriffen; eine manuelle Auswahl von LUKS1 über den Parameter `--type luks1` bleibt allerdings bestehen.

Ein bereits initialisiertes LUKS-Gerät wird mit dem nachfolgenden Kurzbefehl geöffnet. Mit <device> wird das zugrunde liegende physische/logische Blockgerät bezeichnet; mit <target> das neu anzulegende Blockgerät. Da die Version aus dem Header entnommen wird, ist deren Angabe nicht von belang.

```
# cryptsetup luksOpen <device> <target>
```

LUKS1 hatte gegenüber der “plain“ Variante den Fokus auf kryptografischer Sicherheit. Das Hinzufügen von Salts für Schlüssel bzw. Masterschlüssel soll Angriffe mit vorberechneten Hashes erschweren. Um einen Schlüssel vom verwendeten Passwort abzuleiten, wurde die Password-Based Key Derivation Function 2 (PBKDF2) implementiert. Aufgrund frei konfigurierbarer Iterationen erfordert sie einen erhöhten Rechenaufwand und verlangsamt dadurch letztlich Wörterbuchangriffe. Um verschiedene Passwörter auf einem einzigen Gerät nutzen zu können, wurden darüber hinaus 8 Keyslots eingeführt; jeder einzelne von ihnen kann zudem verändert oder gelöscht werden. [17]

Seit LUKS2 (ab Kernel 4.12) werden Metadaten und Header fortan im JSON-Format gespeichert. Um parallele Brute-Force-Angriffe zu erschweren, wurde zudem die Funktion zur Ableitung des Schlüssels aus dem Passwort durch Argon2 abgelöst. Es besteht eine partielle Abwärtskompatibilität; eine Konvertierung von LUKS1 nach LUKS2 und umgekehrt ist damit unter gewissen Voraussetzungen möglich. [18]

3.4 Gegenüberstellung der Verschlüsselungsmethoden

Da die Gesamtdatenmenge grundsätzlich variabel gehalten werden sollte bzw. von vornherein noch gar nicht feststeht eignen sich Verschlüsselungsverfahren auf Dateiebene nur bedingt. Das manuelle Chiffrieren jeder einzelnen Datei wäre sicherlich zu aufwendig. Zudem bliebe die Struktur der Daten selbst - wie die Anordnung in Verzeichnissen, sowie zusätzliche Informationen - wie die Dateigröße erhalten. Dies ließe sich umgehen, indem zunächst die gesamte Struktur in einem Archiv komprimiert und dieses im Anschluss verschlüsselt würde. Bei sehr großen Archiven mit hoher Kompression ist der Aufwand allerdings auch bei geringen Änderungen durch Hinzufügen oder Löschen von Daten enorm.

Verfahren wie GnuPG oder OpenSSL eignen sich dennoch sehr gut für die “Überschlüsselung“ von Datenströmen. Aus anti-forensischer Sicht können entsprechende Dateisignaturen verborgen werden. Eine solche Tarnung kann allerdings auch durch bloße Kodierung erreicht werden; so soll im Prototyp der Software auf eine zusätzliche Überschlüsselung zunächst verzichtet werden.

In einer der Arbeit vorausgehenden Untersuchung wurden auch Verfahren zur Verschlüsselung ganzer Verzeichnisse auf Basis eines bestehenden Dateisystems betrachtet. Sie eignen sich für den angestrebten Zweck nur bedingt, da die steganografischen Aspekte nur schwierig umzusetzen sind. Diese Dateisysteme spielen ihre großen Vorteile aus, wenn es darum geht, vorhandene Dateisysteme zu schützen, ohne den Gerätezugriff zu blockieren (z. B. Netzwerk-Freigaben, Cloud-Speicher, etc.) oder dateibasierte Sicherungen verschlüsselter Dateien offline durchführbar sein müssen.

Verfahren zur Blockgeräteverschlüsselung eignen sich hierzu eher, da deren größter Nachteil - die mangelnde Flexibilität hinsichtlich des Speicherplatzes - eigentlich nicht von Belang ist. Ist die Steganografie selbst stark genug, die Tatsache des Verschlüsselns selbst zu tarnen, gibt es für einen potentiellen Angreifer grundsätzlich keinen Grund nach chiffrierten Daten zu suchen. Zudem werden bei der Blockgeräteverschlüsselung sensible Metadaten mitverschlüsselt. Dies betrifft vor allem die Anzahl der Dateien, deren Größe, Zeitstempel oder Zugriffsberechtigungen sowie die grundlegende Verzeichnisstruktur.

Es kann somit zusammenfassend festgestellt werden, dass die Blockgeräteverschlüsselung die größten Aussichten auf Erfolg verspricht, wenn es um möglichst viel Speicherplatz geht. Zudem erhält man Cryptsetup - insbesondere dm-crypt mit LUKS - ein ausgereiftes Toolkit für die Blockgeräteverschlüsselung unter Linux mit vielen Konfigurationsparametern hinsichtlich des Headers, der die Metadaten enthält und des Payloads, in dem sich ausschließlich Chiffretext befindet. Die Möglichkeit des abtrennbaren LUKS-Headers macht es möglich Header und Payload an verschiedenen Orten abzulegen bzw. in diesem Fall unterschiedliche steganografische Verfahren anzuwenden.

Der LUKS-Header soll hierzu zunächst kodiert werden, um dessen Signatur zu tarnen und anschließend mittels Bildsteganografie verborgen werden. Durch diverse Konfigurationsparameter lässt sich die Größe des Headers auch auf unterschiedliche Szenarien anpassen, sollte auf die Verwendung von LUKS2 Wert gelegt werden. Auf eine Überschlüsselung des Headers soll in der ersten Implementierung der Software zunächst verzichtet werden, da dies ggf. die Eingabe eines weiteren Passworts mit sich bringen würde. Durch den modularen Aufbau lässt sich dies jedoch später nachrüsten.

Der LUKS-Payload kann auf verschiedenste Weise verborgen werden. Die einfachste Möglichkeit wäre sicherlich das Konkatenieren mit einer unverfänglichen Datei, die nie verändert wird. Allerdings dürfte die Herangehensweise zu leicht detektierbar sein, weshalb die Datenträgersteganografie die bessere Alternative darstellt. Hierbei soll eine Swap-Partition bzw. eine Swap-Datei “zweckentfremdet“ werden, da deren Vorhandensein grundsätzlich keinen Verdacht schöpfen lässt und die Auslastung meist eher gering, bzw. der empfohlene Swap-Space oftmals überdimensioniert ist.

Damit der Swap-Space für das Vorhaben genutzt werden kann, muss er bei Verwendung der Software deaktiviert werden, da dm-crypt exklusiven Zugriff auf das Blockgerät bzw. die Datei benötigt. Dies hat aber auch den positiven Nebeneffekt, dass sodann auch kein Suspend-to-Disk mehr möglich ist; das Abspeichern der Schlüssel durch “swapping“ des RAM auf den lokalen Datenträger wird dadurch unterbunden. Die Vertraulichkeit soll als eines der Schutzziele der Informationssicherheit schließlich auch nicht außer Acht gelassen werden.

4 Design der Software

Wie den vorausgegangenen Kapiteln entnommen werden kann, wird die Software auf eine Kombination aus Blockgeräteverschlüsselung (via dm-crypt/LUKS) unter Nutzung eines abgetrennten Headers und diversen steganografischen Verfahren aufbauen. Die verschlüsselten Daten werden dabei innerhalb der Datenstruktur einer Swap-Partition bzw. eines Swap-Files verborgen. Zur Verschleierung sämtlicher Metadaten werden selbige mittels Bildsteganografie über gewöhnliche Hintergrundbilder oder Urlaubsfotos aufgeteilt. Dabei werden von n Bildern k für die Rekonstruktion des LUKS-Headers benötigt, wobei $k << n$ gilt. Als zusätzliches Sicherheitsfeature kann die Reihenfolge der Bilder dabei ebenfalls relevant sein. Um eine zusätzliche Überschlüsselung des Headers zu vermeiden müssen statische Daten, die eine Identifizierung als LUKS-Header zulassen würden, ebenfalls unkenntlich gemacht werden. Die Software muss dabei die nachfolgenden, grundlegenden Schritte durchlaufen:

- Auswahl der Trägerdateien durch Nutzerinteraktion
- Extraktion der benötigten Daten zur Rekonstruktion des Headers
- Entschlüsselung des Blockgerätes und Einhängen des Dateisystems
- Überschreiben bzw. Löschen des zusammengesetzten Headers
- Aushängen des Dateisystems und Schließen des Krypto-Containers

Hinsichtlich des Speicherortes der Trägerdaten werden durch die Software keinerlei Vorgaben gemacht. Der Nutzer kann gemäß eigener Paranoia selbst entscheiden, ob er die Daten in der Cloud oder direkt auf dem verwendeten Computer abspeichert. Auch die Wahl geeigneter Trägermedien bleibt dem Nutzer selbst überlassen, es soll jedoch nochmals darauf hingewiesen werden, keine “Allerweltsbilder“ zu verwenden, da die Bildsteganografie dadurch leichter auffliegen könnte.

Grundsätzlich ist es ratsam, die Software nicht offensichtlich auf dem Gerät mitzuführen, da deren bloßes Vorhandensein bereits Vermutungen oder Schlüsse zulassen und letztlich die plausible Abstreitbarkeit gefährden könnte. Ein Download bei Bedarf funktioniert natürlich nur außerhalb von Nord-Korea.

4.1 Namensgebung und Wahl der Programmiersprache

Die Software wurde unter dem Codenamen **Stingray** (englisch für “Stachelrochen“) entwickelt. Mit seiner flachen Körperform und dessen der Meeresvegetation angepassten Farbgebung ist es einem Rochen möglich sich behutsam und relativ unauffällig zur bewegen bzw. in Meeresbodennähe mit selbigem optisch zu verschmelzen.

Dieses Ziel verfolgt auch die Software hinsichtlich ihrer steganografischen Eigenschaften, das Vorhandensein verschlüsselter Daten zu verbergen, indem die kryptografischen Metadaten mit dem Bildmaterial der Trägerdaten verschmelzen. Durch die Integration einiger von *cryptsetup* bereitgestellter Verschlüsselungsverfahren können sensible Daten nach aktuellem Stand der Technik vor fremden Zugriffen geschützt werden, was *Stingray* letztendlich zu einem Werkzeug macht, das nicht unterschätzt werden sollte. Selbiges gilt auch für Stachelrochen, die - obwohl grundsätzlich friedlich lebend - ebenfalls nicht unterschätzt werden sollten, wie die traurige Geschichte von Stephen Robert "Steve" Irwin beweist. Er wurde am 4. September 2006 von einem Stachelrochen ins Herz gestochen und starb unmittelbar an den Folgen. [19]

Die Software wurde mit der Programmiersprache **Python 3** erstellt. Sie ist frei erhältlich, bei gängigen Linux-Distributionen bereits vorinstalliert und bringt weitere Vorteile mit sich, die bei der Erstellung von *Stingray* relevant waren. Eine der größten Stärken von Python ist die mächtige Standardbibliothek, so dass für zahlreiche Anwendungsfälle auf die mitgelieferten Klassen zurückgegriffen werden kann. Zur Umsetzung des Projekts sei vor allem das Prozess-Management zur Steuerung der Standardein- und -ausgabe zu erwähnen. Die Module der Standardbibliothek können durch weitere, ggf. selbst geschriebene Module ergänzt werden. Für die Bildsteganografie wird hierzu das Modul *PIL* (Python Image Library) importiert, um die erforderlichen Manipulationen an Bilddateien vorzunehmen. Durch die Module *os* und *sys* bietet Python darüber hinaus eine umfangreiche Schnittstelle zur Kommunikation mit dem Betriebssystem. Das macht sich vor allem bei der Dateiarbeit positiv bemerkbar. Da die Software für das Betriebssystem Linux konzipiert wurde und unter Linux auch Blockgeräte "als Datei" angesehen werden, erfolgt der Zugriff auf Datenträger bzw. Partitionen ebenfalls über diese Module. Ein weiterer Vorteil von Python ist dessen ausgereifte Ausnahmebehandlung (exception handling), mit der nicht nur diverse Fehlerbedingungen getestet, sondern Syntaxfehler auch noch zur Laufzeit abgefangen und behandelt werden können. Dies kann man sich bei der Implementierung von interaktiven Eingaben zu Nutze machen, indem selbige zur Laufzeit überprüft werden und der Programmcode entsprechend reagieren kann. Zuletzt nutzt Python dynamische Datentypen, sodass eine Typumwandlung (casting) von Variablen direkt erfolgen kann. Dies muss in *Stingray* häufig passieren, wenn beispielsweise zwischen den Datentypen *string* und *bytes* "geswitcht" wird. [20]

4.2 Grundlegender Aufbau

Bei Planung der Software wurde besonderer Wert auf eine intuitive Bedienbarkeit gelegt. Da *Stingray* hinsichtlich der Verschlüsselung auf *Cryptsetup* aufbaut, lag es nahe, die grundlegende Syntax daran anzulehnen und diese um benötigte Optionen zu erweitern. Zugriff auf die Grundfunktionalitäten der Software erhält man dahingehend - in Anlehnung an `cryptsetup luksFormat|open|close <device>` - folglich durch den Aufruf von: `stingray create|open|close <device>`.

Ebenso wie *Cryptsetup* bietet auch *Stingray* lediglich ein CLI (Command Line Interface). Auf die Entwicklung einer Grafischen Benutzeroberfläche wurde zugunsten eines interaktiven CLI verzichtet, das in drei Schritten arbeitet. Alle benötigten Informationen werden dabei zunächst durch direkte Interaktion abgefragt. Stehen alle Variablen fest, werden sie in einer Übersicht aufbereitet, um eine Kontrolle durch den Nutzer zu ermöglichen. Erst nach aktiver Bestätigung erfolgt der eigentliche Prozess, womit eine Fehlbedienung grundsätzlich verhindert werden soll. Dennoch sollte *Stingray* stets mit Sorgfalt und nur bei Vorhandensein entsprechender Kenntnisse verwendet werden; es handelt schließlich nicht um ein “Rookie Tool“.
Durch die Übergabe weiterer Funktionsparameter in Form von Schlüsselwortvariablen kann der Interaktive Modus auf ein Minimum reduziert werden. So können beispielsweise der zu verwendende Modus oder der Pfad zu den Trägermedien direkt übergeben werden. Die Informationen werden durch einen *argparser* in die jeweiligen Variablen überführt und müssen somit nicht mehr interaktiv abgefragt werden.

Um die Steganografie umzusetzen, stellt *Stingray* derzeit vier Modi für unterschiedliche Anwendungsfälle zur Verfügung:

default Im Standardmodus wird mit einem verkleinerten LUKS2-Header gearbeitet, der von 16 MB auf akzeptable 2020 KB geschrumpft wurde, um die benötigte Kapazität für den Payload möglichst gering zu halten. Dadurch können weniger Trägermedien genutzt und/oder weniger Pixelbits verändert werden. Die Reihenfolge der verwendeten Bilder wird zudem gespeichert, sodass diese lediglich bekannt sein müssen.

legacy Im Kompatibilitätsmodus wird auf LUKS1 zurückgegriffen, sollte seitens des Nutzers eine Abwärtskompatibilität gewünscht werden; die Headergröße beträgt von Haus aus 2020 KB. Ansonsten arbeitet *legacy* analog zu *default*.

paranoid Der verschärfte Modus verwendet LUKS2 mit regulärer Headergröße von 16 MB. Durch den größeren Schlüsselbereich ist das Verfahren damit sicherer als die beiden vorstehenden, es werden dadurch aber auch mehr Trägerdateien für den Payload benötigt. Bei einer Auflösung von 1920 x 1080 Pixeln und der Verwendung von 3 Pixelbits werden ca. 8 Bilder benötigt; bei 2560 x 1440 und 2 Pixelbits sind es immer noch 6. Zusätzlich ist die Reihenfolge der Bilder relevant, da diese nicht gespeichert wird.

portable Der portable Modus arbeitet rein datenträgersteganografisch, also ohne Verwendung bildsteganografischer Verfahren. Sämtliche Metadaten werden somit anderweitig auf geeignete Weise verschleiert.

Das nachfolgende Programmablaufdiagramm zeigt die wichtigen Einzelschritte bei der Erstellung eines neuen “Stingrays“ auf Basis einer Swap-Partition bzw. Swap-Datei.

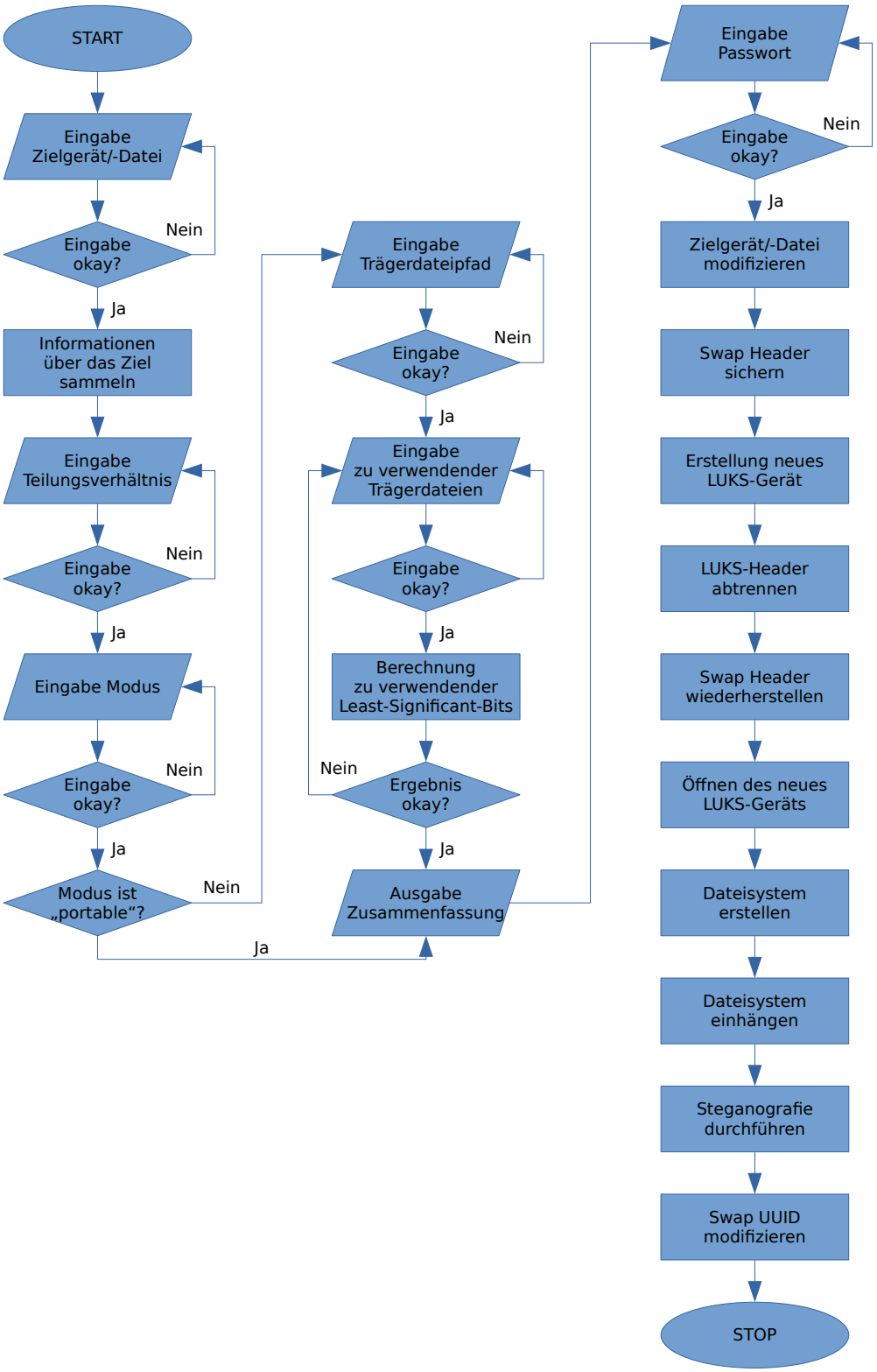

Abbildung 4.1: Programmablaufdiagramm zur Erstellung eines neuen Datenträgers/Containers

Die **Erstellung eines Stingrays** erfolgt durch den Aufruf mit dem Parameter `create`, grundsätzlich gefolgt von `<target>` - dem zu modifizierenden Ziel (Swap-Partition bzw. -Datei), welches später den verschlüsselten Payload enthält. Die Übergabe der nachfolgenden Schlüsselwortvariablen ist dabei optional. Sie wurden eingeführt, um den Initialisierungsvorgang auf ein Minimum verkürzen zu können.

`quota=` Teilungsverhältnis zwischen dem Speicherplatz, der künftig weiterhin als Swap genutzt werden kann und dem Rest, der für den Payload verwendet wird; z. B. `quota=25` (25% Swap-Anteil und 75% Payload-Anteil)

`mode=` Angabe eines der vier steganografischen Modi, wie bereits in diesem Abschnitt beschrieben; z. B. `mode=legacy` (Bildsteganografie und LUKS1)

`path=` absoluter oder relativer Pfad zu den zu verwendenen Trägerdateien für die bildsteganografischen Verfahren (wird im portablen Modus nicht benötigt und würde dort ignoriert); z. B. `path=Pictures/Wallpapers/`

Betrachtet man das Programmablaufdiagramm auf der vorherigen Seite, so werden genau diese Schlüsselwortvariablen auch interaktiv abgefragt, sollten sie nicht übergeben worden sein. In der linken Spalte erfolgt sowohl die Abfrage als auch eine Plausibilitätsprüfung der Variablen *target*, *quota* und *mode*. Wurde nicht der portable Modus gewählt, erfolgt der Sprung an den Beginn der zweiten Spalte, wo sodann die Variablen *path* und *images* abgefragt werden; bei letzterer handelt es sich im die gewählten Trägerdateien. Im nächsten Schritt wird die Variable *lsb* berechnet; sie legt die Anzahl der zu verwendenden LSBs fest und ist entscheidend für die optische Qualität der Steganogramme. Sind alle notwendigen Variablen gesetzt, wird eine Zusammenfassung der Angaben angezeigt. Bislang wurden noch keinerlei Veränderungen am System durchgeführt. Erst wenn der Nutzer die Freigabe erteilt, werden alle notwendigen Modifikationen sequentiell abgearbeitet, wie in der dritten Spalte des Programmablaufdiagramms dargestellt.

Um eine mehrmalige Abfrage des Passworts zu vermeiden, erfolgt dies noch vor dem Start der eigentlichen Sequenz. Es hat den Hintergrund, da Cryptsetup im Folgenden zweimalig aufgerufen werden wird; zum Anlegen des Gerätes und ein weiteres Mal beim Öffnen. Nach Eingabe des Passworts wird das Zielgerät gemäß des angegebenen Teilungsverhältnisses modifiziert und der neue Swap-Header gesichert. Der Swap wird sodann als LUKS-Gerät initialisiert, ein Backup des LUKS-Headers durchgeführt und der Swap-Header wiederhergestellt; damit wird der auf dem Gerät erstellte LUKS-Header überschrieben. Mit dem abgetrennten LUKS-Header wird das Gerät schließlich geöffnet, ein Dateisystem erstellt und dieses in die Verzeichnisstruktur des System eingehangen. Zum Schluss wird die Steganografie entsprechend des gewählten Modus und die dazugehörige Modifikation am UUID des Swap durchgeführt.

Das **Öffnen** bzw. **Schließen eines Stingrays** erfolgt durch den Aufruf mit dem Parameter `open` bzw. `close`. Sollten hierbei keinerlei Schlüsselwortvariablen übergeben worden sein, erfolgt die interaktive Abfrage analog der Erstellungsroutine.

4.3 Ausgewählte Trägermedien für die Steganografie

Wie bereits am Ende des Kapitels *Steganografische Verfahren* angemerkt, sollten für die Bildsteganografie Dateien im PNG-Format genutzt werden. Da es sich hierbei um ein Rastergrafikformat handelt, lässt sich die Steganografie mit der LSB-Technik unter vertretbarem Aufwand implementieren. Zwar setzen Digitalkameras erfahrungsgemäß immer noch vorwiegend auf JPEG, was die Verwendung von Urlaubsfotos ermöglichen könnte, jedoch liegen beispielsweise Desktop-Hintergründe oftmals auch als PNGs vor, die sodann als Trägermedien verwendet werden können. Bei “defensiver“ Nutzung der Trägerdateien ist die Veränderung mit dem bloßen Auge nicht zu erkennen.

Zum Speichern größerer Datenmengen - in diesem Fall die verschlüsselten Daten des gesamten LUKS-Payloads - wird auf die Möglichkeit zur Manipulation von Swap-Files bzw. -Partitionen zurückgegriffen. Der Vorteil besteht darin, dass die ursprüngliche Funktion zur Auslagerung des Arbeitsspeichers erhalten bleibt, wenngleich die Kapazität dadurch vermindert wird.

4.3.1 Bildformat: Portable Network Graphics (PNG)

Am 1. Oktober 1996 wurde das Dateiformat Portable Network Graphics (PNG) von der PNG Development Group veröffentlicht und ist seither nach ISO/IEC 15948 standardisiert. Es wird vom World Wide Web Consortium (W3C) gefördert und ist laut deren Statistik derzeit auch das meistverwendete verlustfreie Grafikformat im Internet. Wie aus der nachfolgenden Abbildung hervorgeht, werden PNG und JPEG auf weitaus mehr Internetseiten verwendet als andere Grafikformate.

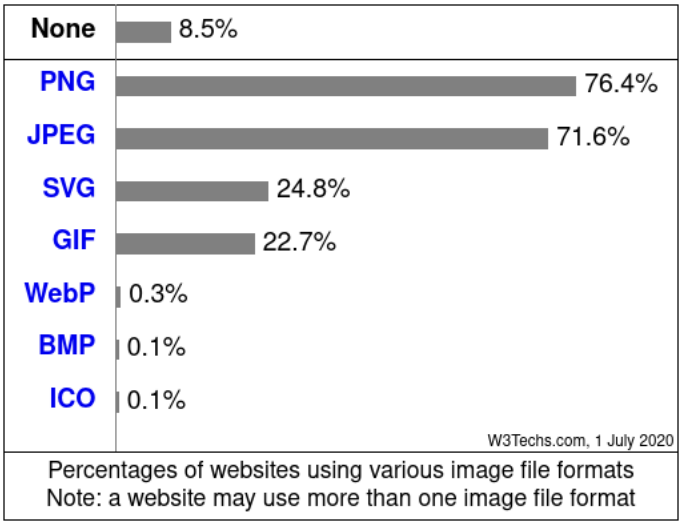

Abbildung 4.2: Prozentuale Nutzung von PNG im Vergleich zu anderen Bildformaten (Quelle: `https://w3techs.com/technologies/overview/image_format`)

Das bis dahin weit verbreitete Graphics Interchange Format (GIF) hatte den Nachteil, dass der dort verwendete und zur Datenkompression genutzte LZW-Algorithmus durch CompuServe patentiert war; er lief erst 2004 aus. Darüber hinaus konnte die Limitierung auf 256 Farben bei der Darstellung realitätsnahe Bilder zum Problem werden. Gegenüber dem JPEG File Interchange Format (JFIF) hatte GIF andererseits jedoch den Vorteil der Verwendung eines Transparenzkanals und der verlustfreien Kompression.

Mit PNG wurde ein Rastergrafikformat mit verlustfreier Datenkompression geschaffen, das die Vorzüge von GIF bzw. JPEG aufgreift und dabei auf keinerlei lizenzrechtliche Probleme stößt. Das neu entstandene Format vereint nachfolgende wichtige Eigenschaften: [21]

- **Progressiver Bildaufbau**
  Da das Dateiformat vornehmlich für die Nutzung im World Wide Web vorgesehen war, musste es netzwerkbasierte Verwendung unterstützen und damit seriell verarbeitet werden können. Das Bild muss also bereits aufgebaut werden können, solange die Datenübertragung noch läuft. Man spricht hierbei von progressivem Bildaufbau.
- **Unterstützte Farbräume**
  Grundsätzlich stehen sowohl Farb- als auch Graustufenbilder zur Verfügung, jeweils mit oder ohne Transparenz- oder Alphakanal; die Verwendung indizierter Farben über eine Farbpalette wie bei GIF ist ebenfalls möglich. In der Regel wird zur realistischen Darstellung der Bildinhalte auf Truecolor mit 24bit RGB zurückgegriffen. Jedes Pixel besteht dann aus jeweils einem vom 256 Werten für die Farben rot, grün und blau.
- **Verlustfreie Kompression**
  Zur Kodierung der Daten werden Filter- und Kompressionsverfahren angewandt, die eine verlustfreie Rekonstruktion ermöglichen.
- **Allgemeine Rechtssicherheit**
  Das wichtigste Ziel war es, ein einfach zu implementierendes Grafikformat zu entwickeln, dass keine patentrechtlichen Probleme mit sich bringt. Nach und nach konnte das PNG-Format von allen gängigen Bildbearbeitungsprogrammen und Webbrowsern verarbeitet werden.

Grundsätzlich wird der Bildinhalt in drei aufeinanderfolgenden Schritten in das PNG-Format codiert; diese sind *Interlacing*, *Filterung* und *Kompression*.

Beim **Interlacing** kommt der von Adam M. Costello entwickelte Adam7-Algorithmus zum Einsatz. Im Gegensatz zu anderen Verfahren arbeitet er nicht nur zeilenweise, sondern nutzt beide Bilddimensionen aus. Durch dieses mehrdimensionale Interlacing werden die Ergebnisse beim Aufbau eines PNG für das menschliche Auge schneller sichtbar, als dies beispielsweise bei GIF (verwendet eindimensionales Interlacing) der Fall wäre.

Das Bild wird zunächst in Blöcke von 8x8 Pixeln unterteilt, die daraufhin die sieben Phasen des Adam7-Algorithmus durchlaufen. In der ersten Phase ist lediglich das Blockpixel (0,0) sichtbar; in der zweiten Phase folgt das Blockpixel (0,4). Von nun an wird die Anzahl der sichtbaren Pixel verdoppelt, bis schließlich alle 64 Blockpixel dargestellt sind. Die nachfolgende Abbildung verdeutlicht den Ablauf der sieben Phasen des Interlacings anhand von vier 8x8-Blöcken. [21]

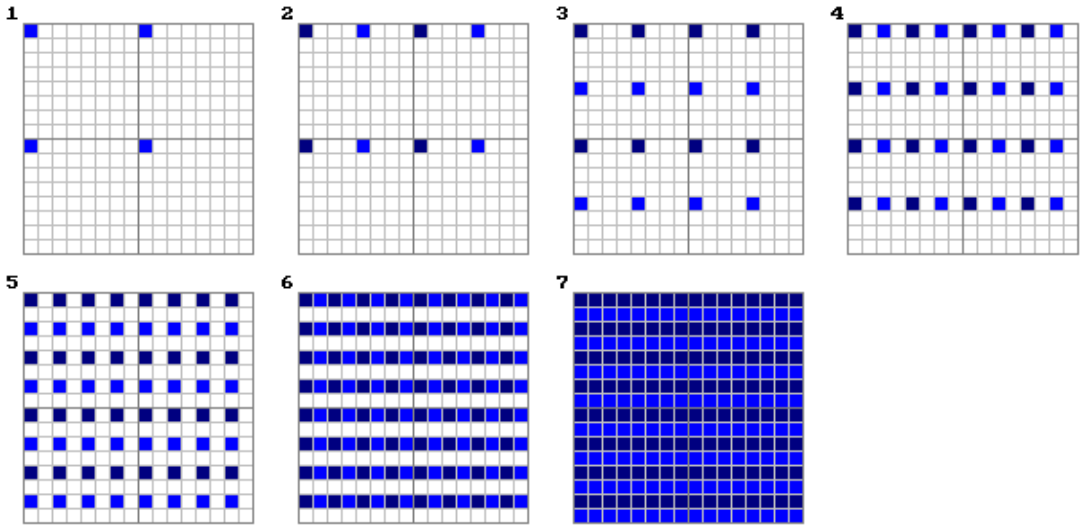

Abbildung 4.3: Ablauf des Interlacing-Verfahrens bei PNG nach Adam7
(Quelle: `https://commons.wikimedia.org/wiki/Category:Adam7_algorithm`)

Als nächster Schritt folgt die **Filterung** der Daten. Hierbei soll erreicht werden, dass der Informationsgehalt der Daten so weit wie möglich reduziert wird um eine bestmögliche Grundlage für die spätere Kompression zu schaffen. Ein Pixel X wird stets in Abhängigkeit von den Nachbarpixeln A (links), B (oben) und C (links oben) codiert. Der Paeth Predictor wurde von Allan Paeth entwickelt und bezieht alle drei bereits gefilterten Nachbarpixel mit ein. Es werden bei der Filterung der Daten entsprechende Differenzwerte gebildet, die letztendlich anstelle der Ursprungswerte gespeichert werden. Sie erfolgt in Form einer Prädiktion und ist in der nachfolgenden Tabelle zusammengefasst. [21]

Tabelle 4.1: PNG Filtertypen
(Quelle: http://www.libpng.org/pub/png/book/chapter09.html)

Filtertyp	**Filtername**	**Filterbeschreibung**
0	None	keine Änderung
1	Sub	Differenz zum Pixel A
2	Up	Differenz zum Pixel B
3	Average	Differenz zum Mittelwert der Pixel A und B
4	Paeth	Differenz zum PaethPredictor der Pixel A, B und C

Der letzte und zweifelsohne wichtigste Schritt ist die **Kompression** der vorgefilterten Daten. Hierbei wurde in der Spezifikation von PNG das von Jean-Loup Gailly und Mark Adler entwickelte DEFLATE-Verfahren als alleinige Kompressionsmethode festgelegt. Die eigentliche Kompression ist eine Kombination aus LZ77 und Huffman-Kodierung. LZ77 wurde 1977 von Abraham Lempel und Jacob Ziv veröffentlicht und kann ohne vorherige Kenntnis der Daten komprimieren, so dass die Daten bei der Kompression nur einmal durchlaufen werden müssen. Die Huffman-Kodierung wurde bereits 1952 von David A. Huffman entwickelt. Wie bei Entropiekodierungen üblich, werden dabei häufig verwendete Symbole auf möglichst kurze Zeichenketten abgebildet, um ein Maximum an Kompression zu erlangen. Beide Methoden wurden schließlich durch Phillip Walter Katz zur Verwendung in seinem Archivierungswerkzeug PKZIP kombiniert und als DEFLATE in der RFC 1951 (1996) spezifiziert. Das Verfahren arbeitet verlustfrei und unterliegt keinen Patenten, die einer freien Nutzung entgegenstehen. [21]

Abschließend sei angemerkt, dass das Interlacing bei PNG grundsätzlich optional ist, da es lediglich dem progressiven Bildaufbau dient. Zudem führt dessen Verwendung zu einer deutlichen Reduktion der Kompressionsrate, da benachbarte Pixel durch den Adam7-Algorithmus voneinander getrennt werden. Es sollte also nur angewandt werden, wenn auf die Bildinhalte vornehmlich bei Datenfernübertragungen zugegriffen wird.

4.3.2 Swap Space als dateisystemähnliche Datenstruktur

Unter Linux wird der physisch verfügbare Arbeitsspeicher (RAM) in einzelne Teilbereiche gesplittet, die sogenannten “pages“. Beim eigentlichen “swapping“ können solche pages vom Betriebssystem aus dem physischen Speicher auf einen vordefinierten Speicherbereich der Festplatte ausgelagert werden. Dadurch soll verhindert werden, dass der Arbeitsspeicher nicht dauerhaft mit Daten belegt wird, die beispielsweise nur beim Start eines Programms erforderlich waren und bis zu dessen Beendigung gar nicht mehr benötigt werden. Sollte nun eine andere Anwendung vom System viel Speicher anfordern, könnte der physisch verfügbare Arbeitsspeicher möglicherweise nicht mehr ausreichen, was im schlimmsten Fall zu einem instabilen Gesamtsystem führen könnte. Natürlich sind Festplatten um ein vielfaches langsamer als der Arbeitsspeicher, weshalb oft genutzte Daten möglichst nicht “geswappt“ werden sollten. Verwaltet wird das Prozedere durch den Kernel des Betriebssystems.

Der “Swap Space“ kann auf unterschiedliche Weise realisiert werden. Grundsätzlich wird hierzu eine eigene Partition vom Typ 82 angelegt. Diese wird durch den Befehl `mkswap` mit der Swap-Datenstruktur initialisiert und mit dem Befehl `swapon` aktiviert. Es besteht zudem die Möglichkeit ein “Swap File“ innerhalb eines bestehenden Dateisystems anzulegen. Initialisierung und Aktivierung verlaufen analog, jedoch kann eine Swap-Datei nicht für den Ruhezustand verwendet werden, da der Zugriff durch auf den Swap durch das dazwischenliegende Dateisystem “hindurch“ erfolgen müsste. [22]

Wie bereits erwähnt, arbeitet Swap nicht mit herkömmlichen Dateien sondern mit RAM-Pages. Es müssen also keine Metadaten gespeichert werden, wie dies bei klassischen Dateisystemstrukturen der Fall ist. Dahingehend ist die Struktur des Swap Space auch um ein vielfaches simpler, als die von "echten" Dateisystemen.

Die erste Page eines jeden Swap-Bereichs ist der **Swap Header**. Die Größe jeder einzelnen Page wird durch den Kernel vorgegeben und beträgt i.d.R. 4096 Byte. Der Header endet nach aktueller Spezifikation mit den "magic bytes" `53 57 41 50 2d 53 50 41 43 45` bzw. `53 57 41 50 53 50 41 43 45 32`, die die Strings `SWAP-SPACE` bzw. `SWAPSPACE2` abbilden. [22]

Listing 4.1: Hexdump einer neu erstellten Swapdatei, Dateigröße 1GB

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  01 00 00 00 ff ff 03 00  00 00 00 00 2d 7f 98 78  |............-..x|
00000410  76 2c 40 32 b7 f9 e2 61  0f 95 d4 f7 41 42 43 44  |v,@2...a....ABCD|
00000420  45 46 47 48 49 4a 4b 4c  4d 4e 4f 00 00 00 00 00  |EFGHIJKLMNO.....|
00000430  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000ff0  00 00 00 00 00 00 53 57  41 50 53 50 41 43 45 32  |......SWAPSPACE2|
00001000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
40000000
```

Wie im vorstehenden Hexdump ersichtlich, endet der Swap Header (Page Zero) nach 4096 Bytes an Offset 0xfff mit dem String `SWAPSPACE2`. Damit ist die Größe einer Page auf 4096 Bytes festgelegt; der Rest des Swap ist offensichtlich "leer."

An Offset 0x404 findet sich über die Länge von 4 Bytes die Anzahl der verwendbaren Pages - exklusive Page Zero. In diesem Beispiel können vom System bis zu 262143 (0x3ffff, little endian) "geswappt" werden. Zuzüglich des Swap-Headers beträgt die absolute Größe in Bytes damit "Anzahl der Pages" $*$ "Größe einer Page" $+$ "Größe des Headers", also in diesem Fall $262143 * 4096 + 4096 = 1073741824$. Dies entspricht exakt 2^{30} Bytes bzw. 1 GB, der Größe der zugrundeliegenden Datei.
An Offset 0x40c findet sich über die Länge von 16 Bytes die UUID des Swap (hier: 2d7f9878-762c-4032-b7f9-e2610f95d4f7). Die UUID kann auch manuell vergeben werden, andernfalls wird sie zufällig generiert.
An Offset 0x41c folgt ein bis zu 15-stelliges Bezeichner (hier: ABCDEFGHIJKLMNO). Das Label kann bei Initialisierung manuell vergeben werden, ansonsten bleiben die 15 Bytes leer (0x00).

Wie bereits im Abschnitt Datenträgersteganografie erläutert, kann das Betriebssystem grundsätzlich nur dort Dateien abspeichern, wo zuvor auch ein Dateisystem angelegt wurde. Beim Swap Space handelt es sich zwar nicht um ein Dateisystem im eigentlichen Sinne, es kann jedoch zumindest von einer "*dateisystemähnlichen Datenstruktur*" gesprochen werden; statt Dateien werden eben RAM-Pages gespeichert.

Der Vollständigkeit halber sollen diese Feststellungen durch einen Blick in den aktuellen Quellcode der für die Erstellung des Swap zuständigen Header-Datei bestätigt werden. (`https://github.com/torvalds/linux/blob/master/include/linux/swap.h`)

Listing 4.2: Definition des Swap-Headers im dem aktuellen Quellcode des Linux-Kernels

```
union swap_header {
  struct {
    char reserved[PAGE_SIZE - 10];
    char magic[10];     /* SWAP-SPACE or SWAPSPACE2 */
  } magic;
  struct {
    char    bootbits[1024]; /* Space for disklabel etc. */
    __u32   version;
    __u32   last_page;
    __u32   nr_badpages;
    unsigned char sws_uuid[16];
    unsigned char sws_volume[16];
    __u32   padding[117];
    __u32   badpages[1];
  } info;
};
```

Die grundlegende Struktur des Swap-Headers ist sehr einfach gehalten und besteht lediglich aus den beiden Bereichen *magic* und *info*. Der Bereich *info* beginnt an Offset 0 und besteht aus mehreren, aufeinanderfolgenden Teilbereichen. Am Angang steht ein Array (bootbits) der Länge 1024 vom Typ `char`. Es handelt sich hierbei um einen reservierten Speicherbereich, der allerdings derzeit nicht benutzt wird. Es folgen die Swap-Version, die letzte verwendbare Page und die Anzahl der defekten Pages jeweils vom Typ `unsigned integer`; sie umfassen also jeweils 4 Bytes. Darauf folgen (jeweils 16-stellig) Swap-UUID und -Label vom Typ `unsigned char` und schließlich noch etwas Padding bzw. ungenutzter Speicherplatz bis der Bereich *magic* den Header abschließt. Er besteht lediglich aus dem gleichnamigen Array vom Typ `char`, ist 10 Byte lang und soll gemäß Spezifikation den String SWAP-SPACE oder SWAPSPACE2 enthalten. Rechnerisch müsste die Seitengröße eines Swap damit mindestens 1550 Bytes betragen. Tatsächlich lassen sich auch Swaps mit einer Seitengröße von 2 KB manuell anlegen, vom Kernel wird sie im Regelfall automatisch auf 4 KB gesetzt.

Bei Initialisierung der Swap-Struktur mittles `mkswap` wird nach Möglichkeit der gesamte zur Verfügung stehende Speicherplatz der Partition bzw. Datei als Swap-Bereich markiert und die letzte nutzbare Page an Offset 0x404 des Headers geschrieben.

Durch Manipulation der 4 Bytes beginnend mit Offset 0x404 kann die Anzahl der Pages im Nachhinein abgeändert werden. Halbiert man beispielsweise diesen Wert und rundet ganzzahlig ab, wird nur noch die Hälfte des Swaps vom Betriebssystem genutzt. Dadurch entsteht ein “Quasi“ Volume Slack, der in der Partitionstabelle - im Gegensatz zu echten Volume Slacks - allerdings nicht sichtbar ist, sondern lediglich bei kritischer Inaugenscheinnahme des Swap Headers. Spricht man den darauffolgenden Speicher nun direkt über den Offset an, lassen sich dort Daten in einem zusammenhängenden Bereich ablegen.

Cryptsetup hat für die Anlage eines Blockgerätes mit dm-crypt/LUKS eine solche Funktion integriert, da ein reguläres LUKS-Gerät zunächst mit einem Header beginnt und der eigentliche Payload mit den verschlüsselten Daten an einem vorgegebenen - im Regelfall errechneten - Offset beginnt. Dieser Offset lässt sich bei der Initialisierung auch manuell festlegen und kann damit auch auf den “freigegebenen“ Bereich innerhalb des Swap gesetzt werden.

Nach außen hin sieht die Partition (bzw. Datei) dann immer noch wie ein regulärer Swap Bereich aus. Auch besteht der Inhalt nur aus verschlüsselten Daten, die keine Rückschlüsse auf deren wahre Herkunft schließen lassen. Es könnte sich hierbei ebenso um Binärdaten handeln, die von einem vormals ausgeführten Programm stammen und aus dem RAM “geswappt“ worden waren. Da im Swap Space keine dateisystemähnlichen Metadaten geschrieben werden, werden auch keine verräterischen Metadaten, insbesondere Zeitstempel gesetzt. Diese Tatsachen schaffen die Grundlage für eine plausible Abstreitbarkeit.

5 Implementierung der Software

Um die Software über verschiedene Linux-Derivate hinweg weitestgehend portabel zu halten, wird auf grundlegende Kommandozeilenbefehle der Basispakete `util-linux` und `coreutils` zurückgegriffen, da die dort enthaltenen Anwendungen in jeder populären Distribution zum Standardumfang gehören. Dazu gehören die Werkzeuge `lsblk` bzw. `stat` zur Abfrage von Informationen über Geräte bzw. Dateien, sowie `swapon`, `swapoff`, `swaplabel` und `mkswap`, zur Manipulation des Swap-Space. Hinsichtlich der Blockgeräteverschlüsselung wird auf `cryptsetup` zurückgegriffen, womit sich dm-crypt samt seiner Erweiterungen steuern lässt.
Diese externen Funktionsaufrufe werden über das (Sub-)Prozess-Management der Programmiersprache Python3 realisiert, welches auch die jeweiligen Rückgabewerte (stdin, stdout, stderr und returncodes) verarbeitet. Die Manipulation von Bilddateien erledigt letztendlich die Python Imaging Library.

5.1 Blockgeräteverschlüsselung mittels Integration von cryptsetup

Die Blockgeräteverschlüsselung wird in der Software mittels dm-crypt und dessen Erweiterung LUKS realisiert. Die Möglichkeit zur Verwendung eines “detached headers“, also die Trennung von Metadaten und den eigentlichen verschlüsselten Daten eignet sich hervorragend für die Umsetzung des Ziels der plausiblen Abstreitbarkeit.

Über die Syntax `cryptsetup [option] <action> <action-specific>` werden zahlreiche Möglichkeiten angeboten verschlüsselte Blockgeräte zu verwalten. Da die Syntax zum einen sehr durchdacht ist und zum anderen die erstellte Software ohnehin auf die LUKS-Funktionalität zurückgreift, soll sich die Syntax von Stingray weitgehend an der von Cryptsetup orientieren.

Um die grundlegenden Aktionen zum Anlegen, Öffnen und Schließen eines “Stingrays“ innerhalb der Software zu ermöglichen, wurde der Aufruf von Cryptsetup über die Funktionen `cs_create_luks`, `cs_detach_header`, `cs_open_luks` bzw. `cs_close_luks` realisiert. Hierbei wurden die Funktionen bewusst ohne direkten Bezug zur Software Stingray implementiert, um sie ohne Änderung auch in anderen Projekten verwenden zu können. Der Aufruf erfolgt stets über das Python3-Modul *subprocess*, um die drei Standard-Datenströme für Ein- und Ausgabe bzw. Fehlermeldungen (stdin, stdout, stderr) ansteuern zu können. Gemäß der Funktionsparameter wird die Syntax von cryptsetup zum Verwalten eines neuen Block- bzw. Schleifengerätes erstellt und schließlich ausgeführt. Bildschirmein- und -ausgaben werden mittels *subprocess* umgeleitet und es wird aus den entsprechenden Variablen gelesen bzw. in diese geschrieben.

Das nachfolgende Listing zeigt die Implementierung zum Anlegen eines neuen LUKS-Gerätes. Hierbei werden der Funktion die Parameter für das zu verwendende Blockgerät bzw. die zu verwendende Datei (*target*), die LUKS Version (*lukstype*), der Offset zum eigentlichen Payload (*offset*) und das zu verwendende Passwort (*password*) übergeben. Es besteht darüber hinaus die Möglichkeit zur Übergabe optionaler Schlüsselwortvariablen (***kwargs*).

Listing 5.1: Funktion zum Anlegen eines neuen LUKS-Gerätes

```
def cs_create_luks(target, lukstype, offset, password, **kwargs):
  pw = bytes(password, encoding='utf-8')
  cmd = ["cryptsetup", "luksFormat", target]
  if lukstype == 1:
    cmd.extend(["--type", "luks1"])
  if lukstype == 2:
    cmd.extend(["--type", "luks2"])
  cmd.extend(["--align-payload="+str(offset)])
  if lukstype == 2 and kwargs.get("downsize"):
    cmd.extend(["--luks2-metadata-size=16k","--luks2-keyslots-size=1988k"])
  x = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, input=pw)
  del pw
  return x
```

Die reguläre Größe eines LUKS2-Headers beträgt 16MB. Dies würde bei der späteren Überführung in die Bildsteganografie zu einer recht großen Zahl von Trägermedien führen oder sie müssten in einer recht großen Auflösung vorliegen. Um diesem Problem entgegenzuwirken, wird die Größe des LUKS2-Headers auf die von LUKS1 verwendete Größe geschrumpft; dies wird durch Verkleinerung des Schlüsselbereiches erreicht.
Um diese Funktionalität zu aktivieren, muss die Schlüsselwortvariable `downsize=True` übergeben werden. Dies kommt im Modus *default* zur Anwendung.
Im Modus *paranoid* wird der Schlüsselbereich nicht verkleinert, was das Verschlüsselungsverfahren grundsätzlich sicherer machen soll, jedoch ist für die Bildsteganografie hinreichendes Trägermaterial von Nöten. Dies betrifft Auflösung der Bilder, sowie deren Anzahl.

Das nachfolgende Listing zeigt die Implementierung zum Abtrennen des LUKS-Headers eines bereits angelegten Block- bzw. Schleifengerätes. Hierbei sei angemerkt, dass bereits beim Anlegen eines neuen LUKS-Gerätes die Verwendung eines “detached headers“ in einer separaten Datei erfolgen könnte, die Umsetzung allerdings nur bei LUKS1 vernünftig funktioniert; bei LUKS2 führt die gleichzeitige Verwendung von `--offset` und `--header` hier zu Problemen, was letztlich in einem übergroßen Header mündet. Deshalb wurde auf die elegantere Option `--header-backup-file` zurückgegriffen, um den Header im Nachhinein in eine Datei zu extrahieren.

Listing 5.2: Funktion zum Abtrennen des LUKS-Headers

```
def cs_detach_header(target, backup):
  cmd = ["cryptsetup", "luksHeaderBackup", target]
  cmd.extend(["--header-backup-file", backup])
  x = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

Um die wichtigsten Grundfunktionalitäten sicherzustellen, musste abschließend das Öffnen bzw. Schließen eines LUKS-Gerätes implementiert werden, was dem nachfolgenden Listing entnommen werden kann. Als Funktionsparameter werden das eigentliche LUKS-Gerät (*target*), sowie der später unter */dev/mapper* zu verwendende Name des virtuellen Blockgerätes (*device*) übergeben. Im Falle des Öffnens müssen darüber hinaus auch das Passwort und der Pfad zum abgetrennten Header übergeben werden. Für die Funktionen, bei denen seitens Cryptsetup eine interaktive Eingabe des Passworts vorgesehen ist, wird dieses beim Aufruf von `subprocess.run` mittels `input = pw` direkt übergeben, sodass der gesamte Funktionsaufruf ohne Nutzerinteraktion an einem Stück erfolgen kann.

Listing 5.3: Funktionen zum Öffnen eines bzw. Schließen eines bestehenden LUKS-Gerätes

```
def cs_open_luks(target, device, password, header):
  pw = bytes(password, encoding='utf-8')
  cmd = ["cryptsetup", "open", target, device]
  cmd.extend(["--header", header])
  x = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, input=pw)
  del pw
  return x

def cs_close_luks(target, device):
  cmd = ["cryptsetup","close",device]
  x = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
  return x
```

Schließlich geben alle Funktionen ein Objekt zurück, welches die von Cryptsetup produzierten Meldungen hinsichtlich Ausgabe und Fehlermeldungen enthält, um sie gegebenenfalls innerhalb von *Stingray* weiterverarbeiten zu können. Der im Normalfall erfolgreiche Durchlauf des Funktionsaufrufs wird im ebenfalls enthaltenen returncode gespeichert. Zwar bietet *Cryptsetup* darüber hinaus noch weitaus größere Funktionalität, wie beispielsweise das Hinzufügen, Ändern oder Löschen einzelner Schlüssel, jedoch sollte sich die erste Version der Software *Stingray* auf die Grundfunktionen beschränken.

5.2 Umsetzung der Steganografie

Die Software *Stingray* verfolgt zur Umsetzung der Steganografie unterschiedliche Ansätze zur Verschleierung der Existenz diverser Daten. Dies können sowohl verschlüsselte Daten oder deren Metadaten, wie auch für das Gesamtverfahren notwendige Metadaten sein. Im Vordergrund steht die Teilung von LUKS-Header und LUKS-Payload eines verschlüsselten Blockgerätes. Hierbei werden in den verschiedenen Software-Modi (default, legacy, paranoid und portable) diverse steganografische Ansätze verfolgt. Dabei werden weitere Metadaten generiert, die ihrerseits ebenfalls steganografisch verarbeitet werden müssen, um ein erfolgreiches Gesamtkonzept zu gewährleisten und die Nutzung der Software weitgehend transparent zu halten.

5.2.1 Datenträgersteganografie mittels Manipulation des Swap-Headers

Aufgrund der übersichtlichen Struktur des Swap-Space lässt sich dieser relativ gut zum Datenversteck umfunktionieren. Aktiviert man das 1 GB große Swap-File aus dem Beispiel im Abschnitt “Swap Space als dateisystemähnliche Datenstruktur“ erhält man folgendes Ergebnis:

```
# swapon --show
NAME            TYPE  SIZE   USED  PRIO
/tmp/swap1G     file  1024M  0B    -2
```

Ändert man nun manuell den Wert der letzten Page von `0x3ffff` (Dezimal: $2^{18} - 1$) auf `0x1ffff` (Dezimal: $2^{17} - 1$) und halbiert damit die Anzahl der Pages, erhält man folgendes Ergebnis:

```
# swapon --show
NAME            TYPE  SIZE  USED  PRIO
/tmp/swap1G     file  512M  0B    -2
```

Im gezeigten Beispiel wurde nur ein einzelnes Bit an Offset 0x406 des Swap-Headers verändert, wodurch das Byte 0x03 (binär: 00000011) zu 0x01 (binär: 00000001) wurde; die Wirkung ist allerdings enorm. Bei Prüfung der Dateigröße wird in beiden Fällen nach Aufruf von `stat -c %s swap1G` der Wert `1073741824` zurückgegeben, da sich die eigentliche Dateigröße von 1 GB durch die Manipulation natürlich nicht geändert hat. Das Betriebssystem nutzt allerdings nur noch die Hälfte des vorhandenen Speicherplatzes als Swap.

Zu Realisierung dieser Idee in *Stingray* mussten diverse Funktionen implementiert werden, um zunächst die aktuellen Informationen über den zu manipulierenden Swap zu erheben und selbige sodann verändern zu können.

Wie im nachfolgenden Listing ersichtlich, wird beim Aufruf von *swap_info* zunächst geprüft, ob es sich beim Swap um ein Blockgerät oder eine Datei handelt; die Größe wird sodann mit der entsprechenden Funktion abgerufen. Daraufhin werden Seitengröße und die letzte Seite bestimmt, die tatsächlich mögliche letzte Seite berechnet und die gesammelten Informationen schließlich zurückgegeben.

Listing 5.4: Funktion zum Auslesen der benötigten Informationen aus dem Swap

```
def swap_info(name):
  if stat.S_ISBLK(os.stat(name).st_mode): maxspace = get_devicesize(name)
  else: maxspace = get_filesize(name)
  pagesize = get_pagesize(name)
  maxpages = (maxspace-pagesize)//pagesize
  usepages = get_swappages(name)
  return name, maxspace, pagesize, maxpages, usepages
```

Listing 5.5: Helferfunktionen zur Bestimmung der Partitions- bzw. Dateigröße, sowie zum Auslesen der verwendeten Seitengröße und der letzten Seite

```
def get_devicesize(device):
  cmd = "lsblk -r -b -o PATH,SIZE | grep "+str(device)).split()[1]
  return int(subprocess.getoutput(cmd)

def get_filesize(file):
  cmd = "stat -c %s "+str(file)
  return int(subprocess.getoutput(cmd))

def get_pagesize(name):
  with open(name, "rb") as file:
    pagesize = file.read(64*1024).find(b"SWAPSPACE2")+10
  return pagesize

def get_swappages(name):
  with open(name, "rb") as file:
    file.seek(1028)
    pages = int.from_bytes(file.read(4), byteorder="little")
  return pages
```

Um die beiden Hauptfunktionen *swap_info* und *swap_modify* übersichtlich zu halten, erfolgt das Sammeln der Einzelinformationen über die vorstehenden Helferfunktionen. Die Größenabfrage einer Partition wird über `lsblk`, die einer Datei über `stat` realisiert. Zur Bestimmung der “pagesize“ wird ein Teil des Swaps eingelesen und nach dem String “SWAPSPACE2“ gesucht, der bekanntlich den Header abschließt. Die Angabe hinsichtlich der letzten Seite ist per Spezifikation auf Offset `0x404` (dezimal: 1028) festgelegt und wird direkt ausgelesen.

Sind alle Informationen vorhanden und ein Multiplikator festgelegt, kann die eigentliche Manipulation durch Aufruf von *swap_modify* erfolgen. Der Multiplikator wird auf Grundlage des vom Nutzer angegebenen Swap/Payload-Verhältnisses berechnet; er ist als Fließkommazahl zwischen 0 und 1 definiert. Möchte der Nutzer beispielsweise noch 25% der zur Verfügung stehenden Speicherplatzes weiterhin als Swap verwenden und die übrigen 75% zur Verwendung des Payloads freigeben, so wird der Multiplikator 0.25 übergeben.

Die Funktionsweise von *swap_modify* ist dabei recht simpel. Zunächst werden die benötigten Rahmeninformationen durch Aufruf von *swap_info* eingeholt, sodann die neue “letzte Seite“ berechnet und letztlich das Ergebnis an den vorgesehenen Offset geschrieben. Das nachfolgende Listing macht deutlich, wie elegant dies vonstatten geht.

Listing 5.6: Funktionen zur Manipulation des Swap-Headers hinsichtlich des zur Verfügung stehenden Speicherplatzes

```
def swap_modify(name, multi):
  name, maxspace, pagesize, maxpages, usepages = swap_info(name)
  newpages = int(maxpages*multi)
  with open(name, "r+b") as file:
    file.seek(1028)
    file.write(newpages.to_bytes(4, byteorder="little"))
  return name, maxspace, pagesize, maxpages, newpages
```

5.2.2 Bildsteganografie mittels Manipulation des Least-Significant-Bits

Bildsteganografie kommt in den Modi *default*, *legacy* und *paranoid* zur Anwendung, um die Existenz des LUKS-Headers steganografisch zu verschleiern. Hierbei wird der Header auf ggf. mehrere Trägerdateien aufgeteilt und mittels Manipulation des bzw. der Least-Significant-Bits eines jeden Pixelwertes eingebettet. Zur Manipulation der Bilddateien wird auf das Modul *PIL* zurückgegriffen. Die “Python Imaging Library“ wurde erstmals 1995 von Fredrik Lundh veröffentlicht.

Bevor mit der Codeimplementierung des bildsteganografischen Verfahrens selbst fortgefahren wird, sollte zunächst ein Augenmerk auf die Auswirkungen der Pixelmanipulationen gelegt werden. Dies lässt sich am besten anhand der nachfolgenden Beispielbilder feststellen; sie zeigen die Spitzen der Petronas-Towers in Kuala Lumpur, Malaysia.

Abbildung 5.1: Vergleich des Originalbildes mit den Ergebnissen nach Manipulation eines einzelnen bis hin zu allen acht Bits pro Pixelfarbwert

Zeilenweise von links oben nach rechts unten befinden sich das Original der Trägerdatei und ihm folgend die entstandenen Steganogramme bei Nutzung von einem einzigen (least significant) Bit, bis hin zu allen acht Bits pro Pixelfarbwert. Als Eingabe für die einzubettende Nachricht diente ein zufällig generierter Datenstrom fester Länge. Die Anzahl der veränderten Bits wird nachfolgend mit (LSB#) angegeben; beispielsweise bei Manipulation von drei der am wenigsten bedeutsamen Bits durch (LSB3).

Wie aus den Abbildungen ersichtlich unterscheidet sich das zweite Bild (LSB1) optisch nicht vom Original. Beim dritten Bild (LSB2) ist lediglich eine minimale Artefaktbildung im Hintergrund zu erkennen; jedoch nur beim direkten Vergleich mit dem Original.

In der zweiten Reihe (LSB3 bis LSB5) werden die Artefaktbildungen im Himmel deutlich sichtbarer, im kontrastreichen Vordergrund fallen die Manipulationen selbst bei Verwendung von fünf der acht Pixelbits erst bei genauerer Betrachtung ins Auge. Je nach Bildinhalt wäre die Nutzung von vier oder gar fünf Bits durchaus im Rahmen des Möglichen. Des weiteren entsteht nach Ende des Payloads eine harte Kante, die im mittleren Bild etwas auf Höhe der Hälfte bzw. beim rechten Bild am Übergang nach dem ersten Drittel sichtbar wird. Ab diesem Punkt folgen auf die Pixel mit eingebetteten Daten wieder Originalpixel, deren Farbübergänge naturgemäß fließender sind.

Die letzte Reihe (LSB6 bis LSB8) ist definitiv unbrauchbar, obgleich sich der Bildinhalt trotz massiver Manipulation noch teilweise erkennen lässt. Im letzten Bild besteht der obere Teil natürlich nur noch aus den eingebetteten Zufallsdaten, da alle Pixelbits überschrieben wurden. Dennoch wäre auch die Manipulation vieler Bits nutzbar, wenn der einzubettende Datenstrom sehr klein ist, so dass beispielsweise nur die erste Pixelreihe verändert würde.

Im Allgemeinen kann also festgestellt werden, dass die Manipulation eines einzelnen “Least-Significant-Bits“ sicherlich keinerlei Bedenken hervorrufen wird. Die Verwendung von zwei oder drei Bits pro Pixel kann bei sorgfältig gewählten Trägerdateien auch noch zum gewünschten Erfolg führen. Die Veränderung von vier oder mehr Bits ist dagegen grundsätzlich nicht mehr empfehlenswert.

Die Berechnung des maximalen Payloads eines PNG-Trägerbildes in Bytes errechnet sich durch das Produkt aus Breite, Höhe, Anzahl der Farbwerte pro Pixel und der Anzahl der verwendeten Least-Significant-Bits dividiert durch 8. In einem Desktop-Hintergrund mit einer Auflösung von 1920 x 1080 Pixeln und einer Farbtiefe von 24 Bit lassen sich bei Verwendung von einem bzw. zwei LSBs somit bereits 777600 Bytes bzw. 1555200 Bytes einbetten, ohne dass Bild dabei auffällig zu verändern. Nutzt man dagegen drei LSBs beträgt der maximale Payload bereits mehr als 2 MB, so dass der LUKS1- bzw. (verkleinerte) LUKS2-Header - dessen Größe jeweils 2020 KB beträgt - in einer einzigen Datei Platz fände.

Das nachfolgende Listing zeigt die Implementierung der LSB-Technik zum Einbetten eines Byte-Strings in ein Trägermedium im PNG-Format und einer Farbtiefe von 24 Bit. Der Funktion `lsb_encode` werden dabei der Pfad zur Trägerdatei, die Anzahl der zu verwendeten Bits, der Pfad zur Erstellung des Steganogramms und der einzubettende Datenstrom (Payload) übergeben. Zusätzlich besteht bei Bedarf die Möglichkeit zur Angabe von Schlüsselwortvariablen.

Listing 5.7: Funktion zum Einbetten der Daten in ein PNG

```
def lsb_encode(coverfile, lsb, stegofile, secretdata, **kwargs):
  with Image.open(coverfile) as cover: pixels = list(cover.getdata())
  lsb_ppx = 3 * lsb
  pl_data = "".join([bin(intbyte)[2:].rjust(8,"0") for intbyte in secretdata])
  crc_sum = bin(binascii.crc32(pl_data.encode('utf8')))[2:].rjust(32,"0")
  pl_size = bin(len(pl_data))[2:].rjust(28,"0")
  padding = lsb_ppx - len(crc_sum + pl_size + pl_data) % lsb_ppx
  padding = bin(random.randint(0,2**padding-1))[2:].rjust(padding,"0")
  payload = crc_sum + pl_size + pl_data + padding
  if kwargs.get("noise"):
    maxpayload = lsb_ppx * len(pixels)
    maxnoise = maxpayload - len(payload)
    noise = bin(random.randint(0, 2**maxnoise-1))[2:].rjust(maxnoise,"0")
    payload = payload + noise
  payload_list = []
  for i in range(0, len(payload), lsb_ppx):
    lsb_list = []
    for j in range(0, lsb_ppx, lsb):
      lsb_list.append(payload[i+j:i+j+lsb])
    payload_list.append(lsb_list)
  for i in range(len(payload_list)):
    r = bin(pixels[i][0])[2:].rjust(8,"0")
    g = bin(pixels[i][1])[2:].rjust(8,"0")
    b = bin(pixels[i][2])[2:].rjust(8,"0")
    r = int(r[:-lsb]+payload_list[i][0],2)
    g = int(g[:-lsb]+payload_list[i][1],2)
    b = int(b[:-lsb]+payload_list[i][2],2)
    pixels[i] = (r, g, b)
  stegoimage = Image.new("RGB", cover.size)
  stegoimage.putdata(pixels)
  if kwargs.get("show"): stegoimage.show()
  else: stegoimage.save(stegofile, optimize=True, quality=100)
```

Zunächst wird das Trägermedium (`coverfile`) über das PIL-Modul geöffnet und die einzelnen Pixel in einer Liste von Tupeln der Werte rot, grün und blau abgelegt. Danach wird der Datenstrom (`secretdata`) in einen binären String überführt, dessen Prüfsumme mittels Zyklischer Redundanzprüfung (CRC) gebildet und die Gesamtlänge des Payloads berechnet.
Da die Größe des Payloads bei Erreichen des letzten genutzten Pixels nicht zwingend dessen gesamten Speicherplatz einnimmt, erfolgt hier ein Padding mit Zufallszahlen. Dieses “nur teilweise gefüllte letzte Pixel“ wäre vergleichbar mit einem File-Slack bei Dateisystemen.
Schließlich wird der Gesamtpayload gebildet; er besteht aus Prüfsumme (`crc_sum`), Länge (`pl_size`), binären Daten (`pl_data`) und den Füllzeichen (`padding`). Letztere werden im Übrigen beim Dekodieren ignoriert, da die Länge des ursprünglichen Datenstroms bekannt ist.

Nach Aufbereitung der Daten folgt das eigentliche Kodieren. In einer Schleife werden die Originaldaten Pixel für Pixel und pro Farbwert um die Anzahl der zu verwendenden Bits gekürzt, an deren Stelle die Daten des Payloads angehängt und die manipulierten Werte wieder in der Variable (`pixels`) abgespeichert. Nach Durchlaufen der Schleife wird das Steganogramm am angegebenen Pfad (`stegofile`) abgespeichert. Durch die Angabe der Schlüsselwortvariable `show` wird das Ergebnis nicht in eine Datei geschrieben, sondern ein Vorschaubild erzeugt. Somit hat der Nutzer die Möglichkeit, das potentielle Steganogramm vorab zu beurteilen, um die Auswahl nochmals zu ändern.

Zusätzlich wurde die Schlüsselwortvariable `noise` implementiert, die bei der Bildsteganografie in der Software eine sehr wichtige Rolle spielt, wenn die Anzahl der manipulierten Bits höher ausfällt. Wie auf den Beispielbildern zum LSB-Vergleich bereits ersichtlich war, ergeben sich nach Ende des Payloads “harte Übergänge“. Diese sind ein offensichtliches Indiz dafür, dass am Bild nachträgliche Veränderungen durchgeführt worden waren. Durch das Hinzufügen eines Rauschens (noise) durch Zufallswerte wird dieser Übergang unsichtbar, da nun jedes Pixel eine Veränderung erfährt. Die nachfolgende Abbildung verdeutlicht diesen Umstand visuell.

Abbildung 5.2: Vergleich erzeugter Steganogramme mit und ohne hinzugefügtes Rauschen

In der oberen Reihe ist der Übergang zwischen veränderten und unveränderten Pixeln deutlich sichtbar, während die untere Reihe durchgängig Artefakte aufweist. Bei letzteren könnte es sich theoretisch auch um schlechte oder stark komprimierte Bildinhalte handeln. Auch dieses künstlich erzeugte Rauschen wird wie das Padding beim Dekodieren ignoriert.

Für die Software Stingray ist das hinzugefügte Rauschen gerade bei höher gewählter Anzahl der zu verwendenden Bits essentiell, da die Manipulation zumindest beim letzten Bild sofort auffallen und genauere Untersuchungen aller Bilder nach sich ziehen könnte.

Das nachfolgende Listing zeigt die Implementierung der LSB-Technik zum Extrahieren eines Byte-Strings aus ein Trägermedium im PNG-Format und einer Farbtiefe von 24 Bit. Der Funktion `lsb_decode` werden dabei der Pfad zum Steganogramms und die Anzahl der zu verwendenden Bits übergeben. Ist der Dekodiervorgang erfolgreich, wird der entsprechende Datenstrom zurückgegeben; andernfalls wird Fehlermeldung generiert.

Listing 5.8: Funktion zum Extrahieren der Daten aus einem PNG

```
def lsb_decode(stegogram, lsb):
  with Image.open(stegogram) as stego: pixels = list(stego.getdata())
  lsb_ppx = 3 * lsb
  data = []
  for pixel in pixels:
    r = bin(pixel[0])[2:].rjust(8,"0")
    g = bin(pixel[1])[2:].rjust(8,"0")
    b = bin(pixel[2])[2:].rjust(8,"0")
    data.append(r[-lsb:]+g[-lsb:]+b[-lsb:])
  data = "".join(data)
  crc_sum = data[:32]
  pl_size = data[32:60]
  pl_size = int(pl_size,2)
  if pl_size > lsb_ppx * len(pixels) - 60:
    return (False, "decoding error")
  pl_data = data[60:60+pl_size]
  if crc_sum != bin(binascii.crc32(pl_data.encode('utf8')))[2:].rjust(32,"0"):
    return (False, "checksum error")
  pl_data = int(pl_data, 2).to_bytes(len(pl_data)//8, byteorder='big')
  return (True, pl_data)
```

Diesmal wird das Steganogramm (`stegogram`) über das PIL-Modul geöffnet und die einzelnen Pixel abermals in einer Liste mit Tupeln der Werte rot, grün und blau abgelegt. In einer Schleife werden Pixel für Pixel jeder Farbwert in die binäre Form überführt und die übergebene Anzahl der Bits (von rechts) in eine Liste geschrieben. Nach Durchlaufen der Schleife wird die Liste zu einen binären String zusammengeführt und dieser entsprechend der Spezifikation in eine Prüfsumme (`crc_sum`, Länge: 32 Bit), Größe des Payloads (`pl_size`, Länge: 28 Bit) und den übrigen Datenstrom (`pl_data`) aufgeteilt.

Für den Fall, dass die dekodierte Länge des Payloads die maximal mögliche Größe übersteigt, gibt die Funktion eine entsprechende Fehlermeldung (“decoding error“) zurück. Ebenso verhält es sich, wenn die berechnete Prüfsumme nicht mit der eingebetteten übereinstimmt (“checksum error“).

Treten keine Fehler auf, wird der Datenstrom (`pl_data`) auf die tatsächliche Länge gekürzt und in Form eines Byte-Strings zur weiteren Verarbeitung zurückgegeben.

5.3 Umsetzung diverser Kodierungsansätze

5.3.1 Kodierung von Daten durch Kompression

Die Header von LUKS1, wie auch LUKS2 weisen, wie bereits im Kapitel Verschlüsselungsverfahren angemerkt, eine erkennbare Datenstruktur auf. Am Beispiel von LUKS1 befinden sich in den ersten 4 KB grundlegende Metadaten, wie das verwendete Verschlüsselungsverfahren oder die Keyslots. Da diese Daten nicht unverändert in die Bildsteganografie überführt werden sollten, bedarf es zumindest einer vorherigen Kodierung, die durch Kompression des Headers erreicht wird.

Mit dem Python-Modul `zlib` wird ein gzip-komprimierter Datenstrom erstellt, der den eigentlichen LUKS-Header enthält. Die maximale Kompression erreicht man mittels `compressed_data = zlib.compress(binarydata, level=9)`. Durch Dekompression mittels `decompressed_data = zlib.decompress(compressed_data)` lässt sich der LUKS-Header wiederherstellen. Er kann sodann zur weiteren Verwendung durch Cryptsetup in eine Datei geschrieben werden.

Obgleich durch die Kompression eine - wenn auch nur marginale - Reduzierung des Speicherbedarfs einhergeht, dient das Verfahren hier in erster Linie der Kodierung, um eine mögliche Entdeckung zu verhindern. Bei durchgeführten Test konnte festgestellt werden, dass das eigentlich komprimierte Ergebnis auch mehr Speicherbedarf benötigen kann als das Original. Aus diesem Grund wird die Kodierung von Daten durch Kompression nur in den bildsteganografischen Modi `default`, `legacy` und `paranoid` angewandt. Im Modus `portable` wird auf ein anderes Verfahren zurückgegriffen, welches im folgenden Abschnitt erläutert wird.

5.3.2 Kodierung von Daten mittels XOR

Werden Dateien auf einem Datenträger unverschlüsselt gespeichert, bleibt deren Inhalt selbst nach dem Löschen noch sichtbar; zumindest solange die Daten nicht anderweitig überschrieben werden. Dateien auf Speichermedien können auch ohne Kenntnis des Dateisystems identifiziert und ggf. wiederhergestellt werden. In der IT-Forensik nennt man diese Methode *Carving*. Dabei werden die Rohdaten nach charakteristischen Zeichenfolgen wie “magic bytes“ oder “Header-Signaturen“ bekannter Dateiformate durchsucht um den Inhalt einer potentiell nachfolgenden Datei zu rekonstruieren.

Im Modus *portable* wird die Existenz des LUKS-Headers im Gegensatz zu den übrigen Modi nicht mittels Bildsteganografie verschleiert, sondern direkt auf dem Datenträger innerhalb des Swap-Space abgelegt. Um eine Entdeckung der LUKS-Signatur zu verhindern, wird der gesamte Datenstrom des Headers vor dem Abspeichern mittels XOR manipuliert.

Die vier möglichen Ergebnisse der Exklusiv-Oder-Funktion bei binären Eingaben sind in der nachfolgenden Tabelle abgebildet. Ist genau eine Eingabe 1 (True), so ist auch die Ausgabe 1 (True). Ist dahingegen keine oder mehr als eine Eingabe 1 (True), so ist die Ausgabe 0 (False). Man spricht hierbei von einer logischen Antivalenz zwischen den Eingangszuständen.

Tabelle 5.1: XOR Wahrheitstabelle

Eingabe A	**Eingabe B**	**Ausgabe**
0	0	0
0	1	1
1	0	1
1	1	0

Da die XOR-Funktion bitweise orientiert arbeitet, ist sie besonders performant und lässt sich darüber hinaus recht leicht implementieren. Sie eignet sich daher besonders gut innerhalb eines Verschlüsselungsverfahrens oder generell zur Kodierung.

Natürlich lässt sich die Antivalenz auch Byteweise umsetzten, um Daten zu kodieren. Eine modifizierte Form des byteweisen XOR wird durch die Software genutzt, da der Header ohnehin bereits als Bytestring vorliegt.

Listing 5.9: “modified-XOR“-Funktion zur Verschleierung von Datei- oder Partitionssignaturen

```
def modxor(bytestr):
  flags = [0,255]
  return bytes([i^255 if i not in flags else i for i in bytestr])
```

Die “modified-XOR“-Funktion (`modxor`) führt grundsätzlich auf jedes Eingangsbyte ein byteweises XOR mit `0xff` (255) durch; allerdings mit der Ausnahme, dass im Falle der Eingangbytes `0x00` oder `0xff`, keine Veränderung stattfindet. Hintergrund dessen ist die Tatsache, dass Null-Bereiche auf Datenträgerebene nicht verändert werden sollen, da diese bei forensischen Untersuchungen i.d.R. keinerlei Beachtung finden. Somit darf `0xff` auch nicht verändert werden, da es sich beim byteweisen XOR mit `0xff` auf `0x00` abbildet und die Dekodierung sodann fehlschlagen würde. Alle anderen Eingangsbytes - von `0x01` bis `0xfe` - werden auf ihr entsprechendes Gegenstück “umgekippt“. Auf diese Weise können Datei- oder Header-Signaturen nicht mehr gefunden werden.

Am Beispiel des LUKS-Headers, der mit der Sequenz `4c 55 4b 53` (LUKS) beginnt, wird diese nach Anwendung von *modxor* zu `b3 aa b4 ac` und damit beim Carving nicht mehr als Signatur erkannt.

5.3.3 Kodierung von Metadaten im Universally Unique Identifier

Nachdem Header und Payload schließlich getrennt voneinander und an den vorgesehenen Orten ihren Platz gefunden haben, müssen diese Informationen sinnvoll abgespeichert werden, damit der Nutzer die Software auch ohne Kenntnis dieser zusätzlichen Metadaten verwenden kann. Ziel ist es schließlich, das Verfahren trotz Hinzufügen der Steganografiekomponente nicht unnötig komplizierter zu machen als dies bei der alleinigen Verwendung von dm-crypt/LUKS der Fall wäre.

In allen Modi ist der Offset zum Payload im LUKS-Header standardmäßig integriert. Liegt der LUKS-Header also (dekodiert) vor, so lässt sich das Gerät mittels Cryptsetup öffnen. Allerdings liegt der Header lediglich in steganografischer und kodierter Form vor, so dass zu dessen Vorverarbeitung die entsprechenden Metadaten bekannt sein müssen. In allen Modi werden diese Informationen in den Universally Unique Identifier (UUID) des Swap kodiert und müssen von dort zunächst ausgelesen werden.

Der UUID besteht stets aus einer hexadezimal notierten Zahl mit einer Länge von 128 Bit und wird standardmäßig bei Initialisierung eines neuen Swap-Space zufällig erzeugt. In Normalnotation wird der UUID in fünf Gruppen unterteilt und könnte folgendermaßen aussehen: `9e223eec-f815-4a19-b490-9182be27b67a`

Da beim Modus *portable* der Offset zum modxor-kodierten LUKS-Header nicht bekannt ist, muss dieser in geeigneter Form gespeichert werden. Dazu wird er in die letzte Gruppe des UUID codiert, der Teil `9182be27b67a` aus dem Beispiel wird also mit der Funktion `uuid_encode_offset` aus dem nachfolgenden Listing entsprechend abgeändert.

Listing 5.10: Funktionen zum Kodieren eines Offsets in den UUID

```
def uuid_encode_offset(uuid, pagesize, offset):
  if offset > 2**44-1: return None
  modifier = sum([int("0x"+i, 16) for i in uuid.split("-")[1:4]])
  string = hex(int(offset/pagesize)+modifier)[2:]
  length = hex(len(string))[2:]
  code = string + length
  return uuid[:-len(code)]+code
```

Übergeben werden die Parameter *uuid*, *pagesize* und *offset* aus denen der neue UUID berechnet wird. Der Offset wird zunächst durch die Größe einer Swap-Page geteilt und mit dessen Länge konkateniert. Der UUID wird schließlich rechtsbündig mit dem kodierten String überschrieben. Da der kodierte Offset nicht in Bytes sondern in Größe einer Swap-Page angegeben wird und dafür elf hexadezimale Stellen zur Verfügung stehen, läge der höchste kodierbare Offset bei $2^{44} - 1$ also bei 64 PB; das dürfte für den “Hausgebrauch“ genügen. Um den Wert noch etwas zusätzlich zu verschleiern werden die mittleren Gruppen des UUID in die Berechnung mit einbezogen und deren Summe auf den tatsächlichen Offset aufaddiert. Sollte nun tatsächlich versucht werden, den Offset lediglich über die letzte Gruppe zu rekonstruieren, würde man mit einem Versatz von bis zu 768 MB “im Trüben fischen“.

Das nachfolgende Listing dekodiert den Offset aus dem UUID wieder; das Verfahren läuft in umgekehrter Reihenfolge ab, sodass bei Übergabe von *uuid* und *pagesize* ein passender Offset zurückgegeben wird, sollten die Rahmenbedingungen - wie die Länge des UUID selbst und des kodierten Offsets - erfüllt sein.

Listing 5.11: Funktionen zum Dekodieren eines Offsets aus dem UUID

```
def uuid_decode_offset(uuid, pagesize):
  if len(uuid) != 36: return None
  length = int(uuid[-1], 16)
  if length < 1 or length > 11: return None
  modifier = sum([int("0x"+i, 16) for i in uuid.split("-")[1:4]])
  code = int(uuid[-1-length:-1],16)
  offset = (code-modifier)*pagesize
  if offset < 0: return None
  return offset
```

Mit den nachfolgenden Funktionen kann der UUID eines Swap ausgelesen und letztendlich in kodierter Form wieder gesetzt werden.

Listing 5.12: Funktionen zur Auslesen und Setzen des Swap-UUID

```
def get_swap_uuid(devicename):
  return subprocess.getoutput("swaplabel %s | grep UUID" % devicename).split()[1]

def set_swap_uuid(devicename, uuid):
  return subprocess.run(["swaplabel",devicename,"-U",uuid]).returncode
```

In den bildsteganografischen Verfahren (`default`, `legacy`, `portable`) bestehen die zusätzlichen Metadaten lediglich aus der Anzahl der verwendeten LSBs. Da für den portablen Modus bereits die hexadezimalen Werte 1 bis b reserviert sind, stehen noch die Werte 0 bzw. c bis f zur Verfügung. Letztere werden zur Kennzeichnung der Anzahl bildsteganografisch modifizierter Bits verwendet, indem zur tatsächlichen Anzahl 11 addiert wird. So werden 1, 2, 3 bzw. 4 durch c, d, e bzw. f kodiert. Zwar wäre die Verwendung von mehr als vier Least-Significant-Bits technisch möglich, dies kommt in der Software *Stingray* allerdings nicht zur Anwendung, da die Veränderungen der Bildinhalte zu erheblich wären. Die Modifikation des UUID erfolgt durch den simplen Aufruf von `set_swap_uuid(target, get_swap_uuid(target)[:-1]+hex(lsb+11)[2:])`. Dabei wird lediglich das letzte Zeichen des UUID durch dessen kodiertes Pendant ersetzt, wobei *target* das Swap-Gerät und *lsb* die Zahl der verwendeten Bits sei.

Durch die Kodierung von Metadaten im UUID des Swap-Space ist es nun möglich, die Software *Stingray* innerhalb des gegebenen Funktionsumfangs analog zum regulärem Cryptsetup zu nutzen, ohne das zusätzliche Informationen oder Passwörter benötigt werden. Der Nutzer muss darüber hinaus allenfalls die verwendeten Bilder (und ggf. deren Reihenfolge) kennen, sollte nicht der portable Modus verwendet worden sein. Während man sich Bilder noch leicht merken bzw. sich eine “Eselsbrücke“ bauen kann, lässt sich ein Offset wohl kaum im Gedächtnis behalten. Durch diese Methodik der Datenkodierung - die im Übrigen eine Form der Textsteganografie darstellt - wird dies auf effektive Weise obsolet.

6 Evaluation der Software

Die Software *Stingray* wurde als “Proof-of-Concept“ einwickelt, um plausible Abstreitbarkeit durch die Kombination von Steganografie und Kryptografie zu erreichen. Wie bei einem Prototypen üblich, wird es immer Bereiche geben, in denen noch “Luft nach oben“ ist. Hierzu wurde die Software einer Evaluation unterzogen und hinsichtlich Benutzerfreundlichkeit bzw. Bedienkomfort, ihrer derzeit bestehenden Einschränkungen und einer generell möglichen Erweiterbarkeit bzw. der Implementierbarkeit neuer Aspekte, die über die Basisfunktionalität hinausgehen untersucht. Abschließend folgt ein Vergleich mit True-/Vera-Crypt, wo ein ähnliches Konzept der plausiblen Abstreitbarkeit verfolgt wird.

6.1 Benutzerfreundlichkeit

Die Software *stingray* verzichtet vollständig auf ein eine grafische Benutzeroberfläche (GUI) . Die Interaktion mit dem Nutzer erfolgt ausschließlich über die Kommandozeile bzw. ein Command Line Interface (CLI) .

Bei der Planung stand die Anlehnung an die Syntax von Cryptsetup im Vordergrund. Es sollte eine Software entstehen, welche die Blockgeräteverschlüsselung von dm-crypt aufgreift und um steganografische Merkmale bzw. Eigenschaften erweitert, ohne dabei das Rad neu zu erfinden. Ist dem Nutzer die durchdachte Syntax von Cryptsetup bereits geläufig, so wird er Stingray ebenfalls intuitiv bedienen können.

Bereits mit dem Prototyp, der bewusst zunächst nur Basisfunktionalität bereitstellt, ist es möglich zahlreiche Parameter zu übergeben, die den interaktiven Modus des CLI auf ein Minimum reduzieren; die Parameter können jedoch ebenso gut zur Laufzeit übergeben werden. Überdies ist durch zahlreiche Plausibilitätsprüfungen während der Laufzeit ein reibungsloser Programmablauf gewährleistet.

Bei Stingray handelt es sich ebenso wie bei Cryptsetup keineswegs um ein “Rookie-Tool“. Es kann davon ausgegangen werden, dass ein potentieller Nutzer, der sich seinem Handeln bewusst ist auch mit einem CLI umzugehen vermag.

Aufgrund des modularen Aufbaus ließe sich sicherlich eine grafische Benutzeroberfläche implementieren, was nach aktuellem Entwicklungsstand allerdings nicht angedacht ist. Dennoch steht es jedem Nutzer bzw. Entwickler seinen Programmierkünsten freien Lauf zu lassen; Stingray ist schließlich Open Source.

6.2 Kompatibilität mit Cryptsetup

Stingray greift zum Initialisieren eines neuen verschlüsselten Blockgerätes sowie bei dessen Öffnen und Schließen auf Cryptsetup zurück. Die Trennung von Metadaten und den chiffrierten Daten wird durch die Abtrennung des LUKS-Headers erreicht, der letztlich in ein Steganogramm überführt wird.

Dadurch besteht zunächst keine direkte Kompatibilität mit Cryptsetup, da die Header-Daten nicht mehr lesbar sind. Sollte ein mit Stingray initialisiertes Blockgerät aus irgendeinem Grund mit Cryptsetup geöffnet werden müssen, wird ein Backup des Originalheaders in Dateiform benötigt. Ein solches Szenario wäre beispielsweise, wenn Stingray nur beim Transit der Daten eingesetzt werden soll und die steganografischen Aspekte in einer sicheren Umgebung nicht mehr relevant sind.

Hierzu wurde eine Funktion implementiert, welche die Daten aus den Bildsteganogrammen wiederherstellt und als regulären “detached header“ abspeichert. Somit kann der verschlüsselte Container auch mittels Cryptsetup geöffnet und verwaltet werden. Die Tatsache, dass es sich bei dem verwendeten Container tatsächlich um Swap-Space handelt ist grundsätzlich nicht relevant. Es muss lediglich exklusiver Zugriff auf die Datei bzw. Partition bestehen; sie darf also nicht als Swap verwendet werden. Dies wird von Stingray automatisch überwacht und bei Bedarf ein `swapoff` durchgeführt; nutzt man das Header-Backup mit Cryptsetup ist dieser Schritt manuell durchzuführen. Natürlich kann die Header-Datei auch als reguläres Backup verwendet werden, für den Fall einer Veränderung oder Zerstörung der Steganogramme.

6.3 Einschränkungen

Derzeit wird hinsichtlich der Bildsteganografie auf im PNG-Format vorliegende Rastergrafiken zurückgegriffen. Dies hat sowohl Vor- als auch Nachteile. Ein offensichtlicher Nachteil ist natürlich die Tatsache, dass JPEG-Bilder nicht verwendet werden können. Sie weisen allerdings auch bei gleicher Auflösung die geringere Payload-Kapazität vor, was PNG wiederum interessanter macht. Feldversuche mit Software, die JPEG-Steganografie beherrschen - wie beispielsweise *steghide*, haben gezeigt, dass für einen Payload der Größe eines LUKS1-Headers bei einer Auflösung von 4000 x 3000 Pixeln zwischen 6-8 Bilder nötig wären. Stingray könnte in ein einzelnes PNG dieser Auflösung den Header bereits zweifach einbetten; und das bei Änderung von lediglich einem einzelnen Bit pro Pixel. Um den 16 MB großen LUKS2-Header aufzunehmen würden lediglich zwei dieser Bilder bei Verwendung von 2 Pixelbits benötigt; während bei JPEG-Steganografie die Zahl deutlich wächst.

Ein zweiter Aspekt sind - gerade aus forensischer Sicht - aktuell die Zeitstempel der Steganogramme. Bei Einbettung des Payloads werden die Dateien überschrieben und damit auch neue Zeitstempel gesetzt. Dem ließe sich sehr einfach abhelfen, indem der Linux-Befehl `touch` auf alle Bilder im Verzeichnis nach dem Einbettungsvorgang ausgeführt wird. Diese Variante ist allerdings nicht optimal, da damit alle Dateien (quasi) den selben Zeitstempel haben würden. Sind in den Trägerdateien allerdings EXIF-Daten - Metadaten, die weitere Informationen über das Bild, wie Kameramodell oder Aufnahmedatum speichern - vorhanden, so könnte dies Fragen aufwerfen.
Doch auch für solche Zwecke stellt die Programmiersprache Python entsprechende Funktionen bereit. Mit `os.utime` können die Zeitstempel der letzten Änderung (modifiedtime) bzw. des letzten Zugriffs (accesstime) verändert werden. Die entsprechenden Werte müssen hierzu mittels `os.stat` vorab ausgelesen werden. Da Zeitstempel aus forensischer Sicht sehr wichtig sind, wird eine passende Implementierung in einer künftigen Software-Version folgen.

Eine dritte Einschränkung ist auf die Modifikation des Swap-UUIDs zurückzuführen. Sowohl bei den bildsteganografischen Modi (Kodierung der verwendeten LSBs) als auch bei der portablen Variante (Kodierung des Offsets) wird die UUID grundsätzlich modifiziert. Dies ist unter einer Bedingung allerdings problematisch, nämlich genau dann, wenn die Zuordnung der Einhängeorte (moint points) über den UUID erfolgt und nicht über die vom Linux-Kernel zugeteilte Gerätedatei. Dies ist gerade dann relevant, wenn die modifizierte Swap-Partition tatsächlich von diesem oder einem anderen System weiterhin als solche genutzt werden soll.
Derzeit muss eine entsprechende Änderung der Datei `/etc/fstab` noch manuell erfolgen. Bei Swapfiles ist dies im Übrigen unproblematisch, da diese nicht über den UUID eingehängt werden können. Dahingehend wird empfohlen im Zweifelsfall auf Swapfiles zurückzugreifen.

6.4 Erweiterbarkeit

Hinsichtlich des Open-Source-Gedankens und der modularen Bauweise bietet die Software reichlich Potential hinsichtlich möglicher Erweiterungen bzw. Funktionalitäten in künftigen Versionen.

Da das Hauptaugenmerk sicherlich auf den steganografischen Aspekten liegt, wäre hier an alternative Verfahren zu denken. Zwar ist die Nutzung von JPEGs durch die Komplexere Implementierung der diskreten Kosinustransformation vergleichsweise aufwendig, doch wurden von den namhaften Herstellern moderner Smartphones mittlerweile Geräte mit einer Kameraauflösung jenseits der 48 Megapixel auf den Markt gebracht. Dadurch werden auch in Richtung JPEG die nötigen Payload-Kapazizäten erreicht, die nicht mehr zu einer allzu großen Menge an Trägerdateien führen würden.

Doch auch auf Datenträgerseite ließe sich über die Implementierung von steganografischen Funktionen nachdenken, z. B. Manipulationen an HPA und DCO durchführen, um sichere Datenverstecke zu erstellen. Diese Variante wäre besonders interessant für den portablen Modus, bei dem vermutlich am ehesten noch auf Magnetfestplatten zurückgegriffen werden wird.

Bereits bei Einführung von LUKS war es möglich bis zu acht verschiedene Schlüssel zu nutzen. Über eine zusätzliche "Modifizier-Funktion" ließen sich diese Keyslots nach Erstellung eines *"Stingrays"* mit weiteren Schlüsseln belegen. Dies wäre für alle Modi durchaus interessant, da die Steganogramme nicht zwangsweise dauerhaft lokal vorgehalten werden müssen, sondern auch bei einem Cloudspeicherdienst gehostet sein könnten. Hierbei sollte dann aber zwingend über eine Überschlüsselung der Payloads statt einer einfachen Kodierung nachgedacht werden, da der Sicherheitsaspekt hier eine größere Rolle spielt.

Die zusätzliche Überschlüsselung - beispielsweise mittels OpenSSL oder GnuPG - statt einer Kodierung, die ja in erster Linie der Signaturverschleierung dient, wirft aber ein neues Problem auf. Für die Überschlüsselung wird ein weiteres Passwort benötigt, so dass am Ende zwei Passwörter eingegeben werden müssen, was eigentlich vermieden werden sollte. Zwar könnte man zweimal dasselbe Passwort benutzen, sodass die Software den Rest erledigt und man nur eine Eingabe braucht; dann wäre allerdings die Multi-Key-Funktoinalität dahin.

Eine Lösung wäre die Verwendung eines zusätzlichen Trägerbildes, dass allerdings nicht in die Bildsteganografie mit einbezogen wird, sondern das Passwort tatsächlich ein zufällig gewählter Datenstrom dieses zusätzlichen Bildes ist. Das letzte (oder erste) Bild wird also gar nicht "steganografiert", sondern dient lediglich der Überschlüsselung aller Payloads. Um hier ein geeignetes Verfahren zu finden, wären allerdings einige Tests von Nöten gewesen, was den Umfang dieser Arbeit sicherlich gesprengt hätte.

Die Manipultion des Swap-Headers könnte hinsichtlich der Verkleinerung des zur verfügung stehenden Speichers Fragen aufwerfen, auch wenn dadurch das gesamte Verfahren nicht zwangsweise kippen sollte. Dennoch ließe sich darüber nachdenken, den Swap-Header wieder auf den Urzustand zu bringen, da die Manipulation nicht für Stingray an sich relevant ist; sie dient lediglich als Schutzmaßnahme. Man müsste dann nämlich auch verhindern, dass der Swap-Space tatsächlich genutzt wird, da sonst die Gefahr eines Überschreibens der chiffrierten Daten und damit die Zerstörung darin befindlichen Dateisystems bestünde.

Zuletzt ließe sich natürlich auch für die weniger versierten potenziellen Nutzer eine trendige grafische Oberfläche mit reichlich Schaltflächen und einer narrensicheren Hilfeseite gestalten, doch das wird sicherlich auch in künftigen Versionen nicht der Fall sein.

6.5 Vergleich mit Truecrypt/Veracrypt hinsichtlich der plausiblen Abstreitbarkeit

Durch TrueCrypt bzw. dessen inoffiziellen Nachfolger VeraCrypt wird ein ähnlicher Ansatz der plausiblen Abstreitbarkeit verfolgt. Umgesetzt wird dies von beiden Programmen durch sogenannte “Hidden Volumes“ also versteckte Container. Ähnlich wie bei dm-crypt/LUKS kann VeraCrypt eine Partition bzw. Container-Datei zu einem Krypto-Laufwerk initialisieren, dessen Inhalt erst nach Eingabe des korrekten Passwortes als virtuelles Laufwerk bereitgestellt wird und somit transparent auf die Daten zugegriffen werden kann. Entsprechende Header-Daten werden dabei sowohl von dm-crypt/LUKS als auch von VeraCrypt geschrieben um das Volume zu öffnen.

Die Hidden Volumes werden innerhalb des freien Speichers eines bestehenden Containers angelegt; sie sind also vereinfacht gesagt ein virtuelles Laufwerk in einem virtuellen Laufwerk. Da VeraCrypt standardmäßig freien Speicherplatz eines regulären Containers mit zufälligen Daten füllt, kann ein Außenstehender nicht beurteilen, ob es sich bei den Daten um einen versteckten Container handelt oder tatsächlich nur freien Speicher. Das Hidden Volume wird mit Hilfe eines separaten Passwortes entsperrt, sodass der Zugriff entsprechend des eingegebenen Passworts entweder auf den äußeren oder auf den inneren Container erfolgen kann. Die plausible Abstreitbarkeit wird also durch die Tatsache erreicht, dass bei Eingabe des Passworts für den verschlüsselten (äußeren) Container keine sensiblen Daten gefunden werden und die Existenz eines Hidden Volumes nicht überprüft wird. [23]

In einem 2017 veröffentlichten Artikel mit dem Titel “Defeating Plausible Deniability of VeraCrypt Hidden Operating Systems“ konnte gezeigt werden, dass die Existenz eines Hidden Volumes (bei geöffnetem äußeren Container) durch Analyse der Gleichverteilung - die bei modernen Verschlüsselungsverfahren stets gegeben sein sollte - relativ leicht bewiesen werden kann. An bestimmten Stellen, wo sich eigentlich Zufallsdaten befinden müssten, finden sich statische Daten wieder, die den Beginn und das Ende eines Hidden Volumes markieren; die plausible Abstreitbarkeit fliegt damit auf. [24]

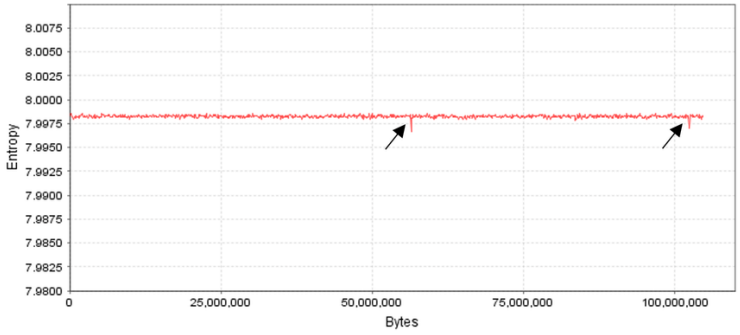

Abbildung 6.1: Bereiche mit deutlich geringerer Entropie im äußeren VeraCrypt-Container (Quelle: `https://www.researchgate.net/publication/318155607`)

Die nachfolgende Abbildung zeigt den schematischen Aufbau eines VeraCrypt Hidden Volumes. Die (grauen) Bereiche vor und nach dem Datenbereich des Hidden Volumes (blau) beinhalten nach dessen Erstellung Metadaten, die zu den Abweichungen in der Entropie der vorherigen Abbildung führen.

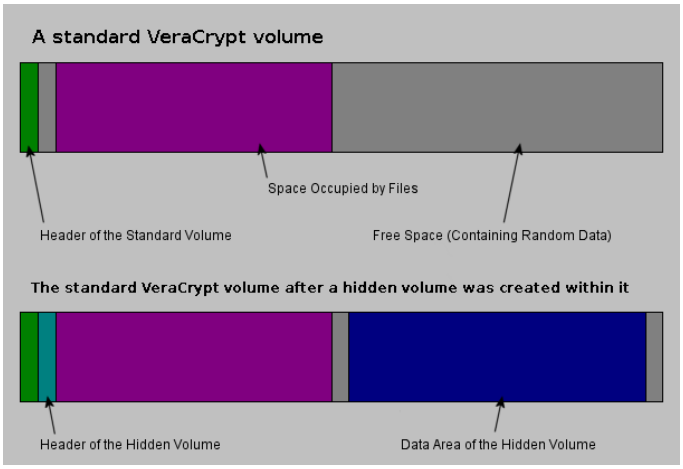

Abbildung 6.2: Standard VeraCrypt-Header vor und nach der Erstellung eines Hidden Volumes (Quelle: `https://www.veracrypt.fr/en/Hidden%20Volume.html`)

Bei Stingray werden gerade diese verräterischen Metadaten, die zum Öffnen des verschlüsselten Blockgerätes dringend notwendig sind, getrennt von den eigentlichen kryptografischen Daten aufbewahrt. Würde der LUKS-Payload einer Gleichverteilungsanalyse unterzogen, so gäbe es vermutlich keinerlei Auffälligkeiten; dazu müsste er ohnehin erst gefunden werden.

Natürlich wird es - wie bei VeraCrypt auch - möglicherweise als verdächtig angesehen, dass eine Verschlüsselungssoftware (cryptsetup) vorhanden ist, allerdings kann man diese ebenfalls zur Öffnung eines unproblematischen Containers nutzen, der seinerseits natürlich kein “Hidden Volume“ oder etwas dergleichen beinhaltet.

Da Stingray sämtliche Metadaten kodiert und steganografisch verarbeitet, lassen sich ohne Kenntnis der Software grundsätzlich keine Hinweise erahnen, die auf einen versteckten Krypto-Container deuten könnten. Hierzu sollte man sich überlegen, wo man die eigentliche Software beim Transit aufbewahrt oder darüber nachdenken, sie vom System zu löschen und erst an einem sicheren Ort wieder aufzuspielen.

Letztendlich könnte lediglich die Tatsache, dass der Swap-Space nicht den gesamten zur Verfügung stehenden Speicherplatz einer Partition einnimmt, Fragen aufwerfen. Dies ließe sich allerdings auch recht elegant mit der nachträglichen Vergrößerung der Partition - wobei das “Swap-Dateisystem“ nicht mit angepasst wurde - begründen und ist ohne Weiteres nicht widerlegbar.

7 Zusammenfassung und Ausblick

Ziel der Arbeit war es, eine Software zu entwickeln, mit der sensible Daten sicher von einem Ort zum anderen transportiert werden können und die Schutzziele der Informationssicherheit unbedingt gewahrt bleiben.

Natürlich lassen sich Daten ganz einfach durch Verschlüsselung über unsichere Kanäle - wie das Internet - von einem Ort zum anderen übertragen, und sich somit die Vertraulichkeit der Daten sichern. Doch es gibt auch Szenarien, wo entweder gar keine Netzanbindung besteht, diese einer harten Zensur unterliegt oder aber nicht die ausreichende Bandbreite zur Verfügung steht. Möglicherweise ist die Anwendung von Verschlüsselung auch verboten bzw. unterliegt Gesetzen zur Überwachung.

Möglicherweise soll auch verhindert werden, dass die zu übermittelnden Daten erst gar nicht den Weg über das Internet nehmen, um eine Gefährdung Datenintegrität ausschließen zu können; denn auch verschlüsselte Daten lassen sich manipulieren, so dass der Inhalt nach der Dechiffrierung verfälscht oder unbrauchbar sein kann.

Für diese Aspekte wurde ein großen Augenmerk auf plausible Abstreitbarkeit gelegt, um bereits die Existenz sensibler und/oder verschlüsselter Daten verschleiern zu können. Denn allgemein gilt: Solange keine Indizien vorliegen, werden vermutlich auch keine entsprechenden Gegenmaßnahmen eingeleitet, bzw. nicht in diese Richtung ermittelt werden. Auch wenn die Verhältnismäßigkeit sich zwischen diversen Ländern, nicht zuletzt wegen der dortigen Regierungsform unterscheidet, können beispielsweise Grenzkontrollen ohne Grund nicht endlos sondern bestenfalls in die Länge gezogen werden. Obgleich Länge ein dehnbarer Begriff ist, hat man sicherlich durch die Verwendung geeigneter steganografischer Verfahren die besseren Karten auf der Hand, was sich sicherlich auch auf die psychische Stabilität positiv auswirken wird und helfen kann, entspannt zu bleiben.

Hinsichtlich der kryptografischen Aspekte wurde auf dm-crypt/LUKS zurückgegriffen, da die von der dazugehörigen Software Cryptsetup angebotenen Chiffren ausgereift und etabliert sind. Zudem ließ sich Cryptsetup sehr gut integrieren und es konnte auch eine gewisse Basiskompatibilität aufrecht erhalten werden. Dies hat zur Folge, dass die von Stingray verarbeiteten Daten als kryptografisch sicher angesehen werden und zur finalen Übergabe nach Abschluss des Transits in einen Standard überführt werden können, für den die steganografische Software nicht mehr benötigt wird. Damit bleibt auch die Tatsache unbekannt, dass die Software Stingray überhaupt Verwendung fand. Hierzu kann grundsätzlich keine pauschale Aussage getroffen werden, ob dies für ein entsprechendes Szenario nötig ist oder nicht.

Bei Evaluierung der Software wurde ein direkter Vergleich mit TrueCrypt/VeraCrypt aufgestellt, wo ein ähnlicher Ansatz der plausiblen Abstreitbarkeit verfolgt wird. Da Stingray eine strikte Teilung von Metadaten und kryptografischen Payload enthält, ist die Chance der Entdeckung deutlich geringer bzw. mit einem größeren Aufwand verbunden. Der Zeitfaktor könnte dabei eine entscheidende Rolle haben die Existenz verschlüsselter Daten zu verschleiern. Der Aspekt der steganografischen Sicherheit sollte dabei nicht als zu unbedeutend angesehen werden, da die Verschlüsselungskomponente beider Verfahren gegen die berühmte "rubber-hose cryptanalysis" jeweils sehr schlechte Karten hat. Der Euphemismus der Gummischlauch-Kryptoanalyse stammt aus dem Usenet und wurde von Marcus J. Ranum wie folgt definiert:
"[...] You still don't get The Master Plan, unless you resort to the rubber-hose technique of cryptanalysis. (in which a rubber hose is applied forcefully and frequently to the soles of the feet until the key to the cryptosystem is discovered, a process that can take a surprisingly short time and is quite computationally inexpensive) [...]" [25]
Unter Gummischlauch-Kryptoanalyse versteht man also sinngemäß und frei übersetzt:
"Wir schlagen solange auf dich ein, bis du uns das Passwort einfach verrätst."

Tabelle 7.1: Gegenüberstellung ausgewählter Lösungen zur Datenverschlüsselung hinsichtlich der Möglichkeit einer plausiblen Abstreitbarkeit

Datenvolumen	Zweck	verwendetes Verfahren	Abstreitbarkeit
Einzeldatei	E-Mail	Public-Key-Kryptografie allgemein	nein
Einzeldatei	lokal	GnuPG/OpenSSL	nein
Einzeldatei	lokal	Dateisteganografiesoftware (z.B. Steghide)	ja
Verzeichnis	lokal	EncFS, eCryptFS	nein
Verzeichnis	Cloud	CryFS, BoxCrypter	nein
Blockgerät	lokal	Cryptsetup/TrueCrypt/VeraCrypt	nein
Blockgerät	lokal	VeraCrypt (Hidden Volume)	bedingt
Blockgerät	lokal	Stingray	ja

Die tabellarische Übersicht zeigt, dass zahlreiche Ansätze für diverse Szenarien denkbar wären. Dabei hat jedes Verfahren sowohl Vor- wie auch Nachteile bzw. eignet sich in erster Linie für bestimmte Zwecke. Gibt es auf Dateiebene viele Möglichkeiten, Verschlüsselung und Steganografie zu kombinieren, so sieht es auf Ebene der Blockgeräte anders aus. Dort erfüllt Stingray hinsichtlich plausibler Abstreitbarkeit definitiv seinen Zweck und muss sich dabei auch keineswegs hinter anderen Lösungen verstecken.

Abschließend kann festgestellt werden, dass das angestrebte Ziel der Arbeit erfüllt wurde, die Software selbst zur Anpassung an diverse Szenarien dabei jedoch noch genug Raum für Veränderungen oder Erweiterungen bietet. Da das Streben nach einem Verbot oder zumindest einer Eindämmung von Verschlüsselung wiederkehrend in den Medien kursiert, werden das Thema Steganografie und die damit verbundenen Möglichkeiten künftig sicherlich nicht an Bedeutung verlieren.

Literaturverzeichnis

[1] S. Channalli, A. Jadhav, “Steganography An Art of Hiding Data“, *International Journal on Computer Science and Engineering Vol. 1*, 2009

[2] J. Cummins, P. Diskin, S. Lau and R. Parlett, “Steganography And Digital Watermarking“, *School of Computer Science, The University of Birmingham*, 2004

[3] D. Kahn, “The Codebreakers - The Story of Secret Writing“, Februar 1973

[4] N. F. Johnson, S. Jajodia, “Exploring Steganography: Seeing the Unseen“, *Computer Journal*, Februar 1998

[5] CompuServe Incorporated, “Graphics Interchange Format, Version 89a“, Juli 1990

[6] N. F. Johnson, S. Jajodia, “Steganalysis of Images Created Using Current Steganography Software“, *Proceedings of the 2nd Information Hiding Workshop*, April 1998

[7] G. K. Wallace, “The JPEG Still Picture Compression Standard“, *IEEE Transactions on Consumer Electronics*, Dezember 1991

[8] H. Berghel, D. Hoelzer, M. Sthultz “Data Hiding Tactics for Windows and Unix File Systems“, *Advances in Computers, Ausgabe 74*, 2008

[9] M. R. Gupta, M. D. Hoeschele, M. K. Rogers, “Hidden Disk Areas: HPA and DCO“, *International Journal of Digital Evidence, Vol. 5*, 2006

[10] B. Esslinger, “The CrypTool Book: Learning and Experiencing Cryptography with CrypTool and SageMath, *CrypTool Project, Vol. 12*, Mai 2018

[11] The Free Software Foundation, “The GNU Privacy Handbook“, August 2000

[12] I. Ristić, “OpenSSL Cookbook“, März 2020

[13] J. Ruusi, “loop-AES.README“, *Sourceforge Repository*, `http://loop-aes.sourceforge.net/loop-AES.README`, Juli 2019

[14] M. Brož, “dm-crypt: Linux kernel device-mapper crypto target“, *GitLab Repository*, `https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt`, Juli 2019

[15] C. Fruhwirth, M. Brož, “Cryptsetup 2.3.0 Release Notes“, https://mirrors.edge.kernel.org/pub/linux/utils/cryptsetup/v2.3/v2.3.0-ReleaseNotes, Februar 2020

[16] M. Brož, “LUKS Linux Unified Key Setup“, *GitLab Repository*, https://gitlab.com/cryptsetup/cryptsetup/-/blob/master/README.md, März 2020

[17] C. Fruhwirth, “LUKS1 On-Disk Format Specification Version 1.2.3“, Januar 2018

[18] M. Brož, “LUKS2 On-Disk Format Specification Version 1.0.0“, Oktober 2018

[19] “’Crocodile hunter’ Steve Irwin killed by a stingray“, *The Guardian*, https://www.theguardian.com/world/2006/sep/04/australia.media, September 2006

[20] Python Software Foundation, “General Python FAQ“, https://docs.python.org/3/faq/general.html, Mai 2020

[21] PNG Development Group, “Portable Network Graphics (PNG) Specification (Second Edition)“, November 2003

[22] D. Bovet, M. Cesati, “Understanding the Linux Kernel, 3rd Edition“, November 2005

[23] IDRIX, “VeraCrypt Documentation“, https://www.veracrypt.fr/en/Plausible%20Deniability.html, März 2020

[24] M. Kedziora, Y. W. Chow, W. Susilo, “Defeating Plausible Deniability of VeraCrypt Hidden Operating Systems“, Juni 2017

[25] M. J. Ranum, “Subject: Re: Cryptography and the Law...“, *Newsgroup: sci.crypt*, Oktober 1990

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 31.07.2020