Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Heiner Perrey

Untersuchung der Vertraulichkeit nach Obfuskierung durch Fountain Codes

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Heiner Perrey

Untersuchung der Vertraulichkeit nach Obfuskierung durch Fountain Codes

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. habil. Dirk Westhoff
Zweitgutachter: Prof. Dr. Ing. Martin Hübner

Abgegeben am 10.08.2010

Heiner Perrey

Thema der Bachelorarbeit

Untersuchung der Vertraulichkeit nach Obfuskierung durch Fountain Codes

Stichworte

Fountain Codes, Erasure Codes, Daten-Obfuskierung, statistische Analyse, Redundanz, Entropie, Kompression, implizite Vertraulichkeit, (Pseudo-) Zufall, Strukturinformation in Dateiformaten, MPEG-2, Java Archive, Bytecode, Broad- und Multicast

Zusammenfassung

Fountain Codes beschreiben eine Kodierungsform, in der die in Pakete unterteilten Daten nach einem bestimmten Schema miteinander verknüpft werden. Außerdem bieten sie gewisse Parameter, um Daten redundant über ein Netzwerk zu verschicken. Hierbei kann auf einen Rückkanal verzichtet werden. In dieser Arbeit wird untersucht, ob sich die Eigenschaften der Fountain-Kodierung nutzen lassen, um mit möglichst wenig zusätzlichen kryptographischen Verfahren Vertraulichkeit implizit zu gewährleisten. Unter diesem Aspekt werden zwei verschiedene Datentypen und ihre Merkmale sowie die Auswirkungen durch die Wahl der Parameter des Kodierungsalgorithmus genauer betrachtet. Ziel ist es, einem potentiellen Angreifer möglichst wenig Ansatzpunkte für eine erfolgreiche statistische Analyse zu bieten. Es werden verschiedene Konfigurationen in einer Testumgebung, welche ein spezielles Angreifermodell simuliert, auf ihre Praxistauglichkeit überprüft.

Abstract

Fountain codes discribe a coding form, in which the data is redundantly linked together into coded packages. They also define a set of parameters for sending these packages through a broadcast channel. This thesis is directed at analysing the obfuscating effect of the fountain coding process as well as pinpointing the extend of the possible resulting implicit privacy. For the analysis two different file formats are checked for characteristic structures and the resulting impact on the obfuscation of structural information enclosed in the data. The parameters of the fountain codes algorithm are varied to analyse their influence on the resulting implicit privacy. The experiments show whether it is more rewarding for a potential attacker to brute-force-attack the system than using a statistical analysis to uncover some structural information. Different configurations of the test environment are used in order to test the practical benefit of the fountain coding for implicit privacy.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Motivation und Problemstellung

In unserer heutigen Welt sind Rechnernetze kaum noch wegzudenken. Der Austausch von Nachrichten zwischen zwei oder mehreren Knoten ist hierbei ein essentieller Bestandteil eines solchen Netzwerkes. Diese Kommunikation dient unter anderem zur Synchronisation von Rechnern eines verteilten Systems und zum Austausch von Daten. Sobald über ein Medium kommuniziert wird, wächst auch das Bedürfnis, die Informationen nicht für jedermann zugänglich zu machen und durch Sicherheitsmaßnahmen bestimmte Schutzziele umzusetzen. Nun bedeutet Sicherheit in der Regel einen Mehraufwand an Rechenleistung und Speicherkapazität. Solange die Geräte eine ausreichende Stromversorgung haben und Verzögerungen akzeptiert werden können, stellt dies für die Funktionalität kaum ein Problem dar.
Doch nicht nur die steigende Verbreitung mobiler Endgeräte mit Anschluss an das Internet, sondern auch die zunehmende Vernetzung alltäglicher Objekte durch Rechner (Ubiquitous Computing), wie zum Beispiel bei Kleidung oder der Nutzung von Sensoren im Gesundheitswesen, machen den Umstieg von Datenkabeln auf einen Funk-Standard und die Nutzung kleinerer Geräte unumgänglich.
So sind Wireless Sensor Networks (WSN) darauf angewiesen, ihre Daten über ein kabelloses Medium zu erhalten. Nun ist die Datenübertragung über eine Funkschnittstelle meistens nicht so zuverlässig wie die Nutzung eines Kabels. Dies führt insbesondere in einer rauschbehafteten Umgebung, in der äußere Einflüsse die gesendeten Signale stark stören, zu enormen Problemen. Viele der Datenpakete gehen verloren und müssen nachgefordert werden. Die Informationen können dann nur sehr langsam gesendet und vielleicht nicht vollständig empfangen werden. Besonders in einem WSN mit einer sehr hohen Anzahl an Knoten bedeutet dies einen erheblichen Mehraufwand in der Kommunikation und kann zur Überlastung des Senders führen [16]. Außerdem ist es möglich, dass ein Knoten sich in einem für einen Menschen nur schwer erreichbaren Ort befindet. Eine Energiezelle so lange wie möglich nutzen zu können ist hier ein entscheidender Vorteil. Auch in großen WSN bedeutet ein häufiger Austausch von Energiezellen einen Mehraufwand, der nach Möglichkeit vermieden werden sollte. Hierdurch könnten auch die Anwendungsgebiete von Sensoren erweitert werden, da eine Wartung nicht mehr in dem gleichen Umfang nötig ist.

Fountain Codes stellen ein Verfahren dar, welches durch geschicktes Kodieren der Daten auf einen Rückkanal verzichten und gleichzeitig die Verluste des Funknetzes kompensieren kann. Da die Daten durch die Kodierung verschleiert werden, soll in dieser Arbeit untersucht werden, inwiefern dies für ein implizites Maß an Vertraulichkeit sorgt. Gelingt dies, müssten nur noch sehr wenige Daten und einige Metadaten des Verfahrens verschlüsselt werden. Hierdurch könnte der Rechenaufwand für die Ver- und Entschlüsselung der meisten Daten eingespart werden, wodurch dieses Verfahren besonders in WSN große Bedeutung gewinnen würde. Peer-to-Peer-Netzwerke, in denen die Daten häufig dezentral bei den einzelnen Endsystemen gespeichert sind, stellen, insbesondere mit Blick auf Ubiquitous Computing, ein spezielles Anwendungsgebiet dar [10]. Da die Robustheit eines solches Netzwerkes auch in der redundanten Datenspeicherung begründet ist, kann ein solcher Ansatz eine sinnvolle Optimierung bieten. Aber auch bei üblichen Computersystemen ist es durchaus sinnvoll, eine performatere Lösung zu finden, um in einer Zeit, in der Energiepreise stetig ansteigen und Umweltschutz eine immer größere Rolle spielt, konkurrenzfähig zu bleiben und mit gutem Vorbild voranzuschreiten.

1.2. Zielsetzung

Fountain Codes nutzen eine Kodierung, in der die Daten redundant in Pakete miteinander verknüpft werden. So können diese Datenpakete zum Bespiel in einem Netzwerk verschickt werden. Hierdurch und durch einen speziellen Sendevorgang kann so auf einen Rückkanal verzichtet und die Verlustrate im Übertragungsmedium kompensiert werden.
Da die Verknüpfung der Daten, zum Beispiel durch die $\oplus$-Operation (s. Abs. 3.2), zu einer Obfuskierung der Daten führt, wird nun der Frage nachgegangen, inwieweit dies eine implizite Umsetzung des Schutzziels „Vertraulichkeit“ darstellt. Da eine Verknüpfung von Klartexten miteinander nicht prinzipiell einem Sicherheitsstandard genügt, sollen bestimmte Bedingungen wie das zugrunde liegende Dateiformat mit untersucht werden. Durch den unterschiedlichen Aufbau der verschiedenen Formate wird angenommen, dass sich einige mehr als andere für eine implizite Vertraulichkeitserweiterung eignen. Die Strukturen der Formate werden hierbei vor und nach dem Kodiervorgang untersucht. Hierfür wird eine Reihe von statistischen Analysen durchgeführt, welche die Struktur, die den Daten zugrunde liegt, sichtbar machen. Diese Anaylse soll die Effekte der Kodierung verdeutlichen und zeigen, bis wann die formatspezifischen Strukturen durch „bloßes Hinschauen“, also durch eine einfache strukturelle Analyse, noch erkennbar sind. Anhand dessen wird schließlich das Vertraulichkeitsniveau bewertet.

1.3. Aufbau der Arbeit

Nach einer Einführung in die im späteren Verlauf genutzten Verfahren und Begriffe im Kapitel „Grundlagen“gibt das Kapitel 3 einen Überblick in das zentrale Thema dieser Arbeit, den „Fountain Codes“. Es sollen Funktionsweise und typische Anwendungsgebiete beschrieben werden. Außerdem wird näher auf den Sicherheitsaspekt eingegangen. Hierbei sollen auch die Nutzung zusätzlicher kryptographischer Verfahren und deren Nachteile vorgestellt werden. Insbesondere wird der obfuskierende Effekt der Fountain Codes erläutert. Schließlich werden in Kapitel 4 die Testumgebung für die Experimente und deren Ergebnisse vorgestellt. In Kapitel 5 werden die Ergebnisse abschließend bewertet, und es wird ein Ausblick auf noch offene und in dieser Arbeit nicht behandelte Forschungsmöglichkeiten gegeben.

2. Grundlagen

In diesem Kapitel werden diejenigen Konzepte und Begriffe eingeführt, die für die Arbeit von zentraler Bedeutung sind. Hierbei wird zuerst eine Einordnung des Begriffs „Informations- und Datensicherheit“ gegeben. Es wird auf kryptographische Verfahren eingegangen, und zwei Angriffstechniken werden aufgezeigt. Des weiteren werden die Dateiformate, die in Kapitel 4 genutzt werden, vorgestellt und Einblicke in ihren internen Aufbau gegeben. Auch der Begriff der Daten-Obfuskierung wird erklärt. Es folgt eine Einführung in das Gebiet der Informationstheorie, in der die Hintergründe der in den Experimenten genutzten Funktionen näher erläutert werden. Da eine Zufallszahlenfolge in den Experimenten als Referenz dient, sind abschließend zwei Arten von Zufallszahlengeneratoren aufgeführt.

2.1. Informations- und Datensicherheit

Ist ein System informations- bzw. datensicher, kann es nicht zu einer unauthorisierten Informationsveränderung oder -gewinnung kommen. Bestimmte Schutzziele konkretisieren die Anforderungen für ein solches System [5, S. 5 ff]. Im Folgenden sollen ein paar wesentliche dieser Schutzziele vorgestellt werden [24, S. 2]:

1. Vertraulichkeit
2. Authentifizierung
3. Verbindlichkeit
4. Integrität

„Vertraulichkeit“ bedeutet die Geheimhaltung der Daten. Sie können also nicht von einem Unbefugten gelesen bzw. korrekt interpretiert werden. „Authentifizierung“ verhindert, dass ein Absender sich als jemand anderes ausgeben kann. Dem Empfänger ist es also möglich, die Identität des Absenders zu überprüfen. Mit der „Verbindlichkeit“ einer Nachricht wird sichergestellt, dass ein Sender später nicht abstreiten kann, dass eine von ihm gesendete Nachricht von ihm stammte. Sie stellt somit eine Art elektronische Unterschrift dar. „Integrität“ erlaubt es dem Empfänger, die Daten auf Korrektheit zu prüfen. Wurden die Daten

zwischen dem Senden und Empfangen verändert, kann der Empfänger dies nachvollziehen [5] [24].

2.1.1. Kryptographische Verfahren

Um die genannten Schutzziele umzusetzen, können verschiedene standardisierte kryptographische Verfahren eingesetzt werden [30] [24]. Zwischen diesen Verfahren gibt es viele Unterschiede in Bezug auf die Funktionsweise, die Anwendungsgebiete und den Sicherheitsstandard. Im weiteren Verlauf dieser Arbeit wird ausschließlich auf das Schutzziel „Vertraulichkeit“ eingegangen. Hierbei wird insbesondere eine implizite Umsetzung ohne weitere kryptographische Mittel untersucht. Da aber auf konventionelle Methoden nicht gänzlich verzichtet werden kann, werden im Folgenden zwei Klassen kryptographischer Verfahren vorgestellt.

Symmetrische Verschlüsselung

Symmetrischen Verfahren liegt ein geheimer Schlüssel zugrunde. Alle Sender und Empfänger müssen diesen Schlüssel besitzen, um die Daten ver- und entschlüsseln zu können. Gelingt es einem Angreifer, den Schlüssel zu bekommen, kann er die Daten ebenfalls entschlüsseln. Ein sicherer Schlüsselaustausch ist also unumgänglich.[1] Die Verschlüsselungsfunktionen können in zwei verschiedene Typen unterteilt werden: die Block- und die Stromchiffrierung. Bei der Blockchiffrierung wird der Klartext in eine feste Blockgröße unterteilt. Die Verschlüsselungsfunktion wird dann jeweils auf die Blöcke angewandt. Zur Steigerung der Sicherheit gibt es hierbei verschiedene Betriebsmodi.[2] Die Stromchiffrierung hingegen nutzt einen potentiell unendlichen Schlüsselstromgenerator, um den Klartext bit- oder byteweise zu chiffrieren [24, Kap. 9] [30].
Ein bekanntes symmetrisches Verfahren, welches eine Stromchiffre nutzt, ist das One-Time-Pad (OTP). Das OTP wurde 1917 von Joseph Mauborgne und Gilbert Vernam entwickelt [21, S. 109]. Es ist ein beweisbar sicheres Verfahren, um Daten mit einem symmetrischen Schlüssel zu chiffrieren. [30, S. 335] [21, S. 109] Soll das One-Time-Pad für digitale Daten genutzt werden, kann hierzu eine einfache XOR-Operation, die auf den Klartext und den Schlüssel ausgeführt wird, genutzt werden [24, S. 19].
Folgende Anforderungen müssen erfüllt sein, um die beweisbare Sicherheit des One-Time-Pads zu garantieren [21, S. 109]:

1. Schlüssellänge entspricht der Länge des Klartextes

[1] Verfahren zum sicheren Schlüsselaustausch werden von Schneier beschrieben [24].
[2] Eine Übersicht hierzu siehe [24, S. 247].

2. Der Schlüssel besteht zu 100% aus Zufallssymbolen
3. Jeder Schlüssel wird nur einmal benutzt

Wenn eine dieser Bedingungen verletzt ist, lässt sich zeigen, dass die Sicherheit nicht mehr garantiert ist [30, S. 41] [24, S. 18ff] [24, S. 233] [21, S. 109ff]. Die Erzeugung eines ausreichend großen Schlüssels, der ausschließlich aus echten Zufallszeichen besteht, und der abhörsichere Austausch des Schlüssels stellten in der Praxis allerdings schon häufig ein Problem dar [21, Kap. 3].

Asymmetrische Verschlüsselung

Bei der asymmetrischen Verschlüsselung handelt es sich um eine Klasse kryptographischer Verfahren, in denen ein öffentlicher und ein privater Schlüssel genutzt werden [30, S. 164ff]. Die Sicherheit beruht auf wohldefinierten mathematischen Problemen. Hierzu zählen insbesondere der diskrete Logarithmus und die Primfaktorzerlegung [30, Kap. 6]. Es ist somit möglich, Daten mit dem öffentlichen Schlüssel einer Person zu verschlüsseln, welche dann nur noch mit dem entsprechenden privaten Schlüssel derselben Person entschlüsselt werden können. Da ein solches Public-Key-Verfahren in der Regel um Größenordnungen langsamer ist als ein symmetrisches Verfahren [24, S. 535], wird es häufig als Hybrid aus symmetrischer und asymmetrischer Verschlüsselung genutzt. Zum Beispiel kann ein Public-Key-Verfahren zum sicheren Austausch eines geheimen symmetrischen Schlüssels genutzt werden [30, Kap. 6] [24, Kap. 19].

2.1.2. Angriffstechniken

Möchte ein Angreifer das Schutzziel „Vertraulichkeit“ eines Systems untergraben, gibt es für ihn neben „Social Engineering“ [5, S. 58] und einer „Seitenkanalattacke“ [12] im Wesentlichen zwei Möglichkeiten, die im Folgenden vorgestellt werden.
Da in dieser Arbeit keine Verschlüsselung, sondern die Kodierung der Daten betrachtet wird, wird hier keine kryptographische, sondern eine strukturelle Analyse der Daten durchgeführt, die zum Teil aber gleiche Untersuchungen erfordert. Wesentlicher Unterschied ist, dass in dieser Arbeit nicht nach einem Schlüssel, sondern nach Strukturen der Daten geforscht wird.

Brute-Force-Angriff

In Anlehnung an die vollständige Schlüsselraumsuche wird hier der Begriff des Brute-Force-Angriffs übernommen. Unter einem Brute-Force-Angriff wird hierbei der Versuch verstanden, die Sicherheit eines Systems durch bloßes Ausprobieren zu knacken. Im weiteren Verlauf wird hierbei nicht der Schlüsselraum, sondern die möglichen Koeffizientenvektoren, die die entscheidenden Informationen zur Dekodierung enthalten, durchsucht.[3]

Statistische Analyse

Eine statistische Analyse interpretiert den Inhalt der Daten auf eine festgelegte Weise und stellt so eine Statistik auf. Hierzu zählen Häufigkeitsanalysen, Berechnung der Entropie und weitere Funktionen zur Suche nach Strukturen oder Regelmäßigkeiten. Hierbei sollen Anhaltspunkte gefunden werden, die es ermöglichen, zusätzliches Wissen über den Klartext, das genutzte Format oder sogar den gesamten Klartext zu erlangen. Anhand dieser zusätzlichen Informationen lassen sich neue, effizientere Angriffe auf das System ausüben. Ist eine statistische Analyse erfolgreich, bietet sie eine wesentlich schnellere Art, um die Sicherheit eines Systems zu knacken, als es mit einem Brute-Force-Angriff in der Regel möglich ist. Konkrete Funktionen für eine statistische Analyse werden in Kapitel 4 beschrieben. Da in dieser Arbeit Strukturen in den Daten gesucht werden, wird auch von einer strukturellen Analyse gesprochen.

2.2. Dateiformate

Dateiformate definieren die Art, wie die in einer Datei gespeicherten Daten zu interpretieren sind. Je nach Format wird ein spezielles Programm benötigt, um die Daten in angemessener Form darzustellen bzw. verarbeiten zu können. Da im weiteren Verlauf der Arbeit auf zwei verschiedene Formate eingegangen wird, sollen diese hier kurz vorgestellt werden. Außerdem sind Strukturen in Dateiformaten ein wesentlicher Bestandteil der Experimente im Kapitel 4, so dass auch eine Einordnung zu diesem Thema folgen soll. Auch der Begriff der Daten-Obfuskierung wird hier vorgestellt.

2.2.1. MPEG-2

Das MPEG-2-Format ist eines der von der „Moving Picture Experts Group“ standardisierten Verfahren zur verlustbehafteten Kompression von Audio- und Videodaten. Wird ein Vi-

[3] Der Koeffizientenvektor ist in Kapitel 3 beschrieben.

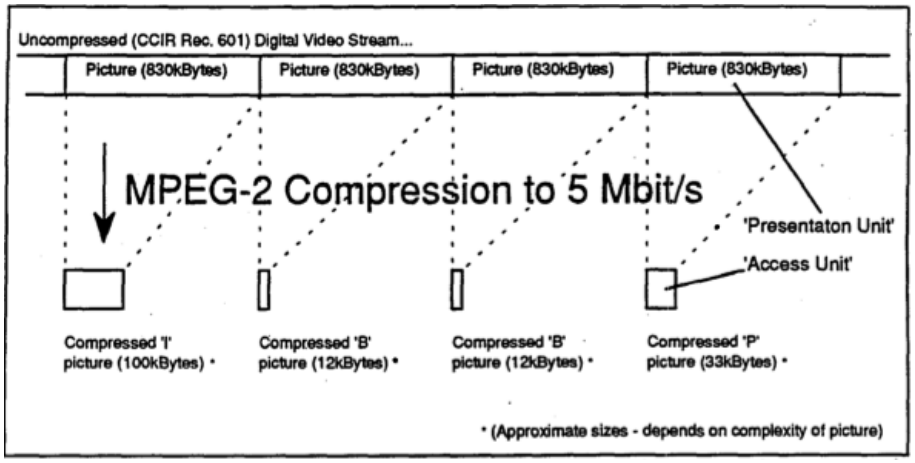

Abbildung 2.1.: Kompression des Video-Materials zur Erstellung einer MPEG-2-Datei (Quelle: [23])

deostrom in MPEG-2 kodiert, werden zunächst die Einzelbilder komprimiert und je nach Komplexität in I-, P- und B-Frames aufgeteilt [23]. Ein I-Frame ist hierbei ein Vollbild (s. Abb. 2.1). P- und B-Frames sind Teilbilder, welche sich auf davor- bzw. dahinterliegende I- oder P-Frames beziehen und Informationen über Änderungen in diesen Bildern enthalten [35]. Ein Video-Elementarstrom ist aus Sequenzen sukzessiv aneinandergereihter Frames aufgebaut [23]. Nun wird aus diesem Strom ein paketisierter Elementarstrom (PES), indem der ursprüngliche Elementarstrom in Pakete unterteilt wird (s. Abb. 2.2). Jedes dieser Pakete wird von einem Startcode eingeleitet [23], welcher eine wesentliche Struktur im MPEG-Format darstellt, auf die später noch eingegangen wird. Aus einer variablen Anzahl an PES kann nun ein Programmstrom zusammengesetzt werden. Dies sind schließlich die Ströme, die klassischerweise als Video auf einer Festplatte gespeichert sind. Solange das MPEG-Format in Medien mit geringen Verlusten eingesetzt wird, bietet sich die Nutzung des Programmstroms an. In stärker verlustbehafteten Medien wird ein Transportstrom genutzt [23]. In dieser Arbeit wird ein Programmstrom näher analysiert, welcher nur einen Video-PES enthält. Auf den Audio-Elementarstrom soll nicht weiter eingegangen werden.

2.2.2. Java Archive (Bytecode)

Bei Java-Bytecode handelt es sich um den kompilierten Java-Quellcode. Dieser kann, sofern es sich um ein ausführbares Programm handelt, von der Java-Virtual-Machine interpretiert und ausgeführt werden [32].
Java Archive (JAR) ist ein Format, welches den Bytecode zusammenfasst und durch zusätzlich enthaltene Metadaten, wie die Angabe der Hauptklasse, die direkte Ausführung des

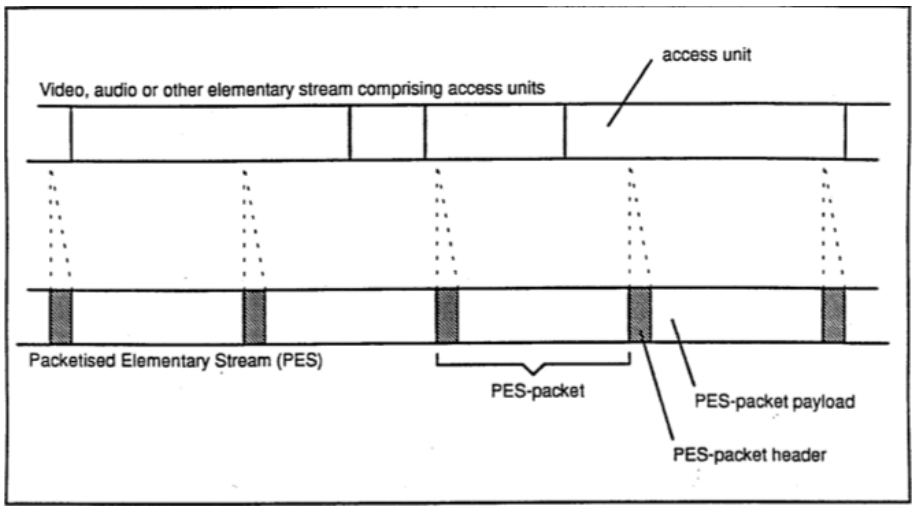

Abbildung 2.2.: Zusammensetzung eines paketisierten Elementarstroms (PES) zur Erstellung einer MPEG-2-Datei (Quelle: [23])

Programms ermöglicht. Die in der JAR enthaltenen Dateien können auch verlustfrei komprimiert werden.
Gibt man den kompilierten Java-Quellcode Zeichen für Zeichen auf der Konsole aus, so beinhaltet dieser viele lesbare Zeichen. Es ist aber auch möglich, den Bytecode direkt in einer für den Menschen lesbaren Form zu erzeugen. Folgende Befehle generieren aus einer Datei mit Java-Quellcode den Java-Bytecode und speichern diesen in einer separaten Datei [34]:

javac *file.java*

javap -c *file* > *file.bc*

Es folgt ein Auszug aus einer so generierten Java-Bytecode-Datei:

```
        (...)
public static void gzip(java.lang.String);
  Code:
    0:   new     #2;
    3:   dup
    4:   aload_0
    5:   invokespecial   #3;
    8:   astore_1
    9:   new     #2;
    12:  dup
    13:  new     #4;
    16:  dup
```

```
17:  invokespecial    #5;
20:  aload_0
21:  (...)
```

Der in dieser Arbeit genutzte Java-Bytecode ist nicht solch ein generierter Bytecode, sondern es werden die *class*-Dateien direkt untersucht.

2.2.3. Strukturen in Dateiformaten

Im Bezug auf Dateiformate werden Strukturen hier als eine interne Ordnung der Daten verstanden. Diese Ordnung ist auf bestimmte Charakteristika des Formats zurückzuführen. Betrachtet man zum Beispiel eine Datei für das Textsatzprogramm TEX, finden sich hier typische Strukturen. Sie enthält Steuerzeichen und Anweisungen, die für eine interne Ordnung sorgen, anhand der TEX das Dokument interpretieren kann. So beginnt jedes Dokument zum Beispiel mit der Anweisung $\backslash begin\{document\}$. Betrachtet man einen Auszug aus einer solchen Datei, ist es möglich, sie durch Merkmale wie zum Beispiel diese Initial-Anweisung dem richtigen Programm zuzuordnen. Auch bei Betrachtung des Quellcodes von Programmiersprachen findet man bestimmte Konstrukte, anhand derer sich der Code der richtigen Sprache zuordnen lässt.
Es gibt viele verschiedene Ausprägungen solcher Merkmale. Schickt man zum Beispiel Daten über ein Netzwerk, bekommt jedes Datenpaket von den Diensten der genutzten Kommunikationsprotokolle entprechende Header-Informationen [14]. Diese enthalten unterschiedliche Informationen, anhand derer eine Identifikation der Protokolle möglich ist. Auch Kodierungsformen tragen ihre eigene Handschrift. Liegt zum Beispiel eine mit ASCII kodierte Textdatei vor, lassen sich Ausprägungen in den Häufigkeiten finden. So sind die Buchstaben des Alphabets in einer Textdatei mit überwiegend lesbarem Text am häufigsten vertreten. Die relative Häufigkeit zueinander ist durch die verwendete Sprache bedingt.
Strukturen gibt es fast überall, wo Informationen gespeichert sind. Es gibt aber auch Daten, die keine ersichtliche Struktur haben und für den Betrachter ungeordnet erscheinen. Erst durch die zugehörige Software bzw. durch zusätzliches Wissen können die Informationen richtig dargestellt und die Ordnung wieder hergestellt werden. So scheint die Ausgabe eines Pseudo-Zufallszahlengenerators zufällig und unstrukturiert, solange der Startwert (Seed) nicht bekannt ist oder die Pseudo-Zufallsfolge sich nicht wiederholt. Aber auch Kodierungs- oder Kompressionsverfahren können Daten produzieren, die ungeordnet und unstrukturiert erscheinen. Hierbei ist geringe Redundanz der Grund dafür, warum die Daten einen relativ unstrukturierten Eindruck erwecken. Betrachtet man ein Bild, in dem alle Farben mit der gleichen Frequenz vorkommen, scheint dieses statistisch gesehen unstrukturiert zu sein. Dies trifft insbesondere auf verlustfrei oder verlustbehaftet komprimierte Bilder zu [22, S. 253]. Diese Annahmen dienen als Grundlage für die Betrachtung der Formatabhängigkeit in den Experimenten in Kapitel 4.

2.2.4. Daten-Obfuskierung

Daten-Obfuskierung bedeutet das Verschleiern oder Verstecken von Informationen. Hierbei wird eine Funktion ausgeführt, die die Daten unkenntlich macht. Diese Funktion kann zum Beispiel aus einem einfachen Bit-Operator oder einer Shift-Operation bestehen. Aber auch komplexere Implementationen sind denkbar.
Hierbei kann das Schutzziel „Vertraulichkeit“ zum gewissen Grad erfüllt werden, da unautorisierter Zugriff teilweise verhindert werden kann [19]. Nicht immer ist hierfür eine Verschlüsselung der Daten angemessen, da eine Daten-Obfuskierung Vorteile in Bezug auf die Performanz haben kann. Ist eine Anwendung also nicht auf ein hohes Vertraulichkeitsniveau angewiesen, kann eine Obfuskierung der Daten hier ausreichend sein.
Bevor Datenpakete über ein Netzwerk geschickt werden, ist es ggf. nötig, diese vorher zu obfuskieren. Hierbei sollen zum Beispiel automatisierte Prozesse, die Inhalte der Datenpakete identifizieren oder Data Mining betreiben, ausgeschlossen werden [19]. Obwohl Daten-Obfuskierung in der Theorie nicht das Vertraulichkeitsniveau einer standardisierten Verschlüsselungsfunktion erreichen kann, soll in der vorliegenden Arbeit untersucht werden, inwieweit in der Praxis durch die Obfuskierung der Daten mit Fountain Codes implizit das Schutzziel „Vertraulichkeit“ erreicht werden kann. Hierbei wird insbesondere Bezug auf das verwendete Dateiformat genommen und untersucht, wie stark die Ergebnisse hiervon beeinflusst werden.

2.3. Informationstheorie

Da viele Ansätze in einer statistischen Analyse auf informationstheoretischen Überlegungen basieren und auch im weiteren Verlauf dieser Arbeit noch näher auf informationstheoretische Grundlagen eingegangen wird, soll hier eine Einführung in die Thematik folgen. Dabei wird nicht im Detail auf dieses sehr komplexe Gebiet eingegangen, sondern es werden nur die nötigen Grundlagen geschaffen.[4] Im Folgenden wird bei der Darstellung von Nachrichten bzw. Informationen von einer binären Darstellungsweise ausgegangen.

2.3.1. Entropie

Der Informationsgehalt einer Nachricht beschreibt, wie viele Bits nötig sind, um alle möglichen Bedeutungen einer Nachricht abzubilden. Angegeben wird der Informationsgehalt anhand der Entropie der Nachricht [24]. Eine Bedeutung kann hierbei zum Beispiel ein Buchstabe in einer Textdatei sein. Besteht eine Nachricht also nur aus den 26 kleingedruckten

[4] Ausführliche Informationen zum Thema „Informationstheorie“ siehe [22] [24] [30].

Buchstaben des Alphabets, so sind hierfür maximal $\log_2 26 = 4.7$ Bits nötig. Die Entropie berücksichtigt aber auch die Wahrscheinlichkeit $\mathcal{P}_i$ bzw. die Häufigkeit, mit der ein Ereignis oder eine Bedeutung einer Nachricht auftreten. Die Entropie, H, einer Nachricht, die $N = \{n_1, n_2, \ldots, n_m\}$ verschiedene Bedeutungen beinhaltet, berechnet sich anhand der Formel [22, S. 50]:

$$H = -\sum_{i=1}^{m} \mathcal{P}_i * \log_2 \mathcal{P}_i \tag{2.1}$$

Je ausgeglichener die Wahrscheinlichkeiten der Ereignisse sind, desto höher ist die Entropie. Sind die Wahrscheinlichkeiten gleich, ist die Entropie maximal. Somit besteht auch eine maximale Unsicherheit, da kein Ereignis wahrscheinlicher ist als ein anderes [30, S. 328].

2.3.2. Redundanz

Unter Redundanz wird das mehrfache Vorhandensein von Informationen bezeichnet. Dies kann zum Beispiel bei Datensicherungen durchaus sinnvoll und gewollt sein. Auch bei Fountain Codes ist Redundanz ein wesentlicher Bestandteil des Kodierverfahrens, um unter anderem auf einen Rückkanal zum Sender verzichten zu können. In Datenbanken kann sich Redundanz jedoch aus Speicherplatz- und Performanzgründen negativ auswirken. Auch in kryptograpischen Systemen kann Redundanz Rückschlüsse auf den Inhalt der Klartextnachricht liefern. Somit ist es hier wünschenwert, dass so wenig Redundanz wie möglich vorhanden ist [24, S. 275]. Redundanz kann sich also je nach Anwendungsfall als nützlich oder hinderlich herausstellen.
Eine Nachricht beeinhaltet Redundanz, wenn sie mit mehr Bits als nötig kodiert wird. Beinhaltet eine Nachricht zum Beispiel die Information *wahr* oder *falsch*, so ist hierfür 1 Bit nötig. Wird sie mit mehr Bits kodiert, enthält die Nachricht Daten, die ohne Verlust an Informationen weggelassen werden können, also redundant sind [24, S. 274].
Berechnet werden kann die Redundanz einer Nachricht aus der Addition ihrer maximal möglichen mit der tatsächlichen Entropie [22, S. 47]:

$$R = \log_2 m + \sum_{i=1}^{m} \mathcal{P}_i * \log_2 \mathcal{P}_i \tag{2.2}$$

2.3.3. Kompression

Das Komprimieren einer Nachricht bedeutet, ihre Länge zu verringern. Hierbei können sowohl verlustfreie als auch verlustbehaftete Verfahren angewandt werden. Verlustfreie Kompressionsverfahren funktionieren, indem sie Redundanz aus der Nachricht entfernen und

somit die Länge der Nachricht bei gleichbleibendem Informationsgehalt verringern. Das Verfahren ist problemlos umkehrbar, und die Nachricht kann so in ihre ursprüngliche Form zurückgewandelt werden. Die Huffmann-Kodierung ist ein Beispiel für ein verlustfreies Kompressionsverfahren [30, S. 333].
Verlustbehaftete Kompressionsverfahren verringern die Länge der Nachricht, indem sie Informationen reduzieren. Hierbei wird in der Regel die Information entfernt, deren Verlust für den Betrachter möglichst unauffällig ist. Wird beispielsweise eine Audiodatei komprimiert, könnten für den Menschen kaum wahrnehmbare Informationen herausgefiltert werden. Somit reduziert sich die Größe der Datei, während die Qualität für den Hörer unwesentlich abnimmt. Je höher hierbei die Kompression, desto deutlicher werden die Verluste.

2.3.4. Forward Error Correction (FEC)

Forward Error Correction ist eine Klasse von Fehlerkorrekturverfahren, bei der redundante bzw. Paritäts-Bits mit an den Empfänger gesendet werden. Hieraus bilden sich Code-Wörter, anhand derer der Empfänger die Daten durch einen Dekodierprozess wiederherstellen und dabei bis zu einem gewissen Grad Fehler korrigieren kann. Solche Fehler treten zum Beispiel durch Paketverlust oder das Kippen von Bits in einem verlustbehafteten Kanal auf [40]. Erasure Codes sind ein Beispiel für die Anwendung von Forward Error Correction. Hierbei werden die Daten unterteilt und in einem Kodierungsprozess zu kodierten Paketen miteinander verknüpft. Die Informationen werden dabei redundant kodiert. Klartextpakete können sich also in mehreren kodierten Paketen befinden. Es werden je nach Gegebenheiten und Bedarf mehr oder weniger kodierte Pakete erzeugt und versendet. So kann ein Sender, der Erasure Codes benutzt, die Informationen an die Empfänger derart kodieren und verschicken, dass auf einen Rückkanal zum Sender verzichtet werden kann. Dies kann sich in einem besonders verlustbehafteten Funknetzwerk oder beim Senden an eine hohe Empfängerzahl als sehr nützlich herausstellen. Mehr zu diesem Thema ist in Kapitel 3 zu finden, in dem auf Rateless Erasure Codes (speziell: Fountain Codes) näher eingegangen wird.

2.4. Generieren von Zufallszahlen

Ein Zufallszahlengenerator erzeugt eine Folge von zufällig verteilten Zahlen. Ein guter Zufallszahlengenerator soll nach Möglichkeit keine verlässliche Vorhersage über die nächste Ausgabe zulassen. Es werden hier zwei Arten von Zufallszahlengeneratoren unterschieden: Diejenigen, die natürliche physikalische Prozesse nutzen, um Zufallszahlen zu erzeugen, und solche, die berechenbare Zufallszahlenfolgen erzeugen, sogenannte Pseudo-Zufallszahlengeneratoren. Wie in Abschnitt 2.1.1 bereits festgestellt wurde, ist die Erzeugung

von guten, also nicht vorhersehbaren Zufallszahlenfolgen für kryptographische Verfahren unumgänglich [24, S. 54].

2.4.1. Natürliche Zufallszahlengeneratoren

Natürliche Prozesse eignen sich sehr gut, um Zufallszahlen zu erzeugen [31] [30, S. 41]. Hierbei wird zum Beispiel der radioaktive Verfall von Cäsium-137-Kernen gemessen, um echte Zufallszahlen zu erzeugen [31]. Eine echte Zufallszahlenfolge ist nicht reproduzierbar und erzeugt somit keine Periode. Nachteilig sind die Geschwindigkeit, in der die Zufallszahlen erzeugt werden, und der Kostenfaktor für die entsprechende Aparatur, der dieses Verfahren für viele Anwendungen nur begrenzt brauchbar macht [15].

2.4.2. Pseudo-Zufallszahlengeneratoren

Pseudo-Zufallszahlengeneratoren erzeugen eine Folge von Zahlen, die zufällig erscheint. Diese Folge ist periodisch, wiederholt sich also nach einer gewissen Zeit [24, S. 53]. Nutzt man den gleichen Startwert (Seed) und die gleichen internen Parameter, gibt der Generator auch die gleiche Folge an Zahlen aus. Die Folgen sind also reproduzierbar. Die Qualität der Pseudo-Zufallszahlengeneratoren lässt sich anhand bestimmter Eigenschaften messen. So erreicht eine große Folge von Zufallszahlen annähernd eine Gleichverteilung aller Zahlen, sofern alle möglichen Zahlen eine gleich hohe Wahrscheinlichkeit haben, gezogen zu werden. Außerdem sollte es nicht zu lange dauern, bis eine Zahl wieder vorkommt. Regelmäßigkeiten dürfen ebenfalls nicht auftreten. Je mehr solche Tests ein Zufallszahlengenerator besteht, desto höher ist seine Qualität [15].
Als Referenz für die in Kapitel 4 durchgeführten Experimente wird eine Folge von Pseudo-Zufallszahlen genutzt. Hierbei liegt die *random()*-Funktion aus der Klasse *java.lang.Math* der Java SE 6 zugrunde.

3. Fountain Codes

In diesem Kapitel wird die Kodierungsmethode Fountain Codes näher erläutert. Zunächst soll eine Einordnung der Thematik erfolgen. Hierzu werden auch Anwendungsgebiete vorgestellt, in denen Fountain Codes sich als besonders nützlich heraustellen. Um die Funktionsweise näher zu beleuchten, wird in Abschnitt 3.2 eine kurze Einführung in die Kodierungs- und Dekodierungsweise gegeben. Da im Rahmen dieser Arbeit die Auswirkungen der Kodierung verschiedener Dateiformate auf die implizite Vertraulichkeit untersucht werden soll, wird im letzten Abschnitt auf die Sicherheit in Fountain Codes eingegangen. Insbesondere sollen hier die Annahmen der obfuskierenden Effekte der Fountain Codes beschrieben werden. Es wird aber auch kurz auf die Nutzung zusätzlicher kryptographischer Verfahren eingegangen. Die konkreten Experimente sind im Kapitel 4 aufgeführt.

3.1. Einführung in Fountain Codes

3.1.1. Einordnung

Fountain Codes sind eine Form der in Abschnitt 2.3.4 beschriebenen Erasure Codes. Sie haben die Eigenschaft, dass ihre Kodierungsrate potentiell unendlich ist. Deshalb kann ein unendlicher Strom (Fontäne) an kodierten Paketen erstellt und über ein Netzwerk an beliebig viele Empfänger gesendet werden [16]. Sie fallen somit unter die Kategorie Rateless Erasure Codes. Die erste Realisierung wurde 2002 von Michael Luby vorgestellt [16]. Im Zusammenhang mit Fountain Codes werden häufig Daten über ein Broad- oder Multicastmedium gesendet. Fountain Codes bieten hierbei eine Kodierungsform, mit der auf einen Rückkanal zum Sender verzichtet werden kann.

3.1.2. Motivation

Ein Rechnernetz verbindet beliebig viele Rechner physikalisch miteinander und ermöglicht den Austausch von Nachrichten untereinander. Da die Kommunikation nicht immer über

einen optimalen physikalischen Kanal laufen kann, gehen hierbei einzelne Teile der oder sogar ganze Nachrichten verloren. Ein Protokoll, um dieses Problem zu lösen, ist zum Beispiel das Transmission Control Protocol (TCP). Hierbei werden diese Pakete von den Empfängern nachgefordert [14].
Besonders in einem Wireless Sensor Network, in dem eine One-To-Many-Kommunikation genutzt wird, führt der Paketverlust zu einem enormen Mehraufwand in der Kommunikation. Alle Empfänger müssen einzeln dem Sender eine Nachricht schicken, in der sie das nicht erhaltene Paket nachfordern. Außerdem bekommen alle Empfänger die Pakete, die von einem anderen nachgefordert wurden, die sie aber gar nicht benötigen [16]. Je stärker die äußeren Bedingungen die Funkverbindung stören, desto stärker wirken sich diese Effekte negativ auf das System aus. Werden zum Beispiel Sensoren in Kleidung vernäht, um lebenswichtige Funktionen zu überwachen, ist eine zuverlässige Kommunikation mit anderen Sensoren unumgänglich [17]. Hierbei befinden sich die Sensoren nicht zwangsläufig in geographischer Nähe. Aber auch wenn auf einem kabellosen Sensor eine neue Software installiert werden soll, müssen alle Knoten die Daten vollständig empfangen, um die Kompatibilität des Netzwerkes zu gewährleisten.
Sind große stationäre Rechner zum Beispiel durch WLAN miteinander verbunden, können ebenfalls solche Probleme auftreten. Hier werden häufig große Datenmengen übertragen, welche durch einen höheren absoluten Satz an Paketverlusten zunehmend die Leistung des Systems beeinträchtigen können.
In solchen Szenarien wäre es wünschenswert, auf einen Rückkanal verzichten zu können, um in der Kommunikation den Mehraufwand zu sparen und für eine höhere Verlässlichkeit zu sorgen. Nun müssen die Empfänger den Paketverlust jedoch auf andere Weise kompensieren. Dies gelingt durch die Kodierung mit Fountain Codes. Hierbei werden Daten kodiert und redundant versendet. Jeder Empfänger kann ein anderes Set an Datenpaketen bekommen. Der Dekodierprozess besteht darin, dass jeder Knoten sich die Informationen aus den Datenpaketen extrahiert, die er benötigt, um die ursprünglichen Daten komplett wiederherzustellen.
Die Nutzung von Fountain Codes ist aber theoretisch nicht nur auf den Einsatz in einem Rechnernetz beschränkt. So ist es nicht ausgeschlossen, die Daten zum Beispiel vor dem Speichern auf einer Festplatte mit Fountain Codes zu kodieren. Auf diese Weise können fehlerhafte Blöcke kompensiert werden und es kann gleichzeitig, falls möglich, ein implizites Maß an Vertraulichkeit umgesetzt werden. Fountain Codes sind somit auch für die Kommunikation zwischen dem Prozessor eines Rechners und anderer Hardware denkbar.

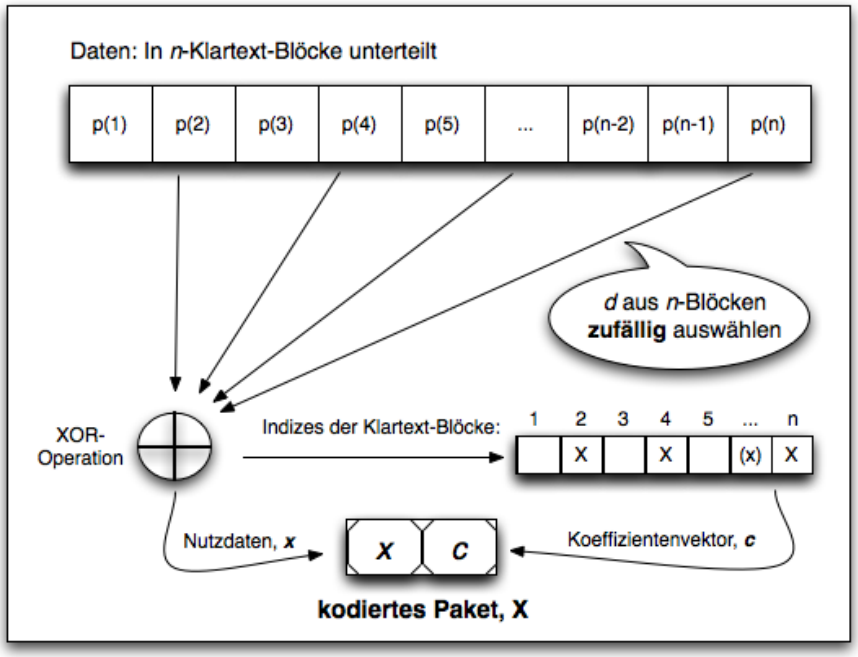

Abbildung 3.1.: Kodierung – Erstellung eines kodierten Paketes

3.2. Funktionsweise

Wie in Abschnitt 2.3.4 dargestellt, basieren Fountain Codes auf einem Kodier- und Dekodierprozess. Zunächst wird eine Möglichkeit zur Kodierung und Dekodierung der Daten vorgestellt. Es folgt die Erklärung zur Degree Distribution und ihre Bedeutung.

3.2.1. Kodierung

Im Wesentlichen muss der Encoder eine bestimmte Menge von kodierten Paketen X erzeugen. Diese Pakete bestehen aus den kodierten Nutzdaten x und den Metadaten (Koeffizientenvektor) c. Die Erstellung wird in Abbildung 3.1 veranschaulicht. Um die Nutzdaten x für ein kodiertes Paket zu erstellen, wird zunächst die unkodierte Datei in n gleichgroße Klartext-Blöcke p unterteilt. Die Größe der Datei muss also vor Beginn des Kodierprozesses bekannt sein. Dann wird nach einer Funktion, Ω (Degree Distribution), pro x ein Degree d ausgewählt. Das Degree bestimmt die Anzahl an Klartext-Blöcken p, die für ein konkretes kodiertes Paket miteinander verknüpft werden. Nun müssen nur noch d aus den n Blöcken zufällig ausgewählt und sukzessive miteinander verknüpft werden. Als Verknüpfungsoperator wird zum Beispiel das „exklusive oder“ bzw. $\oplus$-Operation genutzt (s. Formel 3.1). Die

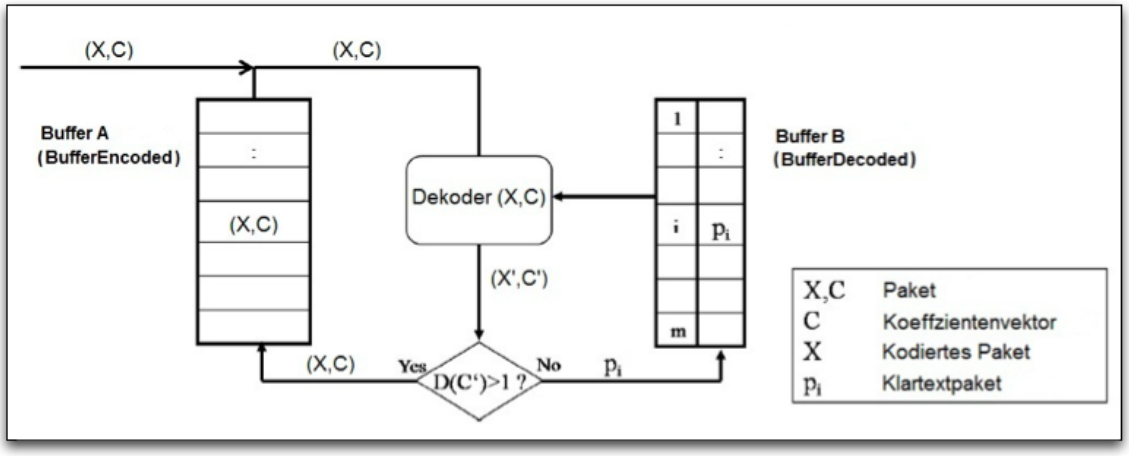

Abbildung 3.2.: Dekodierung – Nutzung des Puffers A und B für den Dekodierprozess (Quelle: [1])

Nutzdaten, x_i, eines kodierten Pakets werden somit nach folgender Formel erstellt:

$$x_i = \oplus_{n=1}^{d} p_{(random)} \tag{3.1}$$

Damit später nachvollziehbar ist, aus welchen Klartexten x besteht, wird für jedes Symbol der Koeffizientenvektor c erstellt, in dem gespeichert ist, welche konkreten Klartextpakete x beinhaltet. Pro Klartextblock, der in x verknüpft ist, wird an dem Index von c ein Flag auf $wahr$ gesetzt. Es können potentiell unendlich viele solcher kodierten Pakete erstellt werden.

3.2.2. Dekodierung

Der Dekodierprozess erfolgt wie in Abbildung 3.2 dargestellt. Nicht jeder Knoten muss das gleiche Set an kodierten Paketen erhalten. Um die ursprünglichen Daten wiederherzustellen, nimmt der Dekoder die empfangenen kodierten Pakete entgegen und überprüft den jeweiligen Koeffizientenvektor. Hat in c nur ein Index den Wert $wahr$, liegt ein Klartextpaket vor, welches nicht weiter dekodiert werden muss. Dieses wird in dem Puffer B gespeichert. Sobald ein X in Puffer B gespeichert wird, werden alle kodierten Pakete aus Puffer A wieder am Dekodierprozess beteiligt. Auch die schon dekodierten Pakete werden weiterhin am Dekodierprozess beteiligt. Bei jedem X_i mit einem $d > 1$, welches also mehrere Klartextpakete beinhaltet, wird geprüft, ob im Puffer B ein Klartextpaket $X_j$enthalten ist, welches auch in X_i vorliegt. Sollte dies nicht der Fall sein, wird X_i im Puffer A gespeichert. Sonst werden die Pakete X_i und X_j miteinander verknüpft. Es ergibt sich X_{result}, das alle Pakete aus X_i

beinhaltet, bis auf das Klartextpaket, welches in X_j enthalten war. Der Koeffizientenvektor von X_{result} muss entsprechend angepasst werden. Hat c von X_{result} nun einen Degree von > 1, so wird es in Puffer A gespeichert. Ist sein Degree genau 1, so wird es in Puffer B gespeichert.[1]

3.2.3. Degree Distribution

Eine Degree Distribution Ω stellt eine Funktion für den Encoder dar, aus der dieser für jedes kodierte Paket das konkrete Degree d ermittelt. Da das Degree für den Dekodierprozess eine zentrale Bedeutung hat [16], ist auch die Verteilung aller Degrees und somit die Degree Distribution entscheidend. Verschiedene Distributions haben unterschiedliche Verteilungsmerkmale, welche für die Funktionsweise des jeweiligen Kodier- und Dekodiervorgangs wichtig sind. In dieser Arbeit wird mit einem festen Degree, also keiner Degree-Verteilung, gearbeitet.[2] Diese Vereinfachung dient der Findung eines Schwellenwertes, um zu untersuchen, bis zu welchem Degree die kodierten Daten noch messbare Strukturen aufweisen.

3.3. Anwendung in der Praxis

Erasure Codes sind in der Praxis zunehmend anzutreffen. Das Unternehmen „Digital Fountain, Inc.“ beschäftigt sich mit der Entwicklung von Software, die unter anderem auf Fountain Codes basiert [3].

3.3.1. LT-Codes

LT-Codes, die von Michael Luby entwickelt wurden, sind die erste Realisierung von Rateless Erasure Codes und somit auch von Fountain Codes [16]. Die Effizienz von LT-Codes ist fast optimal, denn der Dekoder benötigt nur wenig mehr Pakete als die Anzahl der Blöcke, in die die Datei unterteilt wurde (siehe 3.2). Von Bedeutung sind vor allem die Raptor Codes, welche eine Weiterentwicklung der LT-Codes darstellen [3].

[1] Zu einem alternativen Dekodierprozess siehe [7].
[2] Zu den Degree Distributions siehe [16].

3.3.2. Raptor Codes

Raptor Codes sind eine Erweiterung von LT-Codes. Sie wurden 2006 von Amin Shokrollahi vorgestellt [26]. Mit ihnen ist es möglich, die Daten in linearer Zeit zu kodieren und zu dekodieren. Auch hier ist das Ziel, auf einen Rückkanal zu verzichten. Das Unternehmen „Digital Fountain, Inc." stellt viele Software-Lösungen vor, die auf Raptor Codes basieren.[3]

3.3.3. Projekt: Fountain Codes

Im Sommersemester 2010 fand an der Hochschule für Angewandte Wissenschaften Hamburg ein Projekt zum Thema „Fountain Codes" statt. Es ging dabei um die Frage, unter welchen Bedingungen die Kodierung mittels Fountain Codes sich als besonders vorteilhaft herausstellt. Es wurde zuerst eine technische Schnittstelle zur Übertragung der Daten über ein Rechnernetzwerk entworfen. Vor dem Versenden wurden die Daten mittels Fountain Codes kodiert. Der Empfänger musste die Daten wieder entsprechend dekodieren. Außerdem wurde ein Sicherheitskonzept vorgesehen, welches Datenintegrität und weitere Sicherheitsaspekte abdeckte.[4]

3.4. Sicherheitskonzepte für Fountain Codes

Wie in Abschnitt 3.3 schon angemerkt, werden Fountain Codes in der Praxis bereits eingesetzt. Um auch sensible, mit Fountain Codes kodierte Daten austauschen zu können, gibt es verschiedene Ansätze, die sich mit der Sicherheit dieses Verfahrens beschäftigen [8] [7]. Hier werden das für diese Arbeit angenommene Angreifermodell sowie zwei Ansätze für eine Vertraulichkeitserweiterung für Fountain Codes vorgestellt. Zum einen wird hierbei eine Umsetzung einer impliziten Vertraulichkeit untersucht. Zum anderen sollen auch konventionelle Verfahren aufgezeigt werden. So gibt es viele standardisierte Techniken, um Sicherheit in einem Rechnernetz zu gewährleisten [30] [24]. Diese bedeuten aber immer einen Mehraufwand in Bezug auf die Rechenleistung [2]. Da in dem impliziten Ansatz nur der Koeffizientenvektor und ggf. kodierte Pakete mit einem geringen Degree verschlüsselt werden müssen, kann hierbei ein großer Teil des Aufwandes gespart werden. Somit erreicht das System eine bessere Performanz und wird effizienter. Besonders in Systemen mit energiebeschränkter Geräteplattform bringt dies große Vorteile.

[3] Zu den Raptor Codes siehe [3] [26].

[4] Das Spezifikationsdokument der Übertragungsschicht befindet sich im Anhang.

3.4.1. Sicherheit durch zusätzliche kryptographische Verfahren

Ein möglicher Ansatz besteht darin, die Nutzdaten und ggf. den Koeffizientenvektor mit einem symmetrischen Verfahren zu verschlüsseln. Der geheime Schlüssel könnte über ein Public-Key-Verfahren getauscht werden. Vorteilhaft ist hierbei, dass ohne Analyse- und Forschungsaufwand ein Sicherheitsmechanismus benutzt werden kann, der nach dem heutigen Stand der Technik als sicher gilt [24] [30]. In dem in Abschnitt 3.3.3 beschriebenen Projekt wird ein Sicherheitskonzept vorgestellt, welches auf schon existierenden Ansätzen beruht. Hierbei wird das Schutzziel „Integrität“ umgesetzt.
Nachteilig wirkt sich allerdings der Rechenaufwand aus, der für eine separate Verschlüsselung nötig ist [2] [25].

3.4.2. Implizite Sicherheit durch Fountain Codes

Wie in Abschnitt 3.2 beschrieben, können Fountain Codes zur Kodierung die $\oplus$-Operation nutzen. Die $\oplus$-Operation ist häufig Teil von kryptographischen Verfahren. Betrachtet man die Ausgabe dieses Bit-Operators, so sind zwei der vier möglichen Ergebnisse 1, und die restlichen zwei sind 0. Die folgende Auflistung zeigt die vollständige Ausgabe:

$$0 \oplus 0 = 0$$
$$1 \oplus 0 = 1$$
$$0 \oplus 1 = 1$$
$$1 \oplus 1 = 0$$

Basiert ein kryptographisches Verfahren allerdings einzig auf der $\oplus$-Operation, kann es nur unter bestimmten Voraussetzungen ein hohes Sicherheitsniveau halten [24, S.16]. Sind die drei Bedingungen (s. Abs. 2.1.1) des One-Time-Pad erfüllt, kann mit einer einfachen $\oplus$-Operation eine informationstheoretisch beweisbare Sicherheit erreicht werden. Bei Fountain Codes handelt es sich jedoch nicht um eine Verschlüsselungs-, sondern um eine Kodierungsform. Somit haben die Daten, die miteinander verknüpft werden, zwar die gleiche Größe, es handelt sich aber nicht um ein Schlüssel-Klartext-Paar, sondern um ein Klartext-Klartext-Paar. Diese erscheinen im Gegensatz zu einem kryptographischen Schlüssel in der Regel nicht als zufällige Zahlenfolge. Außerdem sollen die gleichen Klartextblöcke mehr als einmal genutzt werden, um die nötige Redundanz zu erzeugen. Eine explizite Sicherheit durch den Kodierungsalgorithmus ist also nicht gegeben.
Wie schon in Abschnitt 2.2 beschrieben, gibt es in verschiedenen Datenformaten spezifische Strukturen, zum Teil aber auch unstrukturierte Daten. Gelingt es durch die Fountain-Kodierung alle Strukturen durch unstrukturierte Daten zu überlagern, sollte die Ausgabe des

Kodiervorgangs immer mehr der einer (Pseudo-)Zufallszahlenfolge ähneln. Je nach Verhältnis von unstrukturierten zu strukturierten Daten wäre es somit möglich, ein gewisses implizites Vertraulichkeitsniveau zu erreichen. Dies kann in einer praktischen Umsetzung natürlich nur gewährleistet sein, wenn der Koeffizientenvektor c und ggf. kodierte Pakete mit einem nicht ausreichend hohen Degree durch ein zusätzliches kryptographisches Verfahren gesichert sind. Es muss also untersucht werden, ab welchem Degree und unter welchen Bedingungen ein kodiertes Paket als sicher angenommen werden kann. Eine weitere Bedingung kann zum Beispiel das Dateiformat darstellen.
Bei der Verknüpfung des Kodiervorgangs werden strukturierte Daten durch unstrukturierte überlagert. Unstrukturierte Informationen sind somit dominant gegenüber den strukturierten [7]. Bei der Kodierung werden zufällig von 1 bis d aus n Blöcken ausgesucht und zu kodierten Paketen verknüpft. Je mehr unstrukturierte Informationen vorhanden sind, deren Eigenschaften denen von Zufallszahlenfolgen ähneln, desto höher ist die Wahrscheinlichkeit, dass in jedem X alle strukturierte Bits durch je ein unstrukturiertes Bit überlagert wurden. Hierbei spielt die Wahl des Degrees und somit auch der Degree Distribution eine wesentliche Rolle. Denn je mehr Blöcke miteinander verknüpft werden, desto höher ist die Wahrscheinlichkeit, dass alle Strukturen überlagert werden. Im Erfolgsfall erhielte man eine Fontäne an kodierten Paketen, in denen keine Strukturen oder Regelmäßigkeiten zu finden sind. Im Idealfall sind keine Unterschiede zwischen dieser Fontäne und einem gleich großen Zufallszahlenstrom zu erkennen.
Um zu analysieren, wie hoch das Vertraulichkeitsniveau der Fountain Codes ist, soll ein weiterer Punkt in die Argumentation aufgenommen werden: Ist ein Angreifer im Besitz eines kodierten Paketes mit $d = 2$ und kann er dieses auch als solches erkennen, liegt im Prinzip eine Situation wie bei dem Resultat eines WEP-Angriffs vor, bei dem der Angreifer den Schlüssel durch zwei Pakete herausgekürzt hat [7]:

$$(p_1 \oplus k) \oplus (p_2 \oplus k) = p_1 \oplus p_2$$

Da der Angreifer weiß, dass er zwei Klartextpakete hat, ist an dieser Stelle die WEP-Attacke fast erfolgreich. Soll also die implizite Vertraulichkeit von Fountain Codes weiterführend untersucht werden, muss gezeigt werden, dass es für einen Angreifer nur sehr schwierig bis unmöglich ist, ein $d = 2$ Paket als solches zu enttarnen. Auch muss es schwierig sein, durch eine statistische Analyse Strukturen zu erkennen. Die Simulationen, die diese Bedingungen überprüfen, werden in Kapitel 4 dargestellt.

3.4.3. Angreifermodell

Damit die Ergebnisse der Experimente nachvollzogen werden können, soll hier das Angreifermodell, welches den Untersuchungen zugrunde liegt, vorgestellt werden. Grundsätzlich

werden zwei Arten eines Angriffs unterschieden, der aktive und der passive. Ein aktiver Angriff bedeutet, dass der Angreifer lebhaft in den Informationsaustausch eingreift, während ein passiver Angreifer nur den Informationsfluss mithören kann. In dieser Arbeit wird nur von einem passiven Angreifer ausgegangen, welcher sich an jeder beliebigen Position zwischen Sender und Empfänger befinden kann. Er liest sämtliche Pakete mit und lauscht so unter optimalen Bedingungen. Es besteht für ihn die Möglichkeit, das System durch einen Brute-Force-Angriff anzugreifen oder zu versuchen, an zusätzliche Informationen über die Daten zu gelangen [7]. Ziel ist es zu zeigen, dass es für den Angreifer lohnender ist, das System durch einen Brute-Force-Angriff zu untergraben, als eine statistische Analyse durchzuführen, um so an entscheidende Informationen zu gelangen. Hierbei wird angenommen, dass ein Brute-Force-Angriff genau dann erfolgversprechender ist, wenn durch eine statistische Analyse keine sichtbaren Strukturen gefunden werden können und der in diesem Abschnitt beschriebene Angriff durch Isolation eines $d = 2$ Pakets, ebenfalls keine Aussicht auf Erfolg birgt.
Unter der Annahme, dass die Kodierung mit Fountain Codes für ein hohes implizites Vertraulichkeitsniveau sorgt, müsste ein Angreifer daher alle möglichen Pakete miteinander verknüpfen, um an den Klartext zu gelangen. Da hier kein Schlüssel zugrunde liegt, ist ein solcher Angriff bei weitem nicht so aufwendig wie der auf ein konventionelles kryptographisches System.[5] [6] Es soll in dieser Arbeit gezeigt werden, dass es einem Angreifer nicht noch leichter fällt, das System durch eine strukturelle bzw. statistische Analyse zu untergraben. Hierbei wird deutlich, dass, selbst wenn eine statistische Analyse nicht erfolgreich ist, das System durch einen Brute-Force-Angriff noch immer relativ leicht zu knacken ist. Es kann also nicht die gleiche Vertraulichkeit erreicht werden, wie es mit einem standardisierten kryptographischen Verfahren möglich ist.

[5] Aufwand eines Brute-Force-Angriff auf ein krpytographisches System mit der Schlüssellänge k: 2^k.
[6] Zum Aufwand eines Brute-Force-Angriffs auf die implizite Vertraulichkeit in Fountain Codes siehe [7].

4. Untersuchung der Obfuskierung durch Fountain Codes

In diesem Kapitel werden die durchgeführten Experimente vorgestellt und ausgewertet. Zunächst geht es darum, die dieser Arbeit zugrunde liegende Hypothese und die entsprechenden Ziele vorzustellen. Danach wird die für die Experimente genutzte Umgebung vorgestellt. In Kapitel 5 folgt eine abschließende Bewertung über die hier aufgezeigten Ergebnisse.

4.1. Hypothese und Zielsetzung

Ein klassisches kryptographisches Verfahren kann, solange eine Kryptoanalyse nicht erfolgreich ist, nur durch einen Brute-Force-Angriff geknackt werden. Analog wird davon ausgegangen, dass, solange geringe Degrees und der Koeffizientenvektor zusätzlich verschlüsselt sind, die Fountain-kodierten Daten nicht in ihre ursprüngliche Klartextform konvertiert werden können. Ziel der Simulation ist es zu zeigen, dass durch eine einfache strukturelle Analyse keine zusätzlichen Informationen durch den Angreifer extrahierbar sind. Somit wäre ein Brute-Force-Angriff erfolgversprechender als der Versuch, mit einer statistischen Analyse an zugrunde liegende Strukturen zu gelangen. Hierfür sollen die Dateien vor und nach der Kodierung auf Strukturen und Regelmäßigkeiten untersucht werden. Dabei wird davon ausgegangen, dass sich strukturierte Daten (s-Bits) durch unstrukturierte (u-Bits) überlagern lassen. So ist es nötig, die kodierten Pakete jeweils mit einem geschickt gewählten Degree zu kodieren, so dass schließlich alle Strukturen wegretuschiert werden. Erst durch den Fountain-Dekodierprozess ließen sich so die Daten und ihre Strukturen wiederherstellen. Hierbei muss jedes Struktur-Bit mit einem unstrukturierten Bit verknüpft werden [7]:

$$\begin{aligned} p_1 &= ssuuuuuu \\ \underline{\oplus p_2 &= ssuussuu} \\ x &= ssuuuuuu \end{aligned}$$

Im Beispiel werden zwei der vorhandenen s-Bits überlagert. Daraus resultieren u-Bits. Zwei s-Bits konnten nicht überlagert werden und müssten durch die Verknüpfung mit einem weiteren Paket verschleiert werden. Gelingt es zu zeigen, dass die Strukturen ausreichend durch

die Kodierung obfuskiert werden können, spricht dies dafür, dass eine statistische Analyse für einen Angreifer keine nützlichen Informationen birgt.
Je näher hierbei die Ausgabe der Kodierfunktion der eines Zufallszahlengenerators kommt, desto schwieriger wird es für den Angreifer, eine Analyse erfolgreich durchzuführen. Sind die Ausgaben gleichwertig bzw. nur äußerst schwer zu unterscheiden, ist es für den Angreifer wahrscheinlich lohnender, einen Brute-Force-Angriff durchzuführen.
Aus diesem Grund dient eine Datei aus Pseudo-Zufallszahlen als Bezugspunkt für einen Vergleich mit den anderen Testdateien. Im Rahmen der Experimente wird hierbei auf echten Zufall verzichtet und ein Pseudo-Zufallszahlengenerator genutzt, um die Referenzdateien zu erzeugen. Dies birgt zwar Nachteile, wie in Kapitel 2 festgestellt wurde, kann aber im Rahmen der hier vorgestellten Analyse toleriert werden.
Da verschiedene Datenformate unterschiedliche strukturelle Informationen enthalten, soll untersucht werden, welche Formate sich in welchem Umfang dafür eignen, ein angemessenes implizites Sicherheitsniveau zu erreichen.

4.2. Die Testumgebung

Die Testumgebung besteht aus verschiedenen Funktionen, die für eine statistische Analyse der Testdateien dienen. Hierbei geht es darum, die einem Angreifer gebotene Umgebung zu simulieren, um somit Aufschluss über die potentielle Gefahr einer Attacke abwägen zu können. Insbesondere soll eine Aussage darüber getroffen werden, ob sich die Obfuskierung der Daten durch die Fountain-Kodierung nutzen lässt, um implizit das Schutzziel „Vertraulichkeit“ umzusetzen. Die folgenden Funktionen werden in den Experimenten genutzt, um zu zeigen, wie schwer es für einen Angreifer ist, durch eine statistische Analyse an Informationen über den Klartext zu gelangen.
Im Rahmen der Experimente werden sowohl komplette Datenströme als auch beispielhaft einzelne kodierte Pakete betrachtet. Insbesondere die Unterscheidbarkeit der Degrees und das in Abschnitt 3.4 angesprochene Angriffsszenario werden näher untersucht.

4.2.1. Das Testprogramm

Das Testprogramm besteht aus verschiedenen Funktionen, die für unterschiedliche Testverfahren stehen, deren Funktionsweise im Folgenden erläutert wird.[1] Als Input-Parameter werden Testdateien übergeben, welche auf Strukturen untersucht werden sollen. Die Datei wird hier jeweils byteweise eingelesen und interpretiert, wobei jedes Byte acht Bit umfasst. Es

[1] Der Quellcode der Funktionen befindet sich im Anhang.

wird also davon ausgegangen, dass es genau $2^8 = 256$ verschiedene interpretierbare Zustände gibt. Im weiteren Verlauf wird ein Zustand auch als $Zeichen$ oder $Byte$ (-$Zustand$) referenziert. Dies steht jeweils für eine bestimmte Anordnung von 8-Bit. Der angegebene absolute statistische Fehler, ΔN_{abs}, für jeden Datenpunkt wird anhand folgender Formel bestimmt [27]:

$$\Delta N_{abs} = \sqrt{N} \tag{4.1}$$

wobei N dem gemessenen Wert entspricht. Da geprüft werden soll, ob eine kodierte Datei von einer Zufallszahlenfolge unterscheidbar ist, wird für die Testdateien dieselbe Formel zur Bestimmung des Fehlers genutzt.

Verhältnis aus Einsen und Nullen

Als erstes werden die Einsen und Nullen in der jeweiligen Testdatei gezählt. Hierzu wird jedes eingelesene Byte separat untersucht. Jedes Bit wird geprüft und für den entsprechenden Zustand der Zähler für Eins oder Null um Eins erhöht. Ist das Ende der Datei erreicht, steht in dem Zähler für Eins, wie viele Bits den Zustand Eins haben, und in dem Zähler für Null das entsprechende Ergebnis für den Zustand Null. Als Rückgabewert folgt der relative Anteil der Nullen an der Gesamtmenge. Die Anzahl der Einsen leitet sich somit aus $100 - N^0_{\%}$ ab.

Häufigkeitsverteilung

Es soll untersucht werden, wie die einzelnen Bytes der Datei auf einen 8-Bit-Wertebereich verteilt sind. Hierzu werden $N = 256$ Zähler genutzt. Jedes eingelesene Byte, b, wird als Integerwert zwischen 0 und 255 interpretiert. Der Zähler, der dem Integerwert entspricht, wird pro Vorkommen des konkreten Byte-Zustandes um 1 erhöht. Sobald das Ende der Datei erreicht ist, steht in jedem Zähler, wie oft dieses Byte in der Datei vorkommt. Die Häufigkeit, h, eines konkreten Byte-Zustandes, b_i, berechnet sich durch Summierung seiner Vorkommnisse über alle Zeichen, N_{char}, der Datei:

$$h = \sum_{i=1}^{N_{char}} b_i \tag{4.2}$$

Da die Summe aller Häufigkeiten der Größe der Datei (S_{file}) entspricht, berechnet sich die mittlere Häufigkeit anhand der Formel:

$$\overline{h} = \frac{1}{N} * S_{file} \tag{4.3}$$

Um ein Maß für die Verteilung der einzelnen Werte zu bekommen, wird die Standardabweichung, σ, vom Mittelwert berechnet. Dies geschieht anhand der Formel:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\overline{h} - h_i)^2} \tag{4.4}$$

Entropie

In dieser Funktion wird anhand der Formel 2.1 die Entropie der Daten berechnet. Hierfür wird die Häufigkeitsverteilung genutzt, anhand der die relative Häufigkeit errechnet wird, welche schließlich als Wahrscheinlichkeit in die Formel einfließt.

Abstandstest

Ähnlich wie beim „Gap-Test“ [15] soll geprüft werden, wie lange es dauert, bis ein bestimmtes Byte, b_x, noch einmal auftritt. Hierbei wird nicht die Zeit, sondern es wird die Anzahl anderer Bytes, b_y, die nicht dem gesuchten entsprechen, gemessen.
Hierfür stehen zwei Funktionen zur Verfügung. Die erste Funktion untersucht für jedes b_x sämtliche Abstände f und speichert diese. Es ergibt sich ein Datenfeld, in dem jeder Abstand zum nächsten Auftreten des Byte-Zustandes gespeichert ist. Für jedes b_x kann schließlich der durchschnittliche Abstand, f_X, aus der Summe aller Abstände, N, berechnet werden:

$$\overline{f}_X = \frac{1}{N}\sum_{i=1}^{N} f_i \tag{4.5}$$

Die zweite Funktion liefert als Ergebnis ein Datenfeld, in dem die Abstände klassifiziert vorliegen. Hierbei steht der Index des Datenfeldes für die Größe des Abstandes und der Wert am jeweiligen Index für die absolute Häufigkeit des jeweiligen Abstandes.

n-Tupel-Suche

Um nicht nur nach Strukturen einzelner Byte-Zustände zu suchen, soll die Datei auch auf Verkettungen gleicher Byte-Zustände und ihrer Folgezustände analysiert werden.
Ziel dieses Tests ist es, alle Bytes aufzuspüren, die n-mal direkt hintereinander vorkommen. Darüber hinaus soll jeweils ein Folgezeichen nach jedem n-Tupel analysiert werden. Für $n = 3$ werden also Zeichenketten gefunden, die wie folgt aufgebaut sind: *0x 61 61 61 70* oder *0x 48 48 48 50* etc.. Mit $n = 2$ kann so auch das Header-Präfix in einem MPEG-2-PES (*0x 00 00 01*) ermittelt werden.

n=1	A	B	C	D	E
A	2	2		1	
B			1		
C					1
D					1
E	2				

Tabelle 4.1.: Beispiel-Matrix der n-Tupel-Suche für $n = 1$

n=2	A	B	C	D	E
A		2			
B					
C					
D					
E					

Tabelle 4.2.: Beispiel-Matrix der n-Tupel-Suche für $n = 2$

Der Wert für n ist hierbei pro Funktionsdurchlauf konstant und wird als Eingabeparameter übergeben. Es werden also alle Ketten gesucht, in denen der Zustand eines Bytes n-mal in Folge vorkommt, und es wird geprüft, welches Byte jeweils auf diese Kette folgt. Da ein Byte 256 Zustände annehmen kann, gibt es auch pro Funktionsdurchlauf maximal 256 verschiedene mögliche Zeichen, die ein n-Tupel bilden können. Als Folgezeichen auf ein n-Tupel gibt es ebenfalls höchstens 256 verschiedene Bytes, die auftreten können.
Um die Quantität der n-Tupel samt Folgezeichen zu bestimmen, wird eine Matrix aus 256×256 Zählern definiert. Jeder Zähler steht für ein Folge Zeichen auf ein n-Tupel. Kommt ein Zeichen n-mal in Folge vor, wird der entsprechende Zähler des Folgezeichens um eins erhöht.

Beispiel: Betrachtet wird ein Alphabet aus den Zeichen $\{A, B, C, D, E\}$. Liegt die Zeichenkette „$AABCEADEAAB$“ vor, so ergibt sich für $n = 1$ die Matrix 4.1 und für $n = 2$ die Matrix 4.2. Jede Zeile stellt die Zähler der Folgezeichen auf ein mögliches n-Tupel dar. Aus jeder Spalte lässt sich ablesen, welche Zeichen wie oft auf ein n-Tupel folgten. Im Fall $n = 1$ ist in der ersten Zeile zu sehen, dass auf das Zeichen A zweimal das Zeichen A, zweimal das Zeichen B und einmal das Zeichen D folgte. Für $n = 2$ wird in der ersten Zeile deutlich, dass das n-Tupel „AAB“ in der untersuchten Zeichenkette zweimal vorkommt, da das B zweimal nach einem doppelten Vorkommen von A auftritt.

Intervalle [0;127] und [128;255]

Insbesondere in der JAR-Datei kommt dieser Test zur Geltung. Hierbei sollen die Intervalle verglichen werden. Für jedes Intervall werden der Mittelwert seiner Häufigkeitsverteilung und die Standardabweichung vom Mittelwert berechnet. Die Standardabweichung wird hierbei als Fehler angenommen, so dass gezeigt werden kann, ab welchem Degree die Mittelwerte der Intervalle zueinander gleichverteilt sind.

Degree-Zwei-Test

Um zu untersuchen, ob ein Paket mit einem Degree von 2 identifiziert werden kann, stehen zwei Funktionen zur Verfügung. Die erste liest eine Datei paketweise ein und führt eine Häufigkeitsanalyse der einzelnen Pakete durch. Hierbei wird die Standardabweichung der mittleren Häufigkeit berechnet, welche zum Vergleich dient.
Die zweite Funktion sucht nach einem bestimmten Degree. Hierzu wird die gesamte kodierte Datei anhand der Konfigurationsdatei eingelesen, und es werden alle hierbei möglichen Kombinationen ausprobiert. Das jeweils resultierende Paket wird untersucht und sein Koeffizientenvektor mit dem Referenz-Degree verglichen. Auf jedes der resultierenden Pakete, welches das gleiche Degree hat, wird eine Häufigkeitsanalyse durchgeführt. So können schließlich die Merkmale der einzelnen Pakete je nach Degree analysiert werden. Werden Pakete mit einem Degree von 2 gesucht, kann so geprüft werden, ob die resultierenden Pakete mit $d = 2$ von denen mit einem höheren Degree unterscheidbar sind.

4.2.2. Der Ewartungswert

In den Experimenten wird eine generierte Pseudo-Zufallszahlenfolge als Benchmark herangezogen. Hier sollen theoretisch das Verhalten bzw. die Eigenschaften einer Zufallszahlenfolge in den zuvor beschriebenen Testverfahren betrachtet werden.

Häufigkeitsverteilung: Da jede der möglichen Zahlen mit einer gleich großen Wahrscheinlichkeit auftritt, kommen alle Zahlen bei einer ausreichend großen Statistik gleichhäufig vor. Die Häufigkeitsverteilung der Zufallszahlenfolge sollte dementsprechend eine Gleichverteilung besitzen [4, S. 349].

Abstandstest: Die Wahrscheinlichkeit für einen konkreten Abstand in einer Zufallszahlenfolge läßt sich wie folgt berechnen:

$$\mathcal{P}(f_i) = \left(\frac{1}{N}\right)^2 * \left(1 - \frac{1}{N}\right)^i \tag{4.6}$$

wobei i die Anzahl der Zeichen zwischen zwei Vorkommnissen ist. N ist die Anzahl aller möglichen Zeichen. $\mathcal{P}(f_i)$ ergibt dann die Wahrscheinlichkeit, dass i Zeichen zwischen dem Vorkommen eines bestimmten Zeichens sind, also ein konkreter Abstand besteht. Multipliziert mit der Größe der Zufallszahlenfolge (S_{file}) ergibt sich schließlich die erwartete Anzahl an Vorkommnissen eines konkreten Abstandes:

$$\mu_i = \mathcal{P}(f_i) * S_{file}$$

***n*-Tupel-Suche:** Da in einer Zufallszahlenfolge jede Zahl die Wahrscheinlichkeit $\frac{1}{N}$ hat, gezogen zu werden, berechnet sich die Wahrscheinlichkeit für ein n-Tupel samt Folgezeichen wie folgt:

$$T_n = \left(\frac{1}{N}\right)^n * \left(1 - \frac{1}{N}\right) \tag{4.7}$$

wobei N die Anzahl aller möglichen Zeichen und n die Quantität des Tupels darstellt.

Intervalle [0;127] und [128;255]: Gemäß der Wahrscheinlichkeitsverteilung sind die Intervalle [0;127] und [128;255] einer Zufallszahlenfolge bei einer ausreichend großen Stichprobe gleichverteilt.

Degree-Zwei-Test: Auch beim Degree-Zwei-Test wird unter anderem auf eine Zufallszahlenfolge als Vergleich zurückgegriffen. Hierfür wird eine bestimmte Anzahl gleichgroßer Zufallsfolgen betrachtet und jeweils die Standardabweichung, σ, von der mittleren Häufigkeit berechnet. Alle σ der Pakete sind unabhängig voneinander. Betrachtet man nun eine große Menge an Paketen, ist die Wahrscheinlichkeit von sehr großen und sehr kleinen σ relativ gering. Es bildet sich eine Gauß'sche Glockenkurve um den Mittelwert aller σ herum, also eine Normalverteilung [18, S. 369], [4, S. 361].

4.2.3. Die Testdateien

Je Datei wird eine Zufallszahlenfolge generiert, die der Größe der jeweiligen Testdatei entspricht und Vergleichszwecken dient. Jede Datei wurde auch für den Fall der verlustfreien Kompression betrachtet.
Die Testdatei 1 ist ein Video im MPEG-2-Format (70 MBytes). Der Audio-Elementarstrom wurde entfernt, um nur die Strukturen des Video-ES zu untersuchen. Das Video wurde im MPEG-Kodiervorgang nur schwach komprimiert und liegt somit in relativ hoher Qualität vor. Das Video dauert ca. 3 Minuten und beinhaltet eine Mischung aus längeren Sequenzen mit wenig Bewegung und Sequenzen mit mehr Bewegung. Auffällig sind besonders lange Schwarzbild-Sequenzen. Insgesamt sind die Bilder eher dunkel.
Bei der Testdatei 2 handelt es sich um eine JAR-Datei. Sie beinhaltet Dateien mit Java-Bytecode (4.3 MBytes). Diese Dateien bilden kein semantisch zusammenhängendes Programm. Beim Zusammenfügen der JAR-Datei wurde der Bytecode nicht komprimiert. Es sind keine Java-Dokumentaionsdateien (Java-Doc) in der JAR enthalten, sondern ausschließlich Java-Bytecode.

4.2.4. Vorbereitung der Experimente

Für jedes Experiment wird eine entsprechende Zufallszahlenfolge als Benchmark generiert. Hierzu wird stets derselbe Pseudo-Zufallszahlengenerator genutzt.[2] Um eine vergleichbare statistische Häufigkeit zu erreichen, besteht jede Folge aus genauso vielen Bytes wie die entsprechende Testdatei. Die Größe der Testdateien wird so gewählt, dass eine korrespondierende Zufallszahlenfolge eine aussagekräftige Wahrscheinlichkeitsverteilung besitzt.
Die Testdateien sollen sowohl als Klartext als auch Fountain-kodiert untersucht werden. Die Kodierung erfolgt nach der in Abschnitt 3.2 beschriebenen Funktionsweise der Fountain Codes. Die Nutzdaten der erstellten kodierten Pakete werden ohne Koeffizientenvektor hintereinander in eine Datei geschrieben. Im Folgenden sind die für die Kodierung genutzten Parameter aufgelistet:

1. Jede Klartextdatei wird in 5000 Blöcke unterteilt.[3]
2. Pro Kodiervorgang werden 5000 kodierte Pakete erstellt.
3. Die Nutzlast der kodierten Pakete wird hintereinander in eine Datei geschrieben (kodierte Datei).
4. Die Koeffizientenvektoren werden in einer separaten Datei gespeichert (Konfigurationsdatei).
5. Es werden mehrere kodierte Dateien mit verschieden hohen Degrees erstellt.
6. Es wird keine konkrete Degree Distribution genutzt.
7. Die Testdateien werden ggf. vor der Kodierung komprimiert.

Um die Klartextdateien in genau 5000 Blöcke zu unterteilen, muss der Datenblock gegebenenfalls gepaddet werden. Dies sorgt zwar für künstliche Strukturen, da das Padding aber in der Praxis ein denkbarer Ansatz wäre, soll dies toleriert werden. Um nur einen Bezugspunkt zu behalten, wird die Zufallszahlenfolge nicht um die Anzahl der ggf. gepaddeten Bytes erweitert. Es werden 5000 kodierte Pakete erstellt, da dies eine ausreichend große Stichprobe für die hier angestellten Untersuchungen ist. Wie in Abs. 3.3.1 beschrieben, ist dies auch eine in der Praxis taugliche Anzahl. Die Experimente werden hauptsächlich mit den kompletten Datenströmen durchgeführt. Dies simuliert einen Angreifer, der alle Pakete mithört und sie somit insgesamt analysieren kann. In Abs. 4.3.2 wird auch kurz auf die Untersuchung einzelner Pakete eingegangen. Hierbei steht die Annahme im Vordergrund, dass auf diese Weise ein Angriff, wie er schon auf WEP ausgeübt wurde, erfolgreich durchgeführt werden kann. Um die Pakete aus der kodierten Datei zu extrahieren, wird die Konfigurationsdatei genutzt. Wie in Abs. 3.2.3 beschreiben, wird je Kodierung keine spezifische Degree Distribution

[2] Der Quellcode des genutzten Pseudo-Zufallszahlengenerators befindet sich im Anhang.
[3] Die Größe der Blöcke ist ggf. relevant für die implizite Vertraulichkeit (s. Abs. 5.2).

herangezogen, sondern ein fixer Wert als Degree genutzt.
Insbesondere sollen in dieser Arbeit die Effekte der Fountain-Kodierung ohne zusätzliche Verfahren untersucht werden. Da verlustfreie Kompression Redundanzen in der Datei entfernt, liegt es auf der Hand, dass Strukturen, die sich aus Redundanzen ergeben, verschwinden. Daraus folgt natürlich, dass die meisten Strukturen, nach denen in dieser Arbeit gesucht wird, nicht mehr aufspürbar sind. In der Praxis ist die Kompression allerdings ein gängiges Verfahren, welches zusätzlich für die Implementierung von Verschlüsselungsalgorithmen genutzt wird, um die Kryptoanalyse zu erschweren [24, S. 275]. Da Kompression auch für Fountain Codes ein durchaus denkbarer Ansatz ist, sollen die Resultate hier analysiert werden. Die Klartext Dateien werden in einem zusätzlichen Experiment vor der Kodierung mit der höchst möglichen Rate der GZIP-Kompression bearbeitet.[4]

4.3. Durchführung der Experimente

Die beschriebenen Testfunktionen werden nun nacheinander mit den Testdateien ausgeführt. Gesucht werden statistische Merkmale, die auf Regelmäßigkeiten hindeuten. Zunächst wird jede Datei als Klartext, dann kodiert untersucht. Beim MPEG-2-Format wird ein zusätzliches Experiment durchgeführt, um zu analysieren, ob ein Angreifer ein kodiertes Paket mit einem Degree von 2 als solches identifizieren kann. Um etwas detaillierter auf die Praxistauglichkeit der impliziten Vertraulichkeitserweiterung für Fountain Codes eingehen zu können, werden die Daten auch nach der Kompression kodiert und untersucht. Als Benchmark wird in den Resultaten auch das Ergebnis der erzeugten Pseudo-Zufallszahlenfolge gegenübergestellt. In den Resultaten der frequentiellen Verteilung wird von diesem Bezugspunkt nur die Verteilung eines Zeichens abgebildet. Da sich alle Zeichen ungefähr gleich verhalten, soll dies ausreichen.[5]

4.3.1. Experiment mit Testdatei 1

Zunächst werden die Testfunktionen mit der Testdatei 1 (MPEG-2-Format) ausgeführt. Sie wird sowohl unkodiert als auch Fountain-kodiert mit verschieden hohen Degrees untersucht. Es wird geprüft, wie hoch das Degree gewählt werden muss, damit die Strukturen weitestgehend obfuskiert sind. Um nicht nur einen Vergleich von kodierter zu nicht-kodierter Datei zu erhalten, wird zusätzlich eine gleichgroße (70 MByte) Folge aus Zufallszahlen als Referenz genutzt. Die Ergebnisse der Testdatei 1 sind in Tabelle 4.3 zusammengefasst.
Prüft man die nicht-kodierte Videodatei auf ihr Verhältnis aus Bits im Zustand 1 und 0, fällt

[4] Der Quellcode der Kompressionsfunktion befindet sich im Anhang.
[5] Die Zahlenwerte der Resultate befinden sich im Anhang.

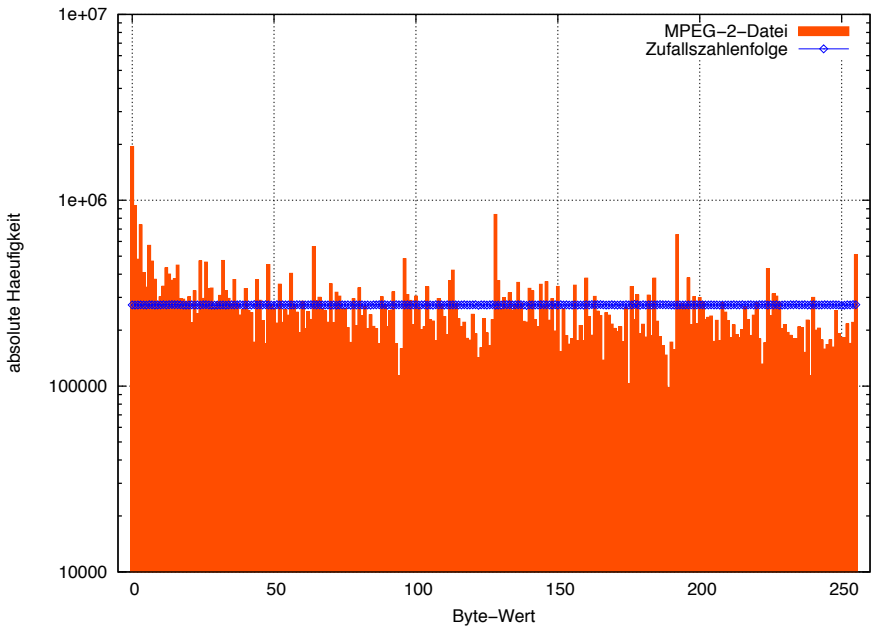

Abbildung 4.1.: Häufigkeitsverteilung der MPEG-2-Datei (logarithmische Darstellung)

auf, dass dieses in Relation zur Zufallszahlenfolge recht unausgeglichen ist. So besteht die Datei aus 55.58% Bits im Zustand Null, gegen 44.42% Bits im Zustand Eins. Bei der Zufallszahlenfolge hingegen findet sich mit je ca. 50% ein ausgeglichenes Verhältnis. Wie in Abbildung 4.1 deutlich wird, ist die Häufigkeitsverteilung aller 8-Bit-Zustände in der MPEG-2-Datei ebenfalls unausgewogen. Für eine Gleichverteilung müsste bei einer Dateigröße von 70 MBytes jedes Zeichen im Schnitt 273437.5mal (0.390625% der Datei) vorkommen. Tatsächlich streuen die Einzelhäufigkeiten um 148975.43 Zeichen, was einem prozentualen Anteil von 0.2128% der Datei entspricht. Besonders auffällig sind in Abbildung 4.1 die Peaks bei $0x00$ (Interger-Wert: 0), $0x80$ (Integer-Wert: 128) und $0xC0$ (Integer-Wert: 192).
Beispielhaft soll nun das Zeichen $0x00$ betrachtet werden. Mit einem prozentualen Anteil von 2.78% an der Gesamtdatei kommt es am häufigsten vor. Eine mögliche Begründung hierfür ist der immer wiederkehrende Header des PES (paketisierter Elementarstrom). Jeder Header beginnt mit dem Präfix $0x000001$ [23] [35]. Auch das Byte $0x01$ ist deshalb mit einem Anteil von 1.34% der Gesamtdatei relativ häufig. Sucht man nach dem n-Tupel des Zeichens $0x00$ mit $n = 2$ und dem Folgezeichen $0x01$, also der Zeichenkette $0x000001$ in der MPEG-2-Datei, so ist sie 106802mal zu finden. Somit gehören $106802 \times 2 = 213604$ der gesamten $0x00$-Vorkommen zu dem Präfix eines Headers. Folglich kann es nicht sein, dass der Peak bei $0x00$ nur durch die Header-Informationen verursacht wird. Sucht man nach n-Tupeln des Zeichens $0x00$ mit sehr großen n, fallen lange Sequenzen des Zeichens $0x00$ auf. Hierfür könnten Bild-Informationen verantwortlich sein, die zum Beispiel durch dunkle Passagen im Bildmaterial verursacht werden und durch die schwache MPEG-2-Kompression nicht redu-

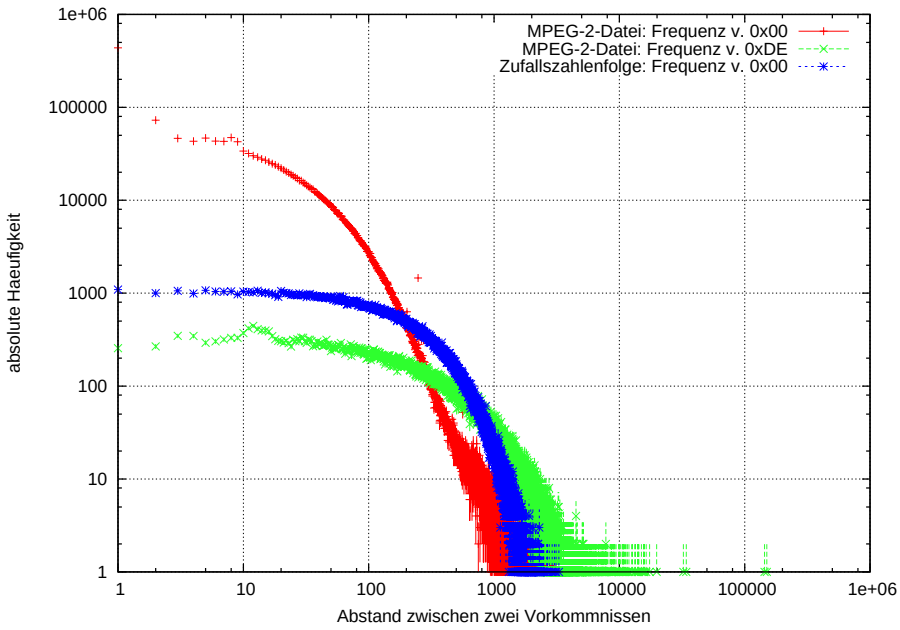

Abbildung 4.2.: Frequentielle Verteilung der MPEG-2-Datei (doppelt logarithmische Darstellung)

ziert wurden. Inwieweit der Inhalt und die Qualität des Videos eine Rolle spielen, soll hier aber nicht weiter untersucht werden.
Betrachtet man nun die Referenzdatei aus Zufallszahlen (Abb. 4.1, blau) hat diese eine Standardabweichung von 537.32, was einem Anteil von 0.0007676% der Datei entspricht. Sie ist somit also annähernd gleichverteilt, wie auch in Abbildung 4.1 (blau) ersichtlich ist. Wie hierdurch deutlich wird, unterscheidet sich die MPEG-2-Datei in ihrer Häufigkeitsverteilung sehr stark von der Zufallszahlenfolge.
Der Abstandstest zeigt, dass die Zeichenfrequenzen der unkodierten MPEG-2-Datei ebenfalls stark variieren. So kommt das Byte $0x00$ im Schnitt alle 36.02 Zeichen wieder vor, $0xDE$ nur alle 532.98 Zeichen. In Abbildung 4.2 ist die frequentielle Verteilung des Bytes $0x00$ (rot) und $0xDE$ (grün) zu sehen. Die Frequenz des Zeichens $0x00$ der Zufallszahlen ist in der Abbildung 4.2 blau dargestellt. In der MPEG-2-Datei ist eine häufig wiederkehrende Struktur des Bytes $0x00$ zu erkennen. Da es weitaus häufiger vorkommt als in der Zufallszahlenfolge, müssen auch die Abstände, in denen es erscheint, wesentlich geringer sein. Ein Abstand von 1, also keine Zeichen zwischen zwei Vorkommen, gibt es 436486mal. Da sich das Header-Präfix aus einem doppelten Vorkommen des Bytes $0x00$ und einem einfachen Vorkommen von $0x01$ bildet, trägt auch diese Struktur zu den geringen Abständen bei. Aber auch die langen Sequenzen des Zeichens $0x00$, die ggf. durch das Bildmaterial verursacht werden, machen sich hier bemerkbar. Des Weiteren fällt auf, dass die Kurve des Bytes $0x00$ der MPEG-2-Datei wesentlich steiler verläuft und etwas früher und stärker abflacht als die

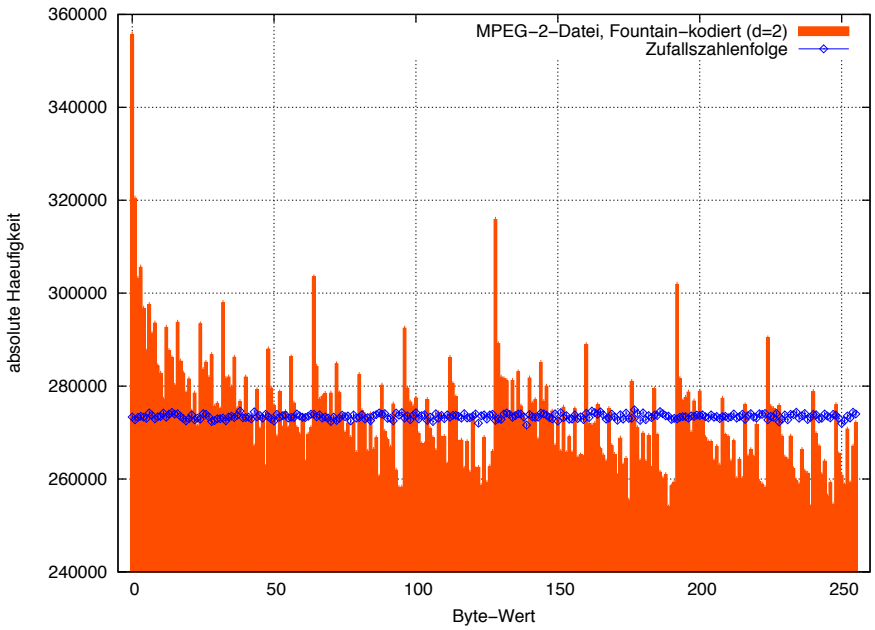

Abbildung 4.3.: Häufigkeitsverteilung der MPEG-2-Datei, Fountain-kodiert mit $d = 2$

der Zufallszahlenfolge. Das Verhältnis zu kleinen und großen Abständen ist hier also sehr viel weniger ausgeglichen. Die Kurve der Zufallszahlenfolge hat einen exponentiellen Abfall, gemäß der Formel 4.6, welche in Abschnitt 4.2.2 definiert wurde.
Das Byte $0xDE$ kommt in der MPEG-2-Datei relativ zum Byte $0x00$ eher selten vor. Auch im Vergleich zum Zeichen $0x00$ der Zufallszahlenfolge ist es weniger häufig und hat dementsprechend eine sehr flache Kurve. So sind in einer nicht-kodierten MPEG-2-Datei viele solcher Abweichungen von einer Zufallszahlenfolge zu finden, die Rückschlüsse auf die interne Struktur, wie zum Beispiel dem Header-Präfix oder sogar inhaltliche Informationen, auf das Dateiformat und die Daten erlauben.
Ebenfalls aufällig ist die unterschiedlich hohe Entropie von 7.861921 Bits pro Byte der MPEG-2-Datei zu 7.999997 der Zufallszahlenfolge. Bei der Zufallszahlenfolge hat sie fast den maximalen Wert von 8 erreicht. Die Auftrittswahrscheinlichkeit der einzelnen Zeichen ist sehr ausgeglichen. Die Differenz beider beträgt 0.138076. Diese Werte werden zum Vergleich nach der Kodierung wieder aufgegriffen.
Nun soll untersucht werden, inwieweit sich diese sichtbaren Strukturen durch die Kodierung obfuskieren lassen. Hierfür wird der Klartext mit einem Degree von 2 kodiert.
Das Verhältnis aus Einsen und Nullen liegt jetzt bei 50.62% Bits im Zustand Null zu 49.38% im Zustand Eins. Das Verhältnis nähert sich also dem der Zufallszahlenfolge deutlich an. Die Häufigkeiten der einzelnen Byte-Zustände weichen nicht mehr so stark voneinander ab. Die Werte streuen nur noch um 11633.96 Zeichen um den Mittelwert von 273437.5 Zeichen. Die prozentuale Abweichung ist also um eine 10er Potenz auf 0.01662% gesunken. Wie im

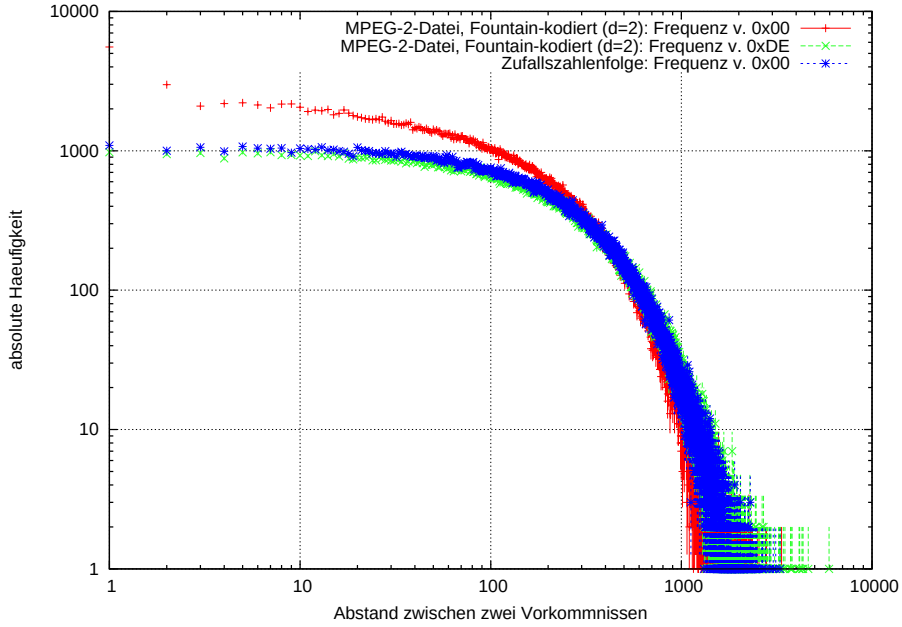

Abbildung 4.4.: Frequentielle Verteilung der MPEG-2-Datei, kodiert mit $d = 2$ (doppelt logarithmische Darstellung)

Vergleich von Abbildung 4.1 mit Abbildung 4.3 deutlich wird, gleicht sich die Verteilung an die der Zufallszahlen zwar an, das Verhältnis der einzelnen Häufigkeiten zueinander bleibt aber relativ ähnlich. So sind die Peaks an den gleichen Stellen zu sehen. Die Strukturen werden gestaucht, bleiben jedoch noch sichtbar.
Auch beim Abstandstest wird deutlich, dass die Ergebnisse nach der Kodierung sich etwas mehr denen der Zufallszahlen annähern. So kommt das Byte $0x00$ nun im Schnitt alle 196.82 Zeichen vor. $0xDE$ hat eine Frequenz von 270.4 Zeichen. Die Kluft zwischen den einzelnen Bytes hat sich also stark reduziert. Wie in Abbildung 4.4 verdeutlicht, nähern sich die beiden Kurven der mit $d = 2$ kodierten MPEG-2-Datei (rot, grün) mehr an die der Zufallszahlen (blau) an. Die Entropie der mit einem Degree von 2 kodierten Datei beträgt 7.998729 Bits pro Byte. Vergleicht man diesen Wert mit der Entropie der Zufallszahlenfolge, ist dies eine Differenz von 0.001268. Sie ist also im Vergleich zu der nicht-kodierten Datei um zwei 10er Potenzen gesunken. Die Auftrittswahrscheinlichkeit der einzelnen Zeichen gleicht sich also entsprechend der Häufigkeit mit an.
Wird die Klartextdatei mit einem Degree von 3 kodiert, verstärken sich die bei einer Kodierung mit $d = 2$ beobachteten Effekte. Dies wird allerdings nicht bei allen Tests gleichermaßen deutlich. Bei Betrachtung der einzelnen Bits lassen sich die Unterschiede erst hinter dem Komma erkennen. So befinden 50.07% der Bits im Zustand 0, die restlichen 49.93% im Zustand 1. Die Streuung um den Mittelwert aller Häufigkeiten reduziert sich allerdings noch einmal um eine 10er Potenz auf 0.001904%, was einem absoluten Wert von 1332.5

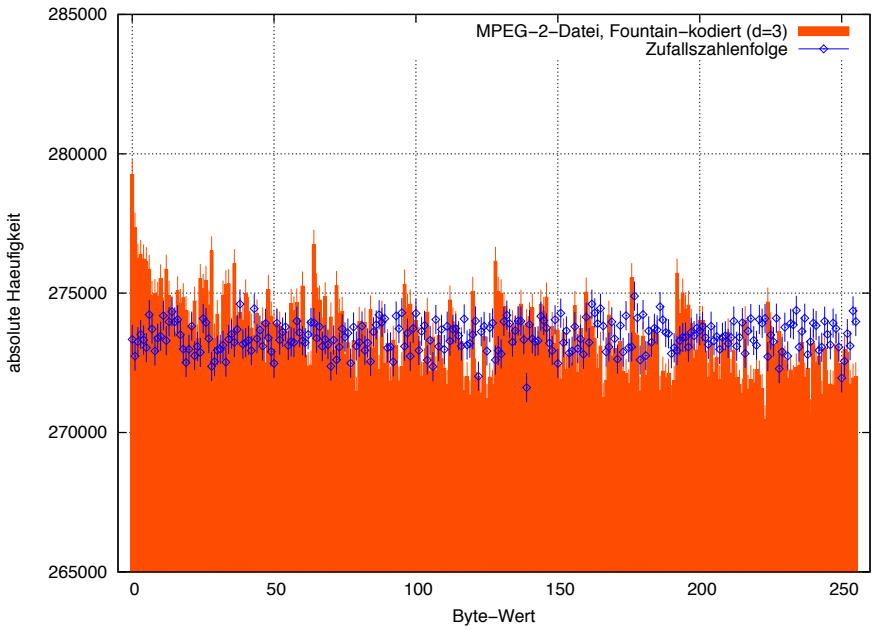

Abbildung 4.5.: Häufigkeitsverteilung der MPEG-2-Datei, Fountain-kodiert mit $d = 3$

Zeichen entspricht. Sie ist somit aber immer noch deutlich größer als die Streuung der Zufallszahlen. In Abb. 4.5 ist zu sehen, dass die Häufigkeiten dennoch immer näher an die der Zufallszahlenfolge herankommen. Die Grundstruktur ist noch leicht zu erkennen. So sind die vorher sehr offensichtlichen Peaks bei den Byte-Zuständen 0, 128 und 192 noch erkennbar. Beim Abstandstest (Abb. 4.6) wird deutlich, dass sich die Bytes $0x00$ und $0xDE$, die sich zuvor stärker unterschieden, nun schon fast identisch sind. Das Byte $0x00$ kommt jetzt durchschnittlich alle 250.67 Bytes vor, $0xD5$ alle 257.7. In Abbildung 4.6 scheinen $0x00$ (rot) und $0xDE$ (grün) zwischen $0x00$ der Zufallszahlen (blau) immer noch etwas durch, sind aber deutlich schwerer von der Zufallszahlenfolge zu unterscheiden, als es in Abbildung 4.4 der Fall ist. Die Entropie vergrößert sich auf 7.999983 und ist der der Zufallszahlenfolge schon recht ähnlich. Die Differenz beträgt nur noch 0.000014 Bits pro Byte.
Es gehen durch fortlaufende Kodierung mit höheren Degrees immer mehr Merkmale der Strukturen verloren. Auffällig wird insbesondere, dass die resultierenden Messwerte zunehmend gestaucht werden (vgl. Abb. 4.1 zu 4.5) und immer mehr Ähnlichkeit mit dem Verhalten der Zufallszahlenfolge haben. Gewisse Charakteristika bleiben jedoch trotz Kodierung bis zu einem Degree von 3 noch relativ leicht erkennbar.
Erst ab einem Degree von 4 sind die Strukturen soweit obfuskiert, dass sie kaum noch von denen der Zufallszahlen zu unterscheiden sind. In Abbildung 4.7 sind die in Abbildung 4.5 noch auffälligen Peaks nicht mehr zu sehen. Die Häufigkeitsverteilung zeigt, dass die Bytes nun fast gleichverteilt sind. Die meisten Werte liegen im Bereich des erwarteten Wertes. Noch bestehende Unterschiede zu der Zufallszahlenfolge sind auf zufällige Schwankungen

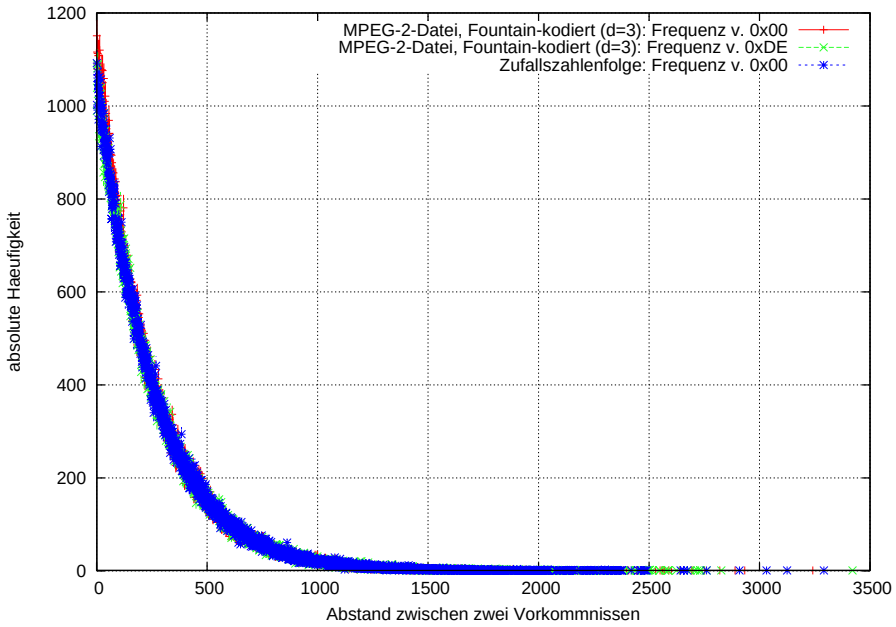

Abbildung 4.6.: Frequentielle Verteilung der MPEG-2-Datei, Fountain-kodiert mit $d = 3$

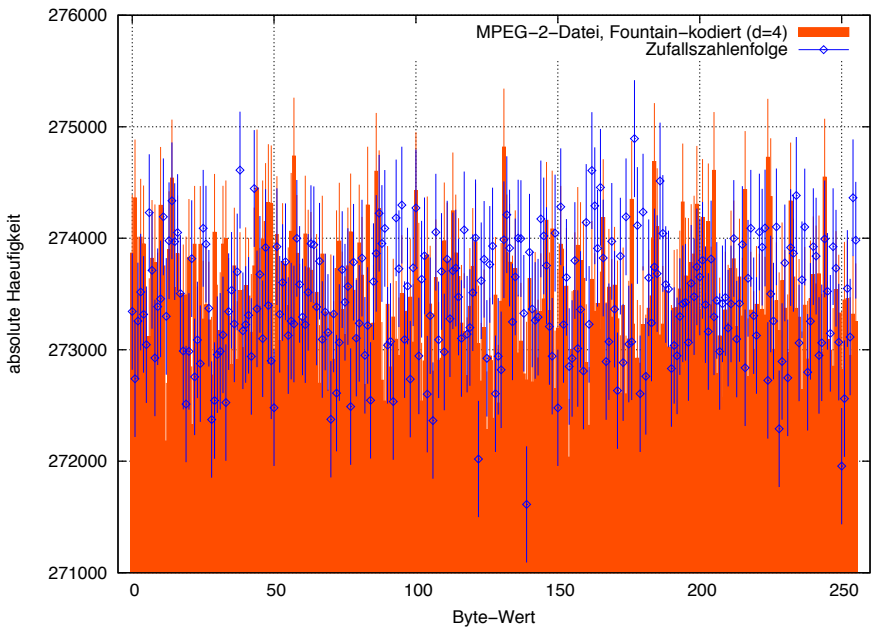

Abbildung 4.7.: Häufigkeitsverteilung der MPEG-2-Datei, Fountain-kodiert mit $d = 4$

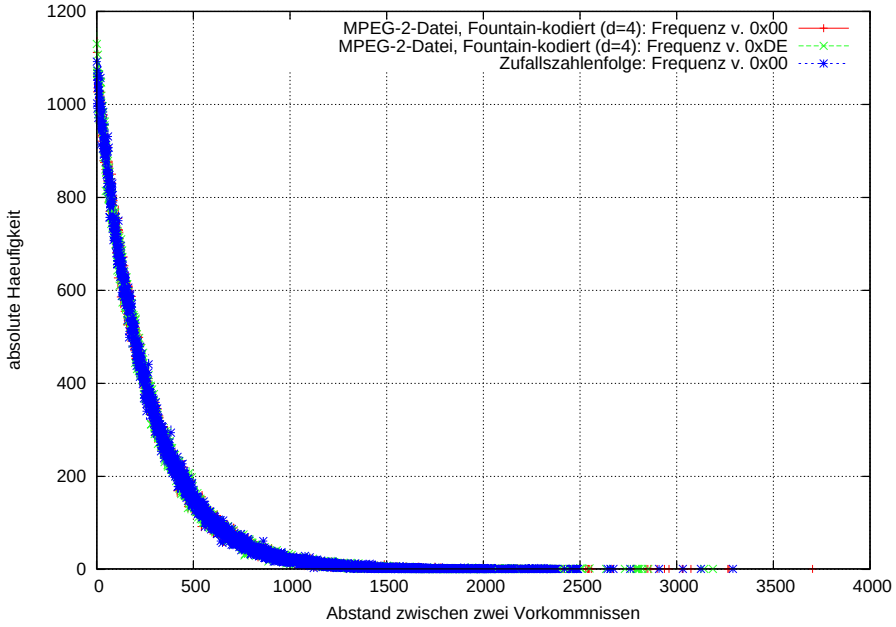

Abbildung 4.8.: Frequentielle Verteilung der MPEG-2-Datei, Fountain-kodiert mit $d = 4$

DEGREE	BITS 0/1 (%)	σ	FREQ. 0X00	FREQ. 0XDE	ENTROPIE
$d = 1$	55.58/44.42	148975.43	36.02	532.98	7.861921
$d = 2$	50.62/49.38	11633.96	196.82	270.4	7.998729
$d = 3$	50.07/49.93	1332.5	250.67	257.7	7.999983
$d = 4$	50.01/49.99	527.28	255.6	256.07	7.999997
$Zufallsfolge$	50/50	537.32	256.1	255.55	7.999997

Tabelle 4.3.: Resultate der Experimente mit Testdatei 1

zurückzuführen. Von einer idealen Gleichverteilung weichen die Werte im Schnitt um 527.28 Bytes ab. Bei der Zufallszahlenfolge sind es 537.32 Bytes. Auch die anderen Tests zeigen, dass sich eine mit $d = 4$ kodierte Datei fast die gleichen Eigenschaften wie eine Zufallszahlenfolge hat. So ist das Verhältnis der Bits im Zustand 0 zu denen im Zustand 1 so gut wie ausgeglichen. Auch durch den Abstandstest können keine Unterschiede zur Zufallszahlenfolge aufgedeckt werden (Abb. 4.8). Die gemessene Entropie ist genauso hoch wie die der Zufallszahlenfolge (s. Tabelle 4.3).
Wird eine MPEG-2-Datei mit Fountain Codes kodiert, lassen sich die Strukturen ab einem Degree von 4 so weit obfuskieren, dass diese mit der hier durchgeführten statistischen Strukturanalyse nicht deutlich sichtbar gemacht werden können. Werden also die Strukturen der gesamten kodierten Datei betrachtet, ist das Niveau an Vertraulichkeit ausreichend, um nicht

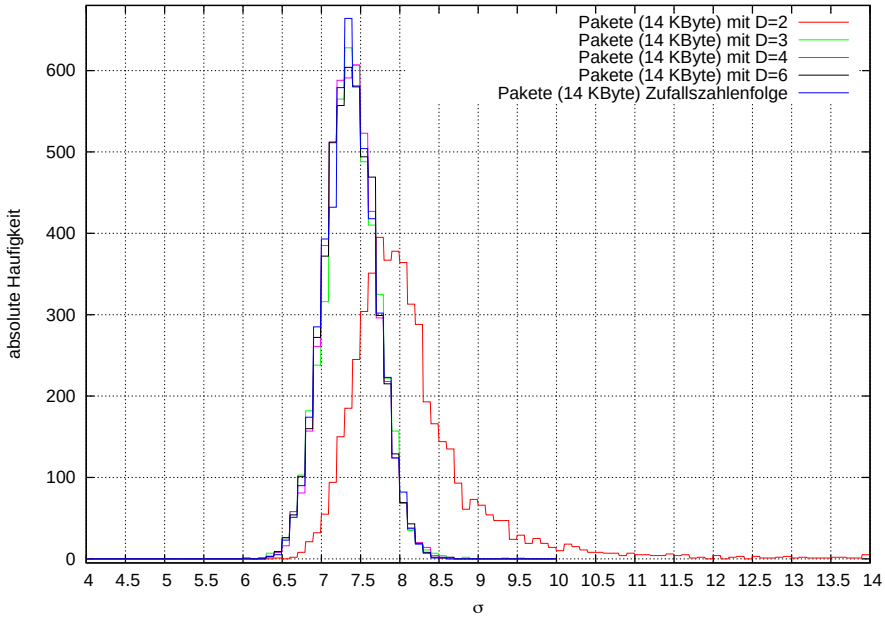

Abbildung 4.9.: Verteilung der Häufigkeitsabweichungen kodierter Pakete (MPEG-2) mit $d = 2, 3, 4$ und 6

durch die in Abschnitt 4.2.1 beschriebenen Tests kompromittiert zu werden. Im Folgenden Abschnitt werden die einzelnen kodierten Pakete näher betrachtet.

4.3.2. Experiment mit kodierten Paketen der Testdatei 1

Als nächstes wird die Frage gestellt, ob der Angreifer ein kodiertes Paket (X) mit einem Degree von 2 als solches entlarven kann. Hierzu sollen einige Pakete mit einem Degree von 2 und andere mit einem höheren Degree untersucht und verglichen werden. Als Benchmark dient die Standardabweichung von der mittleren Häufigkeit aller Bytes und zusätzlich die gleiche Anzahl an gleichgroßen Paketen aus Pseudo-Zufallszahlen. Jedes X und jede Folge aus Zufallszahlen hat eine feste Länge von 14000 Bytes. Sind die Zeichen in einem Paket gleichverteilt, käme jedes Zeichen im Schnitt $\frac{14000}{256} \approx 54.69$mal vor. Vom Degree 2, 3 und 4 liegen je 5000 kodierte Pakete vor. Die Ergebnisse sind in Tabelle 4.4 zusammengefasst. In Abbildung 4.9 sind die einzelnen Standardabweichungen der kodierten Pakete zu sehen. Besonders auffällig sind alle X mit $d = 2$. Sie weichen noch stark von den Zufallszahlen und auch von allen X mit einem höheren Degree ab. So ist der maximale Wert eines kodierten Paketes mit $d = 2$ bei $\sigma = 82.27$. Bei $d = 3$ liegt die maximale Standardabweichung bei 9.59. Die Zufallszahlenfolge hat den maximalen Wert bei $\sigma = 8.56$ (s. Tabelle 4.4). Bei den in Abbildung 4.9 dargestellten X mit $d = 2$ befindet sich der Mittelwert der X-Achse bei

DEGREE	σ_{min}	σ_{avg}	σ_{max}
$d = 2$	6.43	8.34	82.27
$d = 3$	6.27	7.38	9.59
$d = 4$	6.37	7.38	8.6
$d = 6$	6.31	7.38	8.65
$Zufallsfolge$	6.05	7.37	8.56

Tabelle 4.4.: Resultate der Experimente mit kodierten Paketen der Testdatei 1

8.34, wobei nur 1311 Werte, also gut ein Viertel, über dem Mittelwert liegen. Der Mittelpunkt wird also durch „Ausreißer“ stärker nach rechts verschoben und fällt etwas flacher aus als die Kurven der X mit höheren Degrees.
In einem fiktiven Szenario wird davon ausgegangen, dass ein Angreifer alle Datenpakete mithört. Er kennt die verwendete Degree Distribution (in diesem Fall ausschließlich $d = 3$). Bekannt ist ihm die Abbildung 4.9, auf dessen Verteilung er sich bezieht. Er hat also zumindest Wissen über das genutzte Dateiformat. In einem ersten Anlauf hat er die Möglichkeit, alle X miteinander zu verknüpfen. Da es insgesamt 5000 kodierte Pakete gibt, sind

$$\binom{n}{k} = \binom{5000}{2} = 12497500$$

verschiedene Kombinationen denkbar. Nun soll überprüft werden, mit welcher Wahrscheinlichkeit er ein X findet, dessen Degree er als $d = 2$ identifizieren kann. Hierzu werden alle Kombinationen untersucht. Es haben 12474658 der resultierenden X einen Degree von 6, 22834 einen Degree von 4 und 8 kodierte Pakete einen Degree von 2. Keines wurde komplett weggekürzt. Für den Angreifer besteht hierbei die Möglichkeit, ohne Anhaltspunkt ein $d = 2$ zu erraten oder anhand von statistischen Verfahren Pakete auszuwählen, bei denen er mit einer bestimmten Wahrscheinlichkeit davon ausgehen kann, dass sie einen Degree von 2 haben. Hierbei soll auf Möglichkeit zwei eingegangen werden. Er verknüpft alle möglichen Kombinationen miteinander und vergleicht die Standardabweichung aller resultierenden kodierten Pakete miteinander. Um einen Bezugspunkt hierfür zu bekommen, wählt er einen geeigneten Wert aus Abbildung 4.9 und geht davon aus, dass alle Pakete, die ein höheres σ als dieser Bezugspunkt haben, mit einer hohen Wahrscheinlichkeit einen Degree von 2 haben. Zuerst nimmt er den Mittelwert der kodierten Pakete mit einem Degree von 2 (Abb. 4.9, rot) als Bezugspunkt. Der Mittelwert liegt bei $\sigma \approx 8.34$. Bei allen resultierenden kodierten Paketen findet er insgesamt 23206 X mit einem höherem σ. Tatsächlich hat hiervon nur ein Paket einen Degree von 2. Die Wahrscheinlichkeit das $d = 2$ Paket zu erraten, beträgt also 1 zu 23205.
Um die Wahrscheinlichkeit zu erhöhen, wählt er in einem weiteren Versuch den Wert des größten $d = 3$ Pakets aus Abbildung 4.9. Dieses liegt bei $\sigma \approx 9.59$. Jetzt gibt es nur noch

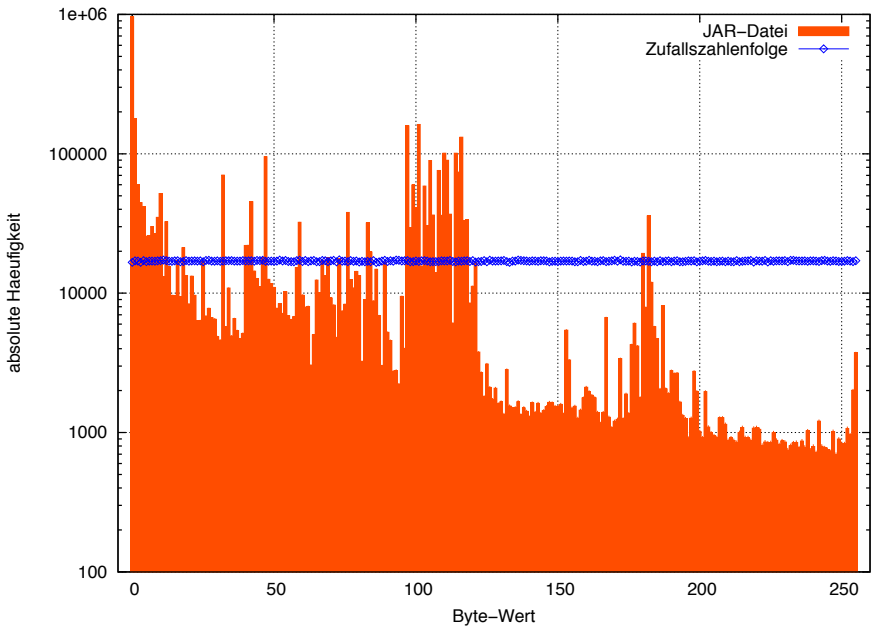

Abbildung 4.10.: Häufigkeitsverteilung der JAR-Datei (logarithmische Darstellung)

ein X, welches größer ist, und zwar eines mit $\sigma \approx 10.54$. Hierbei handelt es sich um ein $d = 2$ Paket. Somit hat er ein Paket mit einem Degree von 2 isoliert.
Selbstverständlich darf aus diesem Beispiel keine allgemeingültige Aussage getroffen werden. Ein Angreifer könnte aber in einem ähnlichen Verfahren einen geeigneten Wert finden, anhand dessen er mit einer ausreichend hohen Wahrscheinlichkeit davon ausgehen kann, dass es sich bei dem isolierten Paket um ein $d = 2$ Paket handelt. Außerdem kann er weiterführend die resultierenden Pakete miteinander verknüpfen. So kann er deutlich mehr Pakete erhalten, als sich in dem ersten Kombinationsversuch ergaben. Es ist nur eine Frage der Zeit, bis er genug Pakete gesammelt hat, um das System zu brechen.
Obwohl dieses Beispiel von unwahrscheinlich guten Bedingungen für den Angreifer und einem festen Wert als Degree ausgeht, kann der Erfolg einer Attacke, wie sie in Abschnitt 3.4.2 dargelegt wurde, nicht ausgeschlossen werden. Somit kann das Niveau der impliziten Vertraulichkeit dieses Verfahrens ohne zusätzliche Verfahren für das MPEG-2-Format nur als sehr gering eingestuft werden.

4.3.3. Experiment mit Testdatei 2

In diesem Experiment wird die Testdatei 2 (JAR-Format) untersucht. Die Ergebnisse sind in Tabelle 4.5 zusammengefasst. In der JAR-Datei fällt der Unterschied zwischen 0-Bits und 1-Bits am stärksten auf. Es haben 65.95% den Zustand 0 und 34.05% den Zustand 1. In

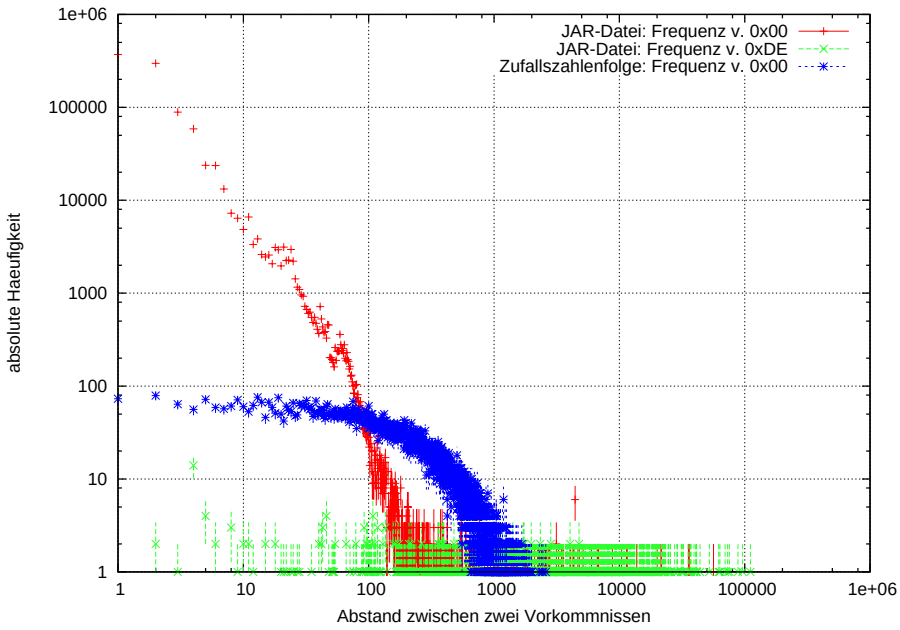

Abbildung 4.11.: Frequentielle Verteilung der JAR-Datei (doppelt logarithmische Darstellung)

der Häufigkeitsverteilung (Abb. 4.10) wird dieses gleichermaßen deutlich. Das Byte $0x00$ überwiegt mit 22.1% der gesamten Datei deutlich gegenüber den anderen Zuständen. Das Zeichen $0xDE$ macht mit einer absoluten Häufigkeit von 808 nur 0.019% der Datei aus. Die Standardabweichung von der mittleren Häufigkeit liegt bei 64432.68 Zeichen, was einen Anteil von 1.48% der gesamten Datei entspricht. Bei der Zufallszahlenfolge ist die Standardabweichung vom Mittelwert 135.47 Zeichen (0.0031%). Zudem liegen 93.61% aller Zeichen zwischen dem Integer-Wert 0 und 128. Hierbei fallen besonders die Peaks zwischen $0x61$ (Integer-Wert: 97) und $0x7A$ (Integer-Wert: 122) auf. Die Bytes über Integer-Wert 128 kommen bis auf wenige kleine Peaks selten vor. Sie liegen fast alle unter den Werten der Zufallszahlenfolge (Abb. 4.10, blau). In der weiteren Analyse werden die Unterschiede zwischen den Intervallen $[0; 127]$ und $[128; 255]$ noch näher untersucht.
Durch den Abstandstest werden die unterschiedlichen Frequenzen der JAR-Datei deutlich. In der Abbildung 4.11 sind die Unterschiede der Zeichen $0x00$ (rot) und $0xDE$ (grün) zu sehen. Durch das seltene Vorkommen des Zeichens $0xDE$ bildet sich hier kaum eine sichtbare Kurve. Der gleiche Abstand kommt hier selten mehr als einmal vor. Viele Punkte haben daher den gleichen Wert auf der Y-Achse. Das Byte $0x00$ fällt im Vergleich zur Kurve der Zufallszahlen (blau) sehr steil ab. Kleine Abstände (< 100) sind gegenüber größeren Abständen relativ häufig. So gibt es in der JAR-Datei 370307mal ein doppeltes Vorkommen, also einen Abstand von 0 des Zeichens $0x00$.
Auch die Entropie der JAR-Datei ist mit 5.873492 relativ niedrig. Die der gleichgroßen Zufallszahlenfolge beträgt 7.999954.

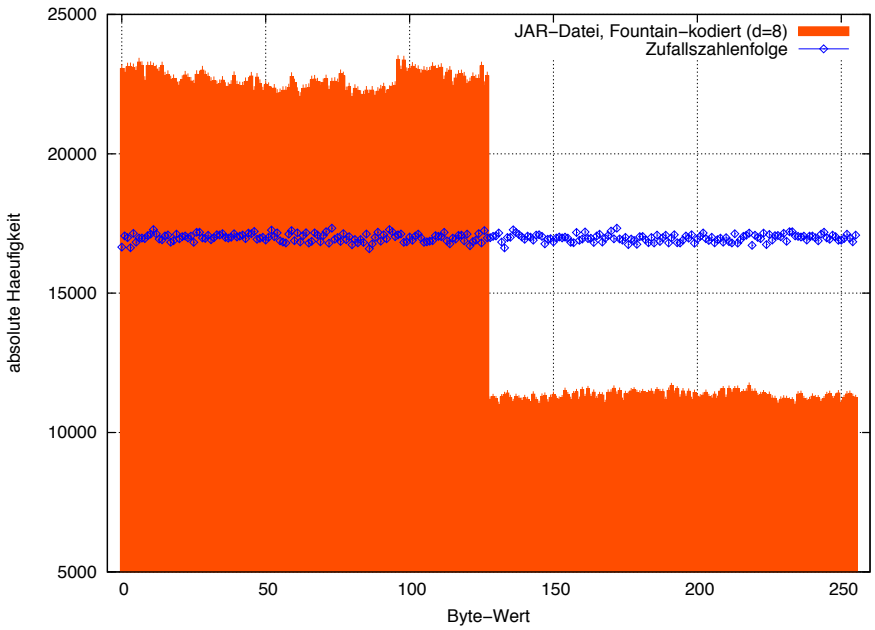

Abbildung 4.12.: Häufigkeitsverteilung der JAR-Datei, Fountain-kodiert mit $d = 8$

Sogar nach der Kodierung mit einem Degree von 8 liegt das Verhältnis der Bits noch bei 52.09% Nullen zu 47.91% Einsen. Betrachtet man Abbildung 4.12, so sticht der Unterschied in der Häufigkeitsverteilung beider Intervalle noch stark hervor. Bei den Bytes im Intervall $[0; 127]$ sind noch strukturelle Erscheinungen zu erkennen. So ist beim Integer-Wert 97 ein kleiner Anstieg zu bemerken, der über die folgenden Zeichen bestehen bleibt. Es wird ein weiterer Effekt deutlich: Die Strukturen der Verteilung innerhalb der Intervalle lassen sich relativ gut obfuskieren, die unterschiedliche Verteilung beider Intervalle zueinander bleibt jedoch erhalten. Auch die Standardabweichung von der mittleren Häufigkeit ist mit 5670.66 Zeichen (0.13036% der gesamt Datei) noch sehr groß.
Im Abstandstest wird ein großer Unterschied zur nicht-kodierten Datei deutlich (vgl. Abb. 4.13 zu Abb. 4.11). Die Strukturen sind schon relativ gut obfuskiert, aber für das hohe Degree von 8 sind noch sehr klare Unterschiede zur Zufallszahlenfolge zu sehen. Die Kurve des Zeichens $0x00$ (Abb. 4.13, rot) liegt noch überhalb der Zufallszahlenfolge (Abb. 4.13, blau), während die Kurve von $0xDE$ (Abb. 4.13, grün) unterhalb liegt. Die Unterschiede sind immer weniger das Resultat aus den Strukturen innerhalb der Intervalle. Vielmehr sind sie aus den unterschiedlichen absoluten Häufigkeiten der Intervalle zueinander ableitbar. Je mehr sich die Häufigkeiten mit zunehmendem Degree annähern, desto mehr ähnelt sich auch die frequentielle Verteilung der einzelnen Zeichen. Dies macht sich auch in der Entropie bemerkbar: So ist diese auch bei einem Degree von 8 mit 7.918126 noch um 0.081828 geringer als die der Zufallszahlenfolge.
Bei einem Degree von 16 sind die Strukturen in der Häufigkeitsverteilung innerhalb der Inter-

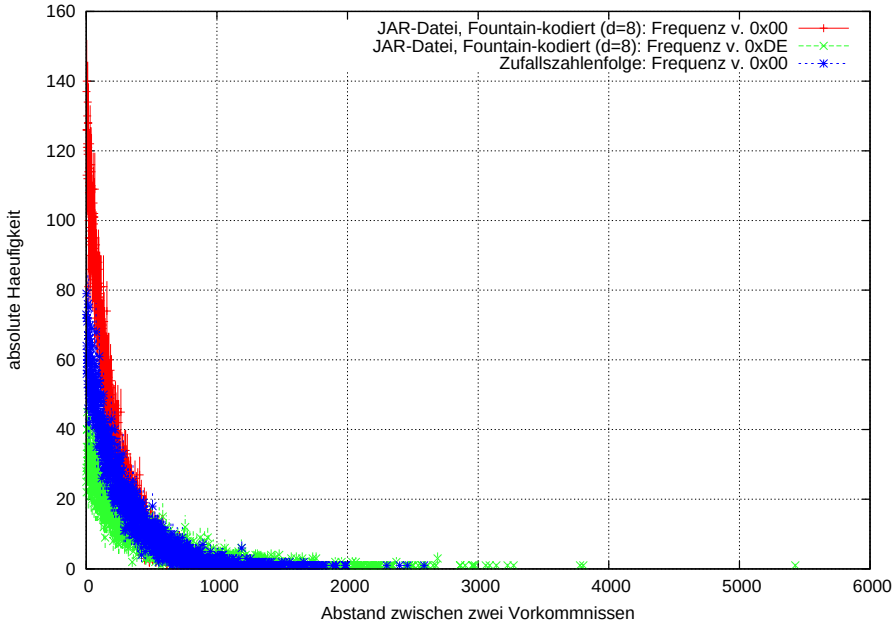

Abbildung 4.13.: Frequentielle Verteilung der JAR-Datei, Fountain-kodiert mit $d = 8$

valle nicht mehr sichtbar (s. Abb. 4.14). Die Kluft zwischen den Werten im Intervall $[0; 127]$ zum Intervall $[128; 255]$ ist hingegen noch zu sehen. So ist auch das Verhältnis von Bits im Zustand 0 zum Zustand 1 mit 50.68% zu 49.32% noch relativ unausgeglichen. Auch die Standardabweichung von der mittleren Häufigkeit und die Entropie machen dies deutlich. So beträgt $\sigma = 1865.53$ Zeichen (0.04289% der Datei) und $H = 7.991287$ Bits pro Byte. Damit ist die Entropie noch 0.008667 geringer als die der Zufallszahlenfolgen.
Die Unterschiede zur Zufallszahlenfolge sind auch beim Abstandstest erkennbar. Das Byte $0x00$ der JAR-Datei (Abb. 4.15, rot) kommt mit einem durchschnittlichen Abstand von 232.58 Zeichen etwas häufiger als die Zufallszahlenfolge (alle 261.17 Zeichen) vor und liegt somit etwas oberhalb der Kurve der Zufallszahlen (Abb. 4.15, blau). Beide haben aber einen ähnlichen Verlauf. Gleiches gilt für den Verlauf der Kurve von $0xDE$ der JAR-Datei (Abb. 4.15, grün), welche bei einem durchschnittlichen Abstand von 285.9 Zeichen etwas unterhalb der Zufallszahlenfolge verläuft. Dass die Kurven etwas versetzt zueinander sind, ist nun fast ausschließlich in den unterschiedlichen Häufigkeitsverteilungen der Intervalle zueinander begründet.
Wird das Degree auf 32 erhöht, fallen in der frequentiellen Verteilung keine Unterschiede mehr auf. In Abbildung 4.16 haben die Kurven der Zeichen einen fast identischen Verlauf.
Die Unterschiede zwischen den Werten der beiden Intervalle sind hingegen noch immer zu sehen. In Abbildung 4.17 wird deutlich, dass die Werte im Intervall $[128; 255]$ wesentlich weiter um ihren Mittelwert streuen als die Werte im Intervall $[0; 127]$. Zurückzuführen ist dies auf die ursprüngliche Häufigkeitsverteilung der nicht-kodierten JAR-Datei (s. Abbildung

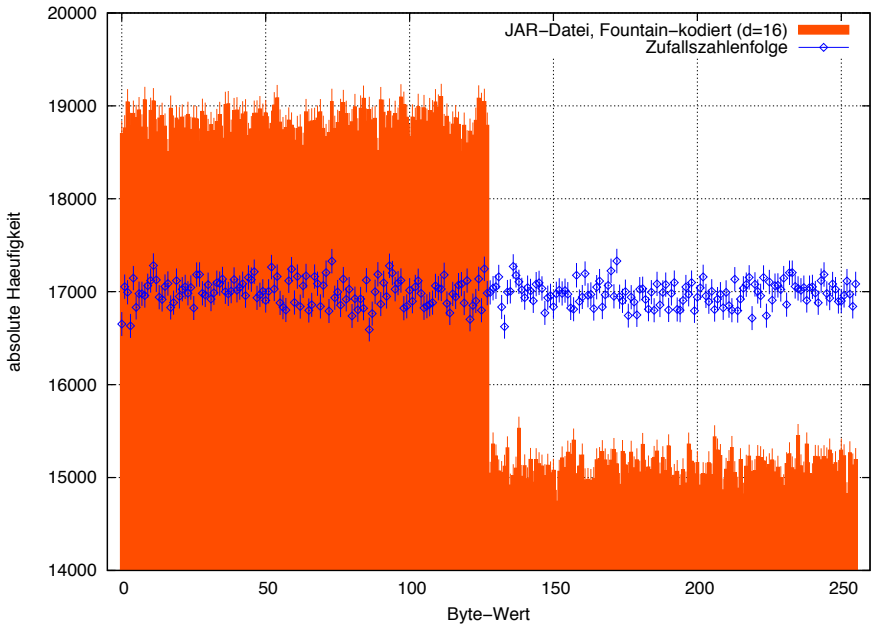

Abbildung 4.14.: Häufigkeitsverteilung der JAR-Datei, Fountain-kodiert mit $d = 16$

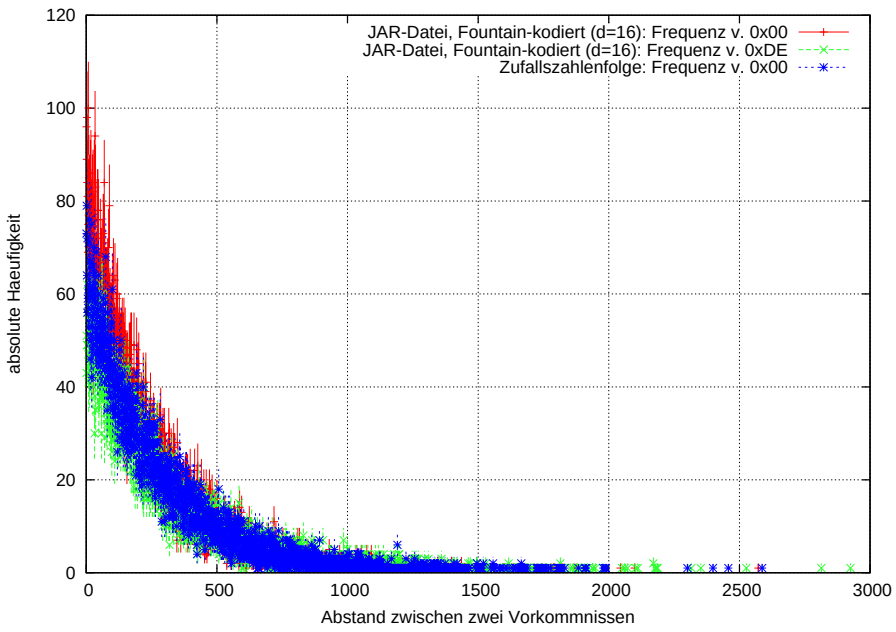

Abbildung 4.15.: Frequentielle Verteilung der JAR-Datei, Fountain-kodiert mit $d = 16$

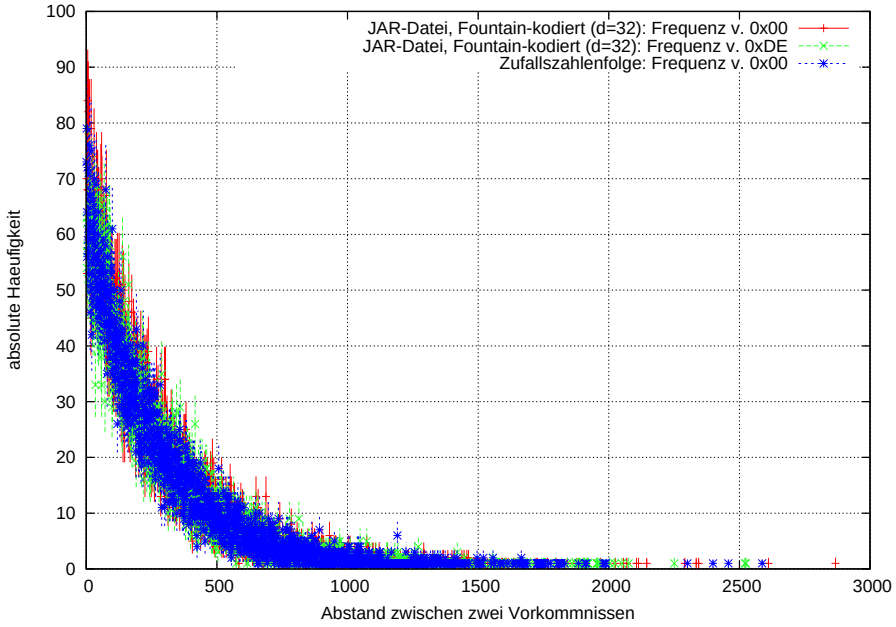

Abbildung 4.16.: Frequentielle Verteilung der JAR-Datei, Fountain-kodiert mit $d = 32$

4.10), in der die Byte-Zustände über Integer-Wert 127 kaum vorkommen. Entscheidend für die Zuordnung eines Bytes zu einem der Intervalle ist das höchstwertige Bit (Most Significant Bit, msb) eines Bytes. So ist das msb im Verhältnis zu den restlichen sieben Bit von vornherein relativ selten. Da die Klartextblöcke zur Kodierung zufällig ausgewählt werden, ist die Wahrscheinlichkeit, dass ein Byte mit $msb = 1$ ausgewählt wird, relativ gering. Es überwiegen die Bytes mit $msb = 0$. Es kommt folglich nur selten zur Verknüpfung eines Bytes mit $msb = 1$ und einem mit $msb = 0$, so dass beim resultierenden Byte ebenfalls $msb = 1$ vorliegt. Die Häufigkeitsverteilung der Intervalle gleichen sich somit nur langsam an. Es entsteht eine deutliche, bis zu einem hohen Degree anhaltene Stufe bei dem Integer-Werten 127 und 128.
Ab einem Degree von 45 ist schließlich auch die Häufigkeitsverteilung weitestgehend ausgeglichen. In Abbildung 4.18 ist noch ein kleiner Unterschied zwischen den Intervallen zu sehen, der aber auf Grund seiner geringen Größe nicht weiter betrachtet wird. Mit einem Degree 45 lassen sich die Strukturen soweit verbergen, dass sie mit den hier aufgezeigten Experimenten nicht mehr sichtbar gemacht werden können.
Die Strukturen des Java-Bytecodes lassen sich also deutlich schwieriger obfuskieren, als es im MPEG-2-Format der Fall ist. Die Strukturen beruhen hauptsächlich auf dem Ungleichgewicht der Zustände 0 bis 127 zu den Zuständen 128 bis 255. So sind alle anderen Strukturen bei einem Degree von 16 im Wesentlichen verschwunden. Das Ungleichgewicht zwischen den beiden Intervallen bleibt jedoch bis zu einem Degree von 45 bestehen. Es wird deutlich, dass eine gewisse Initial-Verteilung in den Daten vorliegen muss, damit sich nach der Kodie-

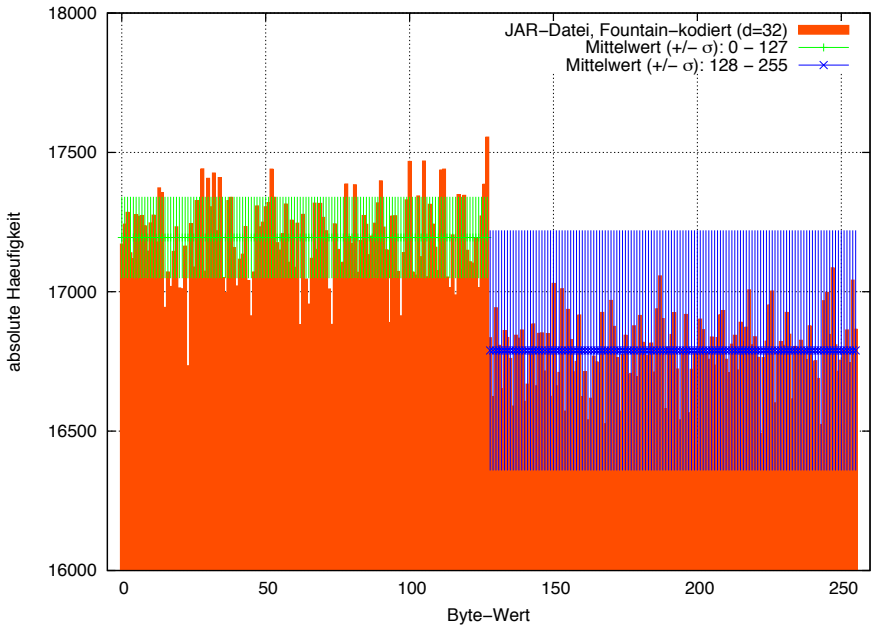

Abbildung 4.17.: Häufigkeitsverteilung der JAR-Datei, Fountain-kodiert mit $d = 32$: Unterschiede der Mittelwerte

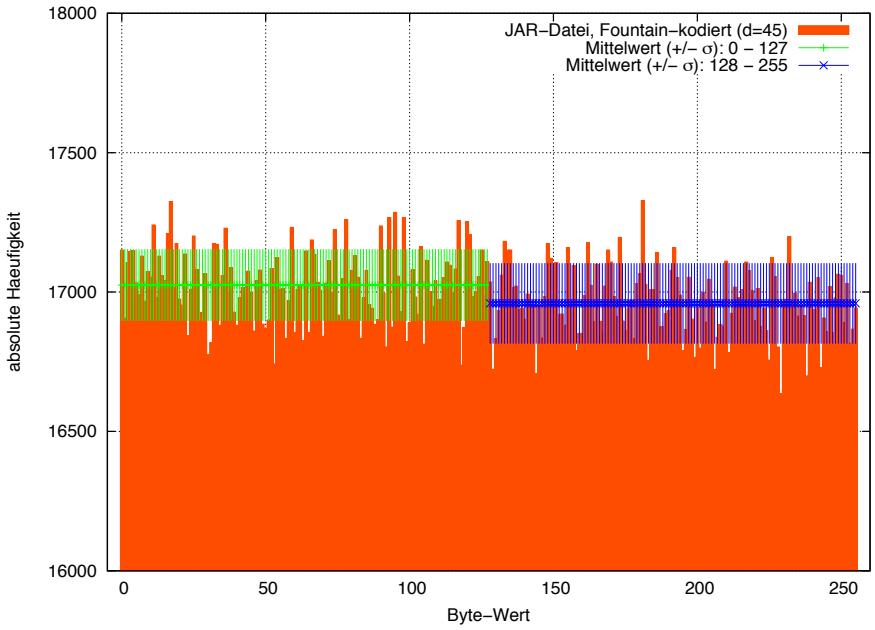

Abbildung 4.18.: Häufigkeitsverteilung der JAR-Datei, Fountain-kodiert mit $d = 45$: Unterschiede der Mittelwerte

DEGREE	BITS 0/1 (%)	σ	FREQ. 0X00	FREQ. 0XDE	ENTROPIE
$d = 1$	65.95/34.05	64432.68	4.52	5380.05	5.873492
$d = 8$	52.09/47.91	5670.66	188.63	383.12	7.918126
$d = 16$	50.68/49.32	1865.53	232.58	285.9	7.991287
$d = 32$	50.05/49.95	244.06	253.33	263.75	7.999851
$d = 45$	50.03/49.97	129.76	253.62	257.72	7.999958
$Zufallsfolge$	50/50	135.47	261.17	256.5	7.999954

Tabelle 4.5.: Resultate der Experimente mit Testdatei 2

rung eine zufällig erscheinende Verteilung ergibt. Ist beispielsweise das msb in keinem Byte einer Datei gesetzt, wird es durch die Ausgabe des $\oplus$-Operators (s. Abs. 3.4.2) in keinem kodierten Paket den Wert 1 haben.
Um eine implizite Vertraulichkeit, wie sie im MPEG-2-Format umsetzbar ist, zu gewährleisten, müssten, neben dem Koeffizientenvektor auch alle kodierten Pakete bis zu einem Degree von mindestens 45 zusätzlich verschlüsselt werden. Die Höhe des Degrees, um unter Verwendung des JAR-Formats die gleiche implizite Vertraulichkeit zu erreichen, wie im MPEG-2-Format möglich ist, steht somit nicht im Verhältnis zum Nutzen. Somit bietet das JAR-Format ohne zusätzliche Verfahren kaum ein Maß an impliziter Vertraulichkeit.

4.3.4. Experiment mit Kompression

In diesem Experiment werden die Klartextdaten vor der Untersuchung mit GZIP komprimiert. Die kodierten Dateien werden anhand der komprimierten Klartextdaten erstellt. Die Ergebnisse sind in Tabelle 4.7 und 4.6 zusammengefasst.
Die JAR-Datei ließ sich hierbei auf 33.46% ihrer ursprünglichen Größe von 4349143 auf 1455091 Bytes komprimieren. Betrachtet man die Häufigkeitsverteilung dieser Datei (Abb. 4.19) und vergleicht sie mit Abbildung 4.10, werden die Effekte der Kompression deutlich. Die Verteilung der Bytes erstreckt sich nach Kompression fast gleichmäßig über den gesamten Wertebereich der interpretierbaren Zustände eines Bytes. Die besonders markanten Muster, die in Abbildung 4.10 noch deutlich waren, sind verschwunden. Es erscheint allerdings eine neue Struktur: Das Byte $0x00$ ist nun gegenüber allen anderen relativ selten. Mit einem Anteil von nur 0.296% der Datei ist es das Zeichen mit der geringsten Häufigkeit. So liegt die Standardabweichung der mittleren Häufigkeit bei ca. 260 Zeichen (0.01787% der gesamten Datei). Die Zufallszahlenfolge hat mit $\sigma = 65.95$ Zeichen (0.0045%) einen etwas kleineren Wert. Des Weiteren sind in Abbildung 4.19 noch kleinere strukturelle Effekte zu sehen, durch die sich die komprimierte JAR-Datei noch von der Zufallszahlenfolge unterscheidet. Die Entropie nach der Kompression beträgt 7.998467 Bits pro Byte (vgl. nicht komprimiert: $H = 5.873492$). Die entsprechend große Zufallszahlenfolge hat eine Entropie

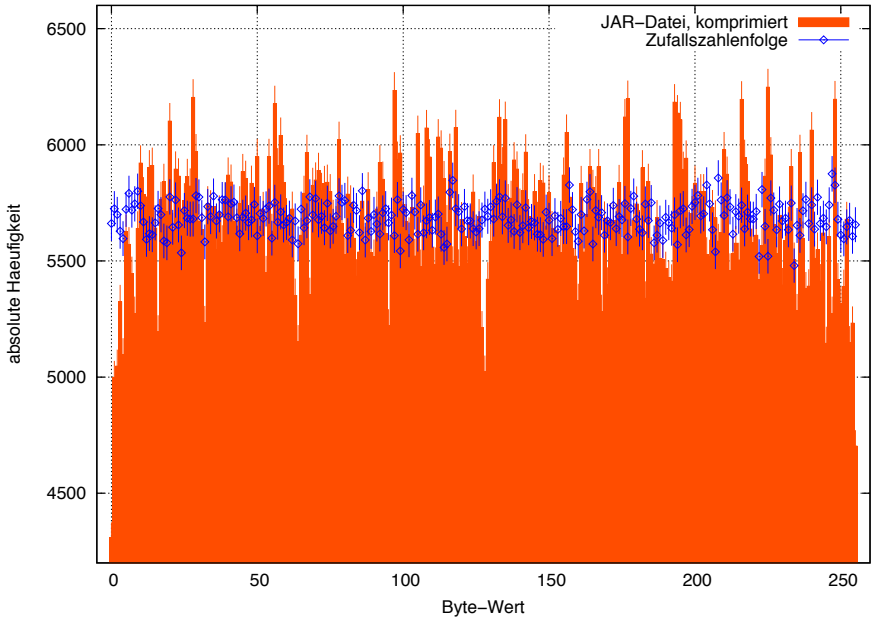

Abbildung 4.19.: Häufigkeitsverteilung der JAR-Datei, komprimiert (GZIP)

von 7.999903 Bits pro Byte.
Beim Abstandstest (Abb. 4.20) werden diese Unterschiede auch deutlich. Die Zufallszahlenfolge ist hier recht ausgeglichen. Das Byte $0x00$ der komprimierten JAR-Datei hat jedoch noch einige Ausreißer auf der X-Achse, während das Byte $0xDE$ noch relativ viele geringe Abstände, also Ausreißer auf der Y-Achse, aufweist.
Wird die komprimierte JAR-Datei mit einem Degree von 2 kodiert, glättet sich die Verteilung. Es fällt allerdings ein Peak beim Integer-Wert 255 auf. Die Standardabweichung der mittleren Häufigkeit ist mit 75.34 Zeichen (0.00516% der gesamt Datei) der Zufallszahlenfolge schon recht ähnlich. Die Entropie ist mit 7.999874 Bits pro Byte der des Erwartungswerts auch schon sehr ähnlich.
Erhöht man das Degree auf drei, so lassen sich bei der Standardabweichung und der Entropie keine messbaren Unterschiede mehr feststellen. Der Peak, welcher bei einem Degree von 2 noch sichtbar war, ist allerdings verschwunden (Abb. 4.22). Hier ist aufgrund von zufälligen Schwankungen keine Aussage über bestehende Unterschiede zu der Zufallszahlenfolge mehr möglich. Auch beim Abstandstest (Abb. 4.23) zeigt sich bei einem Degree von 3, dass keine Strukturen mehr zu sehen sind. Nach der Kompression der JAR-Datei ist somit schon mit einem sehr geringen Degree ein sehr gutes Ergebnis möglich. Bei einem Degree von 2 kann es noch gewisse strukturelle Erscheinungen geben, ab einem Degree von 3 sind aber alle Strukturen weitestgehend obfuskiert.
Bei der MPEG-2-Datei sind ähnliche Resultate zu beobachten. Sie ließ sich auf 95.56% der ursprünglichen Größe von 70000000 auf 66889731 Bytes komprimieren. In Abbildung 4.24

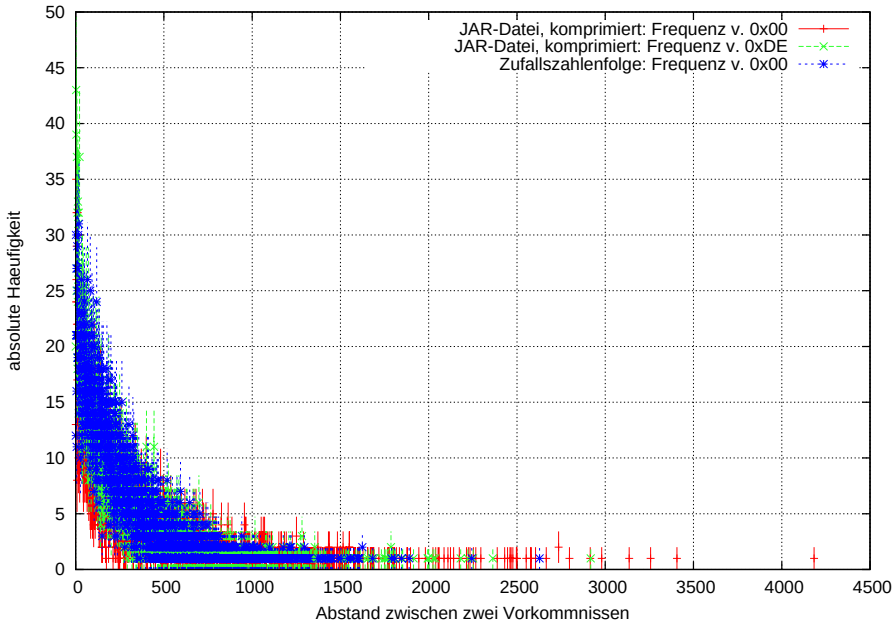

Abbildung 4.20.: Frequentielle Verteilung der JAR-Datei, komprimiert

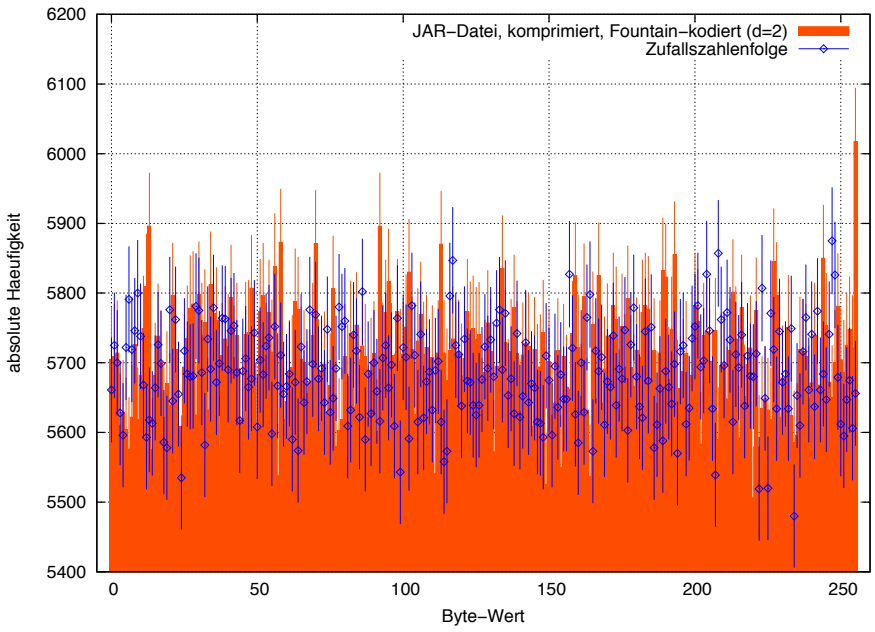

Abbildung 4.21.: Häufigkeitsverteilung der JAR-Datei, komprimiert, Fountain-kodiert mit $d = 2$

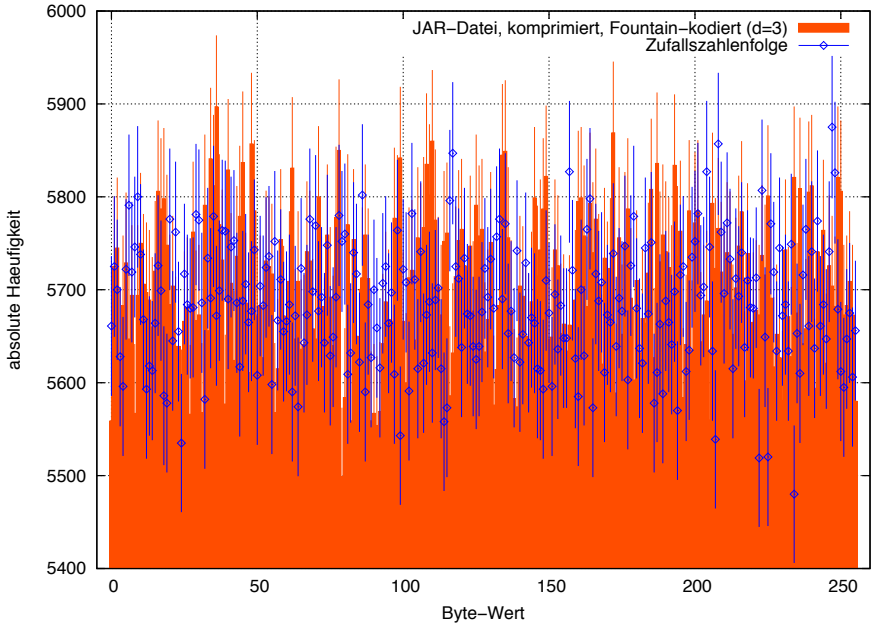

Abbildung 4.22.: Häufigkeitsverteilung der JAR-Datei, komprimiert, Fountain-kodiert mit $d = 3$

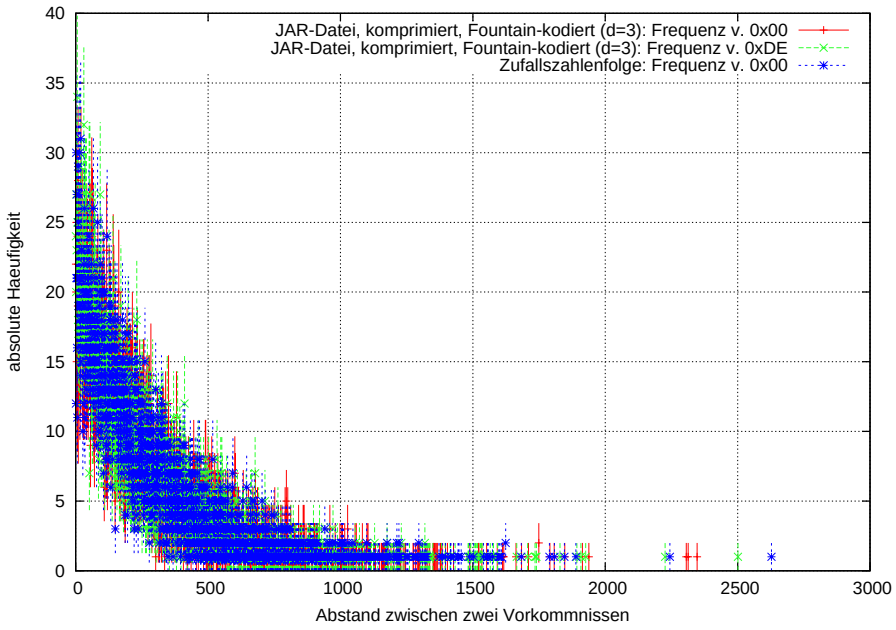

Abbildung 4.23.: Frequentielle Verteilung der JAR-Datei, komprimiert, Fountain-kodiert mit $d = 3$

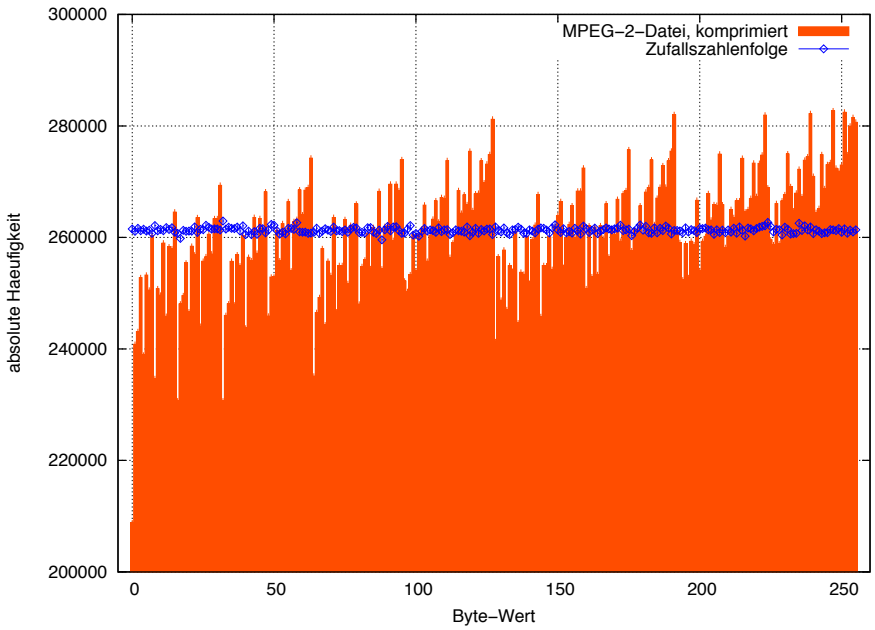

Abbildung 4.24.: Häufigkeitsverteilung der MPEG-2-Datei, komprimiert

DEGREE	BITS 0/1 (%)	σ	FREQ. 0X00	FREQ. 0XDE	ENTROPIE
$d = 1$	50.08/49.92	260	337.69	268.38	7.998467
$d = 2$	49.99/50.01	75.34	255.91	261.67	7.999874
$d = 3$	50.01/49.99	75.5	262.63	257.19	7.999874
$Zufallsfolge$	50.01/49.99	65.95	256.94	263.65	7.999903

Tabelle 4.6.: Resultate der Experimente mit Testdatei 2, komprimiert

kann man den gleichen Effekt beobachten wie zuvor in der JAR-Datei: Das Zeichen $0x00$ kommt mit 0.31% der gesamten Datei am seltensten vor. Die Standardabweichung beträgt hier 9981.42 Zeichen (0.0149% der Datei). Bei der Zufallszahlenfolge beträgt die Standardabweichung 492.85 Zeichen (0.000736% der Datei). Im Vergleich zur nicht-kodierten Datei (Abb. 4.1) sind die Häufigkeiten nach der Kompression denen der Zufallszahlenfolge (Abb. 4.24, blau) etwas ähnlicher. Es sind zwar noch starke Strukturen zu erkennen, die größten Peaks wurden jedoch schon durch die Kompression obfuskiert. Die Entropie beträgt 7.998935 (vgl. nicht komprimiert: $H = 7.861921$). Die gleichgroße Zufallszahlenfolge hat eine Entropie von 7.999997. Die Entropie der komprimierten MPEG-2-Datei ist also um 0.001062 geringer.
Schon ab einer Kodierung mit $d = 2$ können die Strukturen kaum noch sichtbar gemacht werden. Die Werte bewegen sich auf einem Niveau, das ohne Kompression erst ab einem Degree von 4 erreicht werden konnte. Die Standardabweichung der mittleren Häufigkeit be-

DEGREE	BITS 0/1 (%)	σ	FREQ. 0X00	FREQ. 0XDE	ENTROPIE
$d = 1$	49.36/50.64	9981.42	320.26	243.44	7.998935
$d = 2$	50.01/49.99	507.71	255.34	255.51	7.999997
$Zufallsfolge$	50/50	492.85	255.78	255.32	7.999997

Tabelle 4.7.: Resultate der Experimente mit Testdatei 1, komprimiert

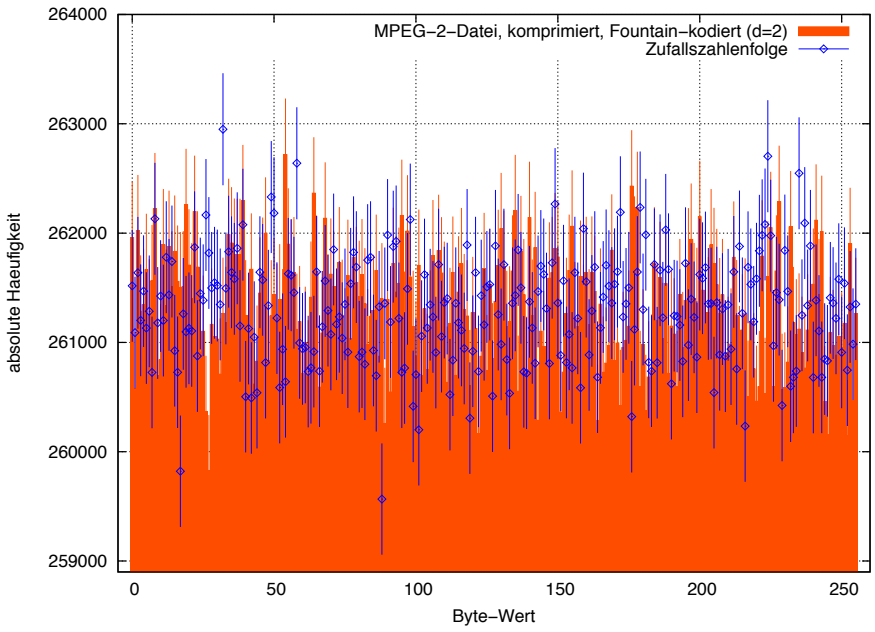

Abbildung 4.25.: Häufigkeitsverteilung der MPEG-2-Datei, komprimiert, Fountain-kodiert mit $d = 2$

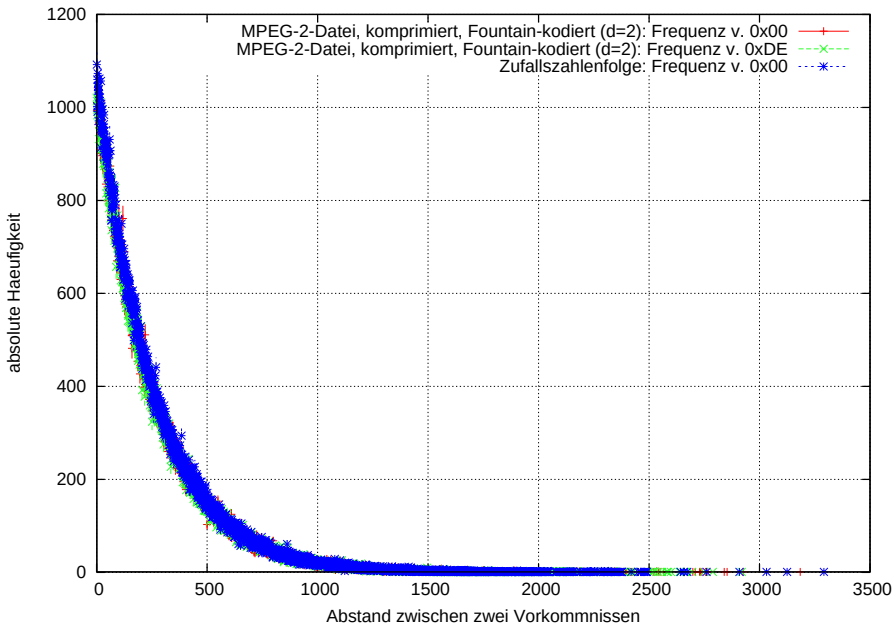

Abbildung 4.26.: Frequentielle Verteilung der MPEG-2-Datei, komprimiert, Fountain-kodiert mit $d = 2$

DEGREE	σ_{min}	σ_{avg}	σ_{max}
$d = 2$	6.15	7.21	8.48
$d = 3$	6.09	7.21	8.46
$Zufallsfolge$	6.02	7.22	8.56

Tabelle 4.8.: Resultate der Experimente mit kodierten Paketen der Testdatei 1, komprimiert

trägt hier noch 507.71 (0.000759% der Datei). Die Entropie steigt auf 7.999997, womit kein messbarer Unterschied mehr besteht. Die noch erkennbaren Unterschiede zu der Zufallszahlenfolge sind im Rahmen der erwarteten statistischen Unsicherheit.
Jetzt wird noch einmal der Frage nachgegangen, ob ein Angreifer ein kodiertes Paket mit einem Degree von 2 als solches entlarven kann. Dieses Mal wird die MPEG-2-Datei vor der Kodierung komprimiert. Hierbei ist jeder Block jetzt noch 13378 Bytes groß. In Abbildung 4.27 und Tabelle 4.8 wird deutlich, dass sich die einzelnen Pakete kaum noch voneinander unterscheiden. Die $d = 2$ Pakete (Abb. 4.27, rot) sind mit denen der Zufallszahlenfolge fast identisch. Für den Angreifer ist es also deutlich schwieriger, ein Paket mit $d = 2$ von solchen mit höherem Degree zu unterscheiden. Da sich nun anhand des Standardabweichung kein hilfreicher Bezugswert festlegen lässt, hat der Angreifer nur noch die Möglichkeit, ein $d = 2$-Paket zu erraten. Die Erfolgsaussichten eines solchen Brute-Force-Angriffs sollen an dieser Stelle nicht untersucht werden.

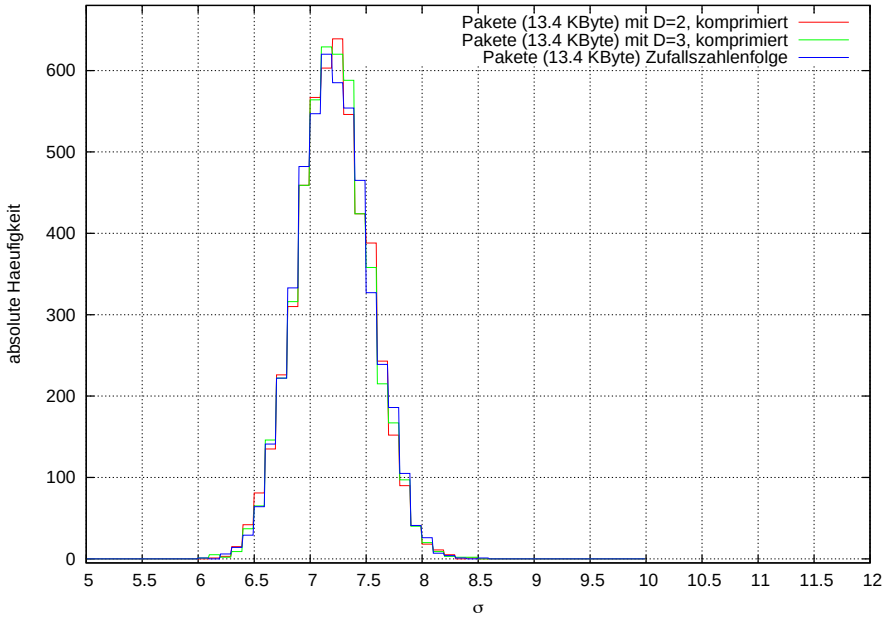

Abbildung 4.27.: Verteilung der Häufigkeitsabweichungen kodierter Pakete (MPEG-2, komprimiert) mit $d = 2$ und $d = 3$

Mit der zusätzlichen Kompression der Daten ist es also möglich, unabhängig von dem Format ein höheres Maß an impliziter Vertraulichkeit zu erlangen als ohne Kompression. Durch die hier angestellten Experimente können so keine auffälligen Strukturen mehr sichtbar gemacht werden. Somit bietet sich für einen Angreifer nach der hier vorgestellten strukturellen Analyse nur noch die Möglichkeit eines Brute-Force-Angriffs.

5. Fazit

5.1. Abschließende Bewertung

In dieser Arbeit wurde untersucht, inwieweit sich die Effekte der Fountain-Kodierung nutzen lassen, um implizit das Schutzziel „Vertraulichkeit“ umzusetzen. Hierbei sollten zusätzliche kryptographische Verfahren ausschließlich für die Sicherung des Koeffizientenvektors und Pakete mit nicht ausreichend hohem Degree genutzt werden. Wie hoch das Degree sein muss, um eine genügende Obfuskierung zu gewährleisten, war hierbei eine der Fragestellungen. In den Experimenten ergab sich diesbezüglich eine starke Abhängigkeit von verschiedenen Parametern. Hierzu zählen das verwendete Dateiformat, der Inhalt der Daten, das Degree und die Degree Distribution.
Zuerst wurden die Experimente mit nicht-komprimierten Dateien durchgeführt. In dem MPEG-2-Format ließen sich die Strukturen mit einem Degree von 4 so gut obfuskieren, dass sie mit den angestellten Analysen nicht sichtbar gemacht werden konnten. Beim JAR-Format war ein weitaus höheres Degree nötig, um denselben Grad an Obfuskierung zu erhalten. Hier waren die Strukturen erst ab einem Degree von 45 nicht mehr sichtbar. Insbesondere die Unterschiede der Intervalle standen hervor. Es wurde deutlich, dass eine Initial-Verteilung in den Daten nötig ist, um die Strukturen durch die $\oplus$-Operation zu verbergen. Durch das seltene Vorkommen des höchstwertigen Bits in der JAR-Datei ergab sich eine sehr auffällige Stufe zwischen den Intervallen. Damit ist nicht jedes Format gleichermaßen für eine implizite Vertraulichkeitserweiterung geeignet. Je nach Ausprägung der internen Strukturen der Daten ist die Daten-Obfuskierung durch Fountain Codes stärker oder schwächer. Beim dem Experiment mit den Paketen des MPEG-2-Formats stellte sich zusätzlich das Problem, dass ein Angreifer mit einer hohen Wahrscheinlichkeit ein $d = 2$ Paket isoliert und somit einen Angriff auf das System erfolgreich durchführt. In den Experimenten konnte gezeigt werden, dass ein Angriff, wie er auf WEP ausgeübt wurde, nicht ausgeschlossen werden kann und als sehr wahrscheinlich angenommen werden muss.
Erst nach der Kompression der Daten wurde die Abhängigkeit vom Degree und dem Dateiformat stark verringert. So wies die komprimierte JAR-Datei schon ab einem Degree von 3 keine sichtbaren Strukturen mehr auf. Beim MPEG-Format waren die Strukturen sogar ab einem Degree von 2 vollständig verschleiert. Hierbei wurde deutlich, dass die Verringerung der Redundanz ein Schlüssel für den Erfolg für die in dieser Arbeit zugrunde gelegten Problemstellung ist.

Das Maß an impliziter Vertraulichkeit, welches allein durch die Kodierung durch Fountain Codes ohne Kompression erreicht wird, kann also nur als sehr gering eingestuft werden. Durch die Verwendung zusätzlicher Kompression lassen sich die Ergebnisse deutlich optimieren und somit auch das hierauf beruhende Vertraulichkeitsniveau erhöhen. Es kann nach den hier angestellten Experimenten allerdings nicht von einer absoluten Vertraulichkeit ausgegangen werden. Damit ist das Verfahren für eine Anwendung, die nicht auf eine extrem hohe Vertraulichkeitsstufe angewiesen ist, ein durchaus denkbarer Ansatz. Ist ein höheres Maß an Vertraulichkeit gefordert, wird an dieser Stelle dazu geraten, bei der Sicherung von Fountain Codes weiterhin standardisierte kryptographische Verfahren zu nutzen. Die Umsetzung impliziter Vertraulichkeit durch Fountain Codes stellt aber einen interessanten Ansatz dar, welcher in Zukunft weiter verfolgt werden sollte. Einige Anregungen hierzu werden im nächsten Abschnitt gegeben.

5.2. Ausblick und Verbesserungen für weitere Experimente

Bei der Durchführung der Experimente wurden nicht alle Parameter und Bedingungen, die einem Angreifer zur Verfügung stehen, berücksichtigt. So wurden hauptsächlich verschiedene Degrees, aber keine Verteilung einer konkreten Degree Distribution untersucht. Diese spielt in einer praktischen Umsetzung eine wesentliche Rolle. Es muss geprüft werden, wie weit sich das Vertraulichkeitsniveau durch die geschickte Wahl einer auf das genutzte Format angepassten Distribution erhöhen lässt. Für verschiedene Anwendungsfälle kann dieses eine besondere Rolle spielen. So nutzt das in [8] beschriebene „Concealed Network Coding for Multicast Traffic“ eine Technik, in der sehr hohe Degrees entstehen. Hierbei leiten die Knoten des Netzwerkes einige kodierte Pakete weiter, nachdem sie die Kodierfunktion wiederholt auf das kodierte Paket angewandt haben, ohne es vorher zu dekodieren. Durch eine homomorphe Verschlüsselungsfunktion kann der Koeffizientenvektor auf das resultierende kodierte Paket angepasst werden. Durch die resultierenden Pakete mit hohen Degrees könnten die Schwierigkeiten der Obfuskierung des JAR-Formats minimiert werden. An dieser Stelle sind noch genauere und detaillierte Auswertungen nötig, um die Abhängigkeit der Formate durch hohe Degrees bei „Concealed Network Coding“ (CNC) zu beurteilen.
In den verschiedenen Formaten können noch weitere Untersuchungen in Bezug auf die interne Struktur angestellt werden. Das MPEG-2-Format schien hierbei relativ gut geeignet. In Bezug auf den Inhalt der Testdatei ist interessant, inwieweit sich Veränderungen im Bildmaterial auswirken. So könnten sich zum Beispiel Musikvideos, mit vielen wechselnden Bildern schlechter eignen als das statische Bild einer Überwachungskamera. Auch die Lichtverhältnisse des Videos sind hierbei zu untersuchen. Aber nicht nur der Inhalt, sondern auch die Stärke der MPEG-Kompression, also die Qualität des Videos und somit die in den Daten

enthaltene Redundanz, können sich auf das Vertraulichkeitsniveau stark auswirken. Diese Faktoren können wesentliche Gründe für strukturelle Informationen im MPEG-2-Format sein, wurden aber in dieser Arbeit nicht weiter berücksichtigt. An dieser Stelle sollten auch weitere Bezugswerte und Testfunktionen herangezogen werden, um eine feinere Differenzierung von Testdatei zu Referenzdatei zu erlauben.
Auch bei Java-Bytecode gibt es weitere Faktoren, die in dieser Arbeit nicht genauer betrachtet wurden. Kompiliert man Java-Quellcode, lassen sich zum Beispiel optionale Debug-Informationen erstellen (Option: g und g:none). Diese können zusätzliche Strukturen darstellen. Für weiterführende Untersuchungen müsste diese Option mit betrachtet werden, um so die Ergebnisse des JAR-Formats zu optimieren. Des Weiteren könnte auch kompilierter Quellcode anderer Sprachen untersucht werden, um den Anwendungsfall des Code-Updates noch genauer zu beleuchten.
Ein weiterer Parameter, der in dieser Arbeit nicht betrachtet wurde, ist die Größe der kodierten Pakete. Für die Funktionalität der Fountain-Kodierung spielt sie zwar keine Rolle [16], für die Verteilung der Häufigkeiten dafür um so mehr. Hierfür muss analysiert werden, ob die Größe der Pakete die Erfolgswahrscheinlichkeit eines Angriffs beeinflusst. Da der statistische Fehler bei kleineren Paketen stärker ins Gewicht fällt, könnte es einem Angreifer mit abnehmender Größe der Pakete immer schwerer fallen, kodierte Pakete von einer Zufallszahlenfolge zu unterscheiden. In dieser Arbeit wurden hauptsächlich Untersuchungen der ganzen Datenströme getätigt, so dass eine weitere Analyse der einzelnen Pakete dabei helfen könnte, das Vertraulichkeitsniveau besser zu bewerten.
In dieser Arbeit konnte gezeigt werden, dass die Kodierung mit Fountain Codes für sich genommen nur eine sehr schwache implizite Vertraulichkeitserweiterung bildet. Im Zusammenhang mit der Kompression der Daten hingegen könnte dieses Verfahren für einige Anwendungen ein sehr hilfreicher Ansatz sein. Das hierdurch erreichte Vertraulichkeitsniveau wird allerdings nie mit standardisierten Verfahren standhalten können. Auch wenn Fountain Codes somit in puncto Sicherheit weiter auf kryptographische Verfahren angewiesen sind, wird man auch hierdurch nie gänzlich immun gegen Angriffe sein. So sagte einst Seth Lloyd treffend [28]:

> *«Verrat wirkt immer»*

A. Anhang

Der Anhang befindet sich auf der beiliegenden CD. Diese CD umfasst folgende Inhalte:

1. Datensätze der Abbildungen (./Plot/)
2. Projekt: Fountain Codes: Spezifikationsdokument der Übertragungsschicht (./Projekt_FC/)
3. Quellcode der Test-Funktionen (./Quellcode/)

Literaturverzeichnis

[1] J. Bohli, A. Hessler, O. Ugus, and D. Westhoff. Security Enhanced Multi-Hop over the Air Reprogramming with Fountain Codes. In *International Workshop on Practical Issues In Building Sensor Network Applications*, Oct 2009.

[2] N. Challa and J. Pradhan. Performance Analysis of Public key Cryptographic Systems RSA and NTRU. *IJCSNS International Journal of Computer Science and Network Security*, 7:87–96, August 2007.

[3] Digital Fountain, Inc. http://www.digitalfountain.com/. Webseite. Abruf: 27.05.2010, 16:57 Uhr.

[4] W. Dörfler. *Mathematik für Informatiker, Band 2*. Carl Hanser Verlag, München, 1978.

[5] C. Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Oldenbourg, München, 2009.

[6] S. Fluhrer, I. Mantin, and A. Shamir. *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, chapter Weaknesses in the Key Scheduling Algorithm of RC4, pages 1–24. Springer, Berlin, 2001.

[7] A. Hessler, T. Kakumaro, D. Westhoff, and H. Perrey. On the Implicit Privacy of Concealed Network Coding. *under review to Elsevier Computer Communications*, 2010.

[8] A. Hessler, T. Kakumaru, and D. Westhoff. When Eco-IT meets Security: Concealed Network Coding for Multicast Traffic. In *International Conference on Pervasive Computing and Communications*, Mar, Apr 2010.

[9] M. Hoffmann and E. Boos. *Großes Handbuch Mathematik: Formeln, Regeln, Merksätze.* Compact Verlag, München, 2005.

[10] T. Iwao, S. Amamiya, G. Zhong, and M. Amamiya. Ubiquitous computing with service adaptation using peer-to-peer communication framework. In *Distributed Computing Systems, 2003. FTDCS 2003. Proceedings. The Ninth IEEE Workshop on Future Trends of*, pages 240 – 248, 28-30 2003.

[11] S. Kapitza. Implementierung einer Kryptanalyse in digitaler Hardware zur quantitativen Verifikation des Geschwindigkeitsvorteils gegenüber softwareseitigen Lösungen. Diplomarbeit, Hochschule für Angewandte Wissenschaften Hamburg, Hamburg, 2010.

[12] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *J. Comput. Secur.*, 8(2,3):141–158, 2000.

[13] A. Klein. Attacks on the RC4 stream cipher. *submitted to Designs, Codes and Cryptography*, July 2007.

[14] J. Kurose and K. Ross. *Computernetze. Ein Top-Down-Ansatz mit Schwerpunkt Internet*. Pearson Studium, München, 2002.

[15] Oliver Lau. Faites vos jeux! Zufallszahlen erzeugen, erkennen und anwenden. *c't*, 2:172–179, 2009.

[16] M. Luby. LT codes. *Foundations of Computer Science*, 43:271–280, November 2002.

[17] L. Müller. Interactive Design - Studien der interdisziplinären Zusammenarbeit von Design und Informatik. Bachelorarbeit, Hochschule für Angewandte Wissenschaften Hamburg, Hamburg, 2010.

[18] L. Papula. *Mathematik für Ingenieure und Naturwissenschaftler, Band 3*. Vieweg Verlag, Braunschweig/Wiesbaden, 2001.

[19] R. Parameswaran. *A robust data obfuscation approach for privacy preserving collaborative filtering*. PhD thesis, School of Electrical and Computer Engineering Georgia Institute of Technology, Atlanta, GA, USA, 2006.

[20] W. Polk, R. Hously, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *Request for Comments: 3280*, April 2002.

[21] B. Röthlein. *Die Quantenrevolution. Neue Nachrichten aus der Teilchenphysik*. Deutscher Taschenbuch-Verlag, München, 2004.

[22] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, New York, 2004.

[23] P.A. Sarginson. MPEG-2: A tutorial introduction to the systems layer. In *MPEG-2 - What it is and What it isn't*, pages 4/1 –413, 24 1995.

[24] B. Schneier. *Angewandte Kryptographie*. Addison-Wesley, 1996.

[25] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Performance Comparison of the AES Submissions. In *Proc. Second AES Candidate Conference*, pages 15–34, March 1999.

[26] Amin Shokrollahi. Raptor Codes. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 52:2552–2567, June 2006.

[27] J. Stiewe. Wir wollen richtige Fehler ! Ein Statistik - Feuilleton zur Lektüre am Feierabend. *Kirchhoff - Institut für Physik, Universität Heidelberg*, October 2002.

[28] G. Stix. Datenschutz mit Quantenschlüsseln. *Spektrum der Wissenschaft*, pages 68–73, May 2005.

[29] E. Tews, R. Weinmann, and A. Pyshkin. Breaking 104 bit wep in less than 60 seconds. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *WISA*, volume 4867 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2007.

[30] W. Trappe and L. Washington. *Introduction to Cryptograhy with Coding Theory*. Pearson Education, Inc., 2002.

[31] http://www.fourmilab.ch/. Webseite. Abruf: 21.05.2010, 13:00 Uhr.

[32] http://openbook.galileocomputing.de/javainsel8/. Webseite. Abruf: 12.07.2010, 12:15 Uhr.

[33] http://www.lucike.info/page_projectx.htm. Webseite. Abruf: 15.07.2010, 13:00 Uhr.

[34] http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/. Webseite. Abruf: 12.07.2010, 12:15 Uhr.

[35] http://www.fr-an.de/. Webseite. Abruf: 21.05.2010, 14:00 Uhr.

[36] http://www.cacert.org/. Webseite. Abruf: 23.05.2010, 12:00 Uhr.

[37] http://mpeg.chiariglione.org/. Webseite. Abruf: 23.05.2010, 13:00 Uhr.

[38] http://dvd.sourceforge.net/. Webseite. Abruf: 23.05.2010, 13:00 Uhr.

[39] http://www.mpgedit.org/. Webseite. Abruf: 23.05.2010, 13:00 Uhr.

[40] http://www.aero.org/publications/crosslink/winter2002/04.html. Webseite. Abruf: 27.05.2010, 12:13 Uhr.

Glossar

Benchmark Als Benchmark wird der Wert bezeichnet, der in einem Experiment als Bezugswert dient. Er hilft hierbei, die Ergebnisse der anderen Testdaten besser einordnen und bewerten zu können.

Code Rate Die Code Rate der Fountain Codes beschreibt das Verhältnis aus Blöcken, in die eine Datei unterteilt ist, und der Anzahl kodierter Symbole, die während des Kodierprozesses erstellt werden.

GZIP GZIP ist ein Programm zur verlustfreien Kompression von Daten, welches den Deflate-Algorithmus implementiert [22].

homomorphe Verschlüsselung Sind Daten mit einem homomorphen Verfahren verschlüsselt, führt die Änderung eines Cipher-Bits bei Entschlüsselung nicht zu einer Änderung anderer Klartext-Bits. Der Ciphertext kann also verändert werden, ohne dass dies Einfluss auf die benachbarten Daten derselben Datei hat.

Most Significant Bit, msb Das Most Significant Bit ist das höchstwertige Bit eines Bytes. Bsp.: *1000 0000b*: hier $msb = 1$.

One-To-Many-Kommunikation beschreibt eine Kommunikation, in der ein Sender eine oder mehrere Nachrichten an viele Empfänger sendet (Broadcast/Unicast).

Rückkanal Von einem Rückkanal wird gesprochen, wenn in einem Client-Server-System eine Verbindung vom Client zum Server besteht, in der noch nicht erhaltene Pakete nachgefordert werden.

WEP Wired Equivalent Privacy (WEP) ist ehemaliger Standard-Verschlüsselungsalgorithmus für IEEE 802.11 (WLAN) Netzwerke.

WSN Ein Wireless Sensor Network ist ein Netzwerk aus Sensoren, die über ein kabelloses Medium miteinander kommunizieren. Hierbei können sie durch entsprechende Sensoren zum Beispiel ihre Umgebung untersuchen und interpretieren.

Erklärung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw. §24(4) bzw. §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 10.08.2010	
Ort, Datum	Unterschrift