Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Viktor Winkelmann

Entwicklung einer Anwendung zur automatisierten forensischen Beweismittelextraktion aus Netzwerkverkehrsdaten

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Viktor Winkelmann

Entwicklung einer Anwendung zur automatisierten forensischen Beweismittelextraktion aus Netzwerkverkehrsdaten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus-Peter Kossakowski
Zweitgutachter: Prof. Dr. Birgit Wendholt

Eingereicht am: 3. März 2016

Viktor Winkelmann

Thema der Arbeit
Entwicklung einer Anwendung zur automatisierten forensischen Beweismittelextraktion aus Netzwerkverkehrsdaten

Stichworte
Forensik, Netzwerk, Datenanalyse, Beweismittelextraktion, Wireshark, PCAP, IT-Sicherheit, pcapfex

Kurzzusammenfassung
Netzwerkanalysetools wie z. B. Wireshark erlauben es dem Anwender den in einem Netzwerk fließenden Datenverkehr mitzuschneiden und einer forensischen Analyse zu unterziehen. Ist das zur Übertragung der Daten verwendete Protokoll jedoch nicht bekannt, bieten diese Tools keinerlei Unterstützung zum Auffinden verwertbarer Datenobjekte als Beweismittel, etwa von Bildern oder Zip-Archiven. Diese Bachelorarbeit umfasst den Entwurf und die Realisierung eines Tools, welches auch wenig erfahrenen Anwendern eine forensische Analyse ermöglicht. Des Weiteren wird untersucht, ob eine Klassifikation von Daten anhand ihrer Entropie Vorteile bei der Erkennung von Datenobjekten bieten kann.

Viktor Winkelmann

Title of the paper
Development of an application that automates extraction of forensic evidence from network traffic data

Keywords
forensics, network, data analysis, evidence extraction, Wireshark, PCAP, IT-Security, pcapfex

Abstract
Network analyzers like Wireshark allow the user to capture network traffic data and apply forensic analysis operations on them. Without knowledge about the protocols being used, these tools fail to assist in finding utilizable data objects as evidence, such as images or zip-archives. This bachelor thesis covers design and realization of a tool, that will assist even less experienced users in performing a forensic analysis. Furthermore, it determines the benefits of classifying data by it's entropy to find data objects.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Netzwerkanalysetools wie z. B. Wireshark erlauben es dem Anwender den in einem Netzwerk fließenden Datenverkehr mitzuschneiden und einer forensischen Analyse zu unterziehen. Ist das zur Übertragung der Daten verwendete Protokoll jedoch nicht bekannt, bieten diese Tools keinerlei Unterstützung zum Auffinden verwertbarer Datenobjekte als Beweismittel, etwa von Bildern oder Zip-Archiven. Diese Bachelorarbeit umfasst den Entwurf und die Realisierung eines Tools, welches auch wenig erfahrenen Anwendern eine forensische Analyse ermöglicht. Des Weiteren wird untersucht, ob eine Klassifikation von Daten anhand ihrer Entropie Vorteile bei der Erkennung von Datenobjekten bringen kann.

1.1 Motivation

Zu Zeiten des Internet of Things (IoT) wird der Grad der Vernetzung im Alltag immer größer. Stetig werden mehr Geräte mit Zugriff auf das Internet in privaten Haushalten in Betrieb genommen. Dabei ist es schwer zu sagen, welche Daten diese Geräte genau mit wem austauschen. Gerade bei besonders günstigen Geräten aus dem asiatischen Raum, z. B. bei Webcams, wird in Foren oft berichtet dass diese ungefragt Daten mit dem Hersteller austauschen. Umgangssprachlich spricht man hier vom “nach Hause telefonieren“. Da die Hersteller hierbei zumeist proprietäre Protokolle verwenden, ist es für wenig erfahrene Anwender schwierig die Inhalte solcher Daten zu finden und aufzudecken. Erst eine recht aufwendige manuelle Analyse des Netzwerkdatenverkehrs kann Aufschluss über die übertragenen Inhalte geben.

1.2 Zielsetzung

In dieser Bachelorarbeit soll ein Programm entwickelt werden, welches eine solche Analyse möglichst eigenständig durchführt. Es soll zuvor mitgeschnittene Netzwerkverkehrsdaten im PCAP-Format als Eingabe entgegen nehmen und als Ausgabe eine geordnete Struktur aller identifizierter Datenobjekte liefern. Die Zielgruppe des Programms sind in erster Linie Anwender, die bereits grundlegende Erfahrung im Mitschnitt von Netzwerkdatenverkehr mittels tcp-

dump oder Wireshark besitzen. Die Bedienung des Programms soll auf Kommandozeilenebene erfolgen.

1.3 Aufbau der Arbeit

Kapitel 1 *Einleitung* stellt eine Einführung in das Thema der Bachelorarbeit dar. In Kapitel 2 *Grundlagen* wird die theoretische Basis erläutert, auf der das Programm später aufbauen soll. Vertieft und auf den Anwendungsfall hin spezifiziert wird diese dann in Kapitel 3 *Anforderungsanalyse*. Zur Verifikation des Programms sowohl während als auch nach der Entwicklung werden in Kapitel 4 *Testszenarien* Beispielsituationen nachgestellt, die es ermöglichen sollen den Entwicklungsstand des Programms in Hinblick auf die gewünschte Funktionalität zu beurteilen. Im 5. Kapitel *Konzeption* wird ein umfassender Entwurf des späteren Programms erstellt und erläutert. Kapitel 6 *Implementierung* beschreibt schließlich anhand der wichtigsten Module die konkrete Umsetzung des Programms in Code. Abschließend wird in Kapitel 7 *Fazit & Ausblick* das umgesetzte Programm retrospektiv beurteilt und auf mögliche Verbesserungen eingegangen.

2 Grundlagen

Zunächst ist es erforderlich, ein Verständnis für die Grundlagen dieser Arbeit zu entwickeln. Sie bilden einerseits die Basis für die zu entwickelnde Anwendung, andererseits zeigen sie auf wo die fachlichen Schwerpunkte dieser Arbeit liegen.

2.1 Netzwerkforensik

Dr. Yong Guan von der Iowa State University definiert den Begriff der Netzwerkforensik als "[...] a science of discovering and retrieving evidential information in a networked environment about a crime in such a way as to make it admissible in court." [Vac13, S. 649] Die in dieser Arbeit behandelte Extraktion übertragener Daten als Beweismittel stellt somit eine Teildisziplin der Netzwerkforensik dar. Laut Guan [Vac13, S. 649f] müssen die Beweismittel folgende Eigenschaften erfüllen: Authentizität, Vollständigkeit, Zuverlässigkeit, Glaubhaftigkeit und Zulässigkeit. Die Zulässigkeit stellt im Kontext der forensischen Analyse eine der wichtigsten Eigenschaften dar. Sie erfordert Kenntnisse über die Methodik zur Gewinnung der Beweismittel, um die Standhaftigkeit vor Gericht sicherzustellen. Hierzu gehören die Zuverlässigkeit, mögliche Fehler und auch die allgemeine Anerkennung der Methodik. Teile dieser Arbeit, die einen Einfluss auf diese Aspekte haben, sind daher hervorzuheben.

2.2 PCAP-Dateien

Der Begriff Packet Capture (PCAP) ist auf das *libpcap*-Projekt [Sou15] zurückzuführen. Er beschreibt im Allgemeinen eine Programmierschnittstelle, die es erlaubt Netzwerkverkehrsdaten mitzuschneiden. Unter Unix-kompatiblen Betriebssystemen gilt *libpcap* als Standartimplementierung dieser Schnittstelle. Die mitgeschnittenen Daten werden von *libpcap* in einem proprietären Binärformat abgelegt, auch *PCAP-Datei* genannt. Die Eingabe von Netzwerkdaten in das zu entwickelnde Programm soll über PCAP-Dateien erfolgen. Einen Überblick über den Aufbau einer solchen Datei vermittelt das Wireshark Wiki [Har15]:

Dateiheader 24 Byte	Paketheader 1 16 Byte	Paketinhalt 1 N Byte	Paketheader 2 16 Byte	...

Abbildung 2.1: Aufbau einer PCAP-Datei

Auf einen globalen Dateiheader folgen beliebig viele Paare aus Paketheader und zugehörigem Inhalt. Der Dateiheader setzt sich dabei aus folgenden Feldern zusammen:

Signatur 4 Byte	Hauptversion 2 Byte	Unterversion 2 Byte	Zeitzone 4 Byte	...

...	Zeitgenauigkeit 4 Byte	Max. Paketlänge 4 Byte	Layer 2 Typ 4 Byte

Abbildung 2.2: Aufbau des PCAP-Dateiheaders

Die einzelnen Paketheader umfassen folgende Felder:

Zeitstempel s 4 Byte	Zeitstempel ms 4 Byte	Mitgeschnittene Paketlänge 4 Byte	Reale Paketlänge 4 Byte

Abbildung 2.3: Aufbau eines PCAP-Paketheaders

Einige der verwendeten Felder haben Einfluss auf die Zulässigkeit der extrahierten Daten als Beweismittel. Insbesondere denjenigen Feldern, die im Zusammenhang mit dem Zeitpunkt des Mitschnitts der Daten stehen, sollte besondere Aufmerksamkeit geschenkt werden. Bedingt durch die Drift der Uhr des aufzeichnenden Systems können hier Fehler entstehen. Auch ist zu beachten, das es sich bei den verwendeten Zeitstempeln um Unix-Zeitstempel handelt. Diese umfassen keine Schaltsekunden. Bei der Umrechnung zur aussagekräftigeren Universal Time Coordinated (UTC)-Zeit kann es daher zu weiteren Fehlern kommen. Es ist also ratsam, beim Mitschnitt der Netzwerkdaten einerseits ein System mit UTC-Empfänger zu benutzen, andererseits die vom Hersteller des UTC-Empfängers angegebene maximale Abweichung zur UTC-Zeit zusammen mit dem Mitschnitt zu notieren.
Auch die im Paketheader enthaltenen Felder zur Paketlänge haben einen Einfluss auf die Eigenschaften als Beweismittel. Ist die mitgeschnittene Länge kleiner als die reale Länge, z. B. weil beim Mitschnitt eine zu kleine maximale Paketlänge gewählt wurde, erfüllt das Paket nicht die für Beweismittel geforderte Vollständigkeit. Es sollte daher nicht zur Extraktion genutzt

werden. Dies wirkt sich auch auf den im Folgenden erklärten Prozess zum Aufbau von Streams aus. Im Falle eines TCP-Streams führt das Fehlen eines Pakets zur Ungültigkeit des gesamten Streams.

2.3 TCP/UDP-Streams

Die innerhalb einer PCAP-Datei mitgeschnittenen Pakete liegen stets atomar und in nicht deterministischer Reihenfolge vor. Wenn jedoch zwei Systeme über ein Netzwerk kommunizieren, tauschen diese in der Regel mehrere Pakete untereinander in Form von Datenströmen aus. Diese Datenströme bezeichnet man als *Streams*. Für eine beiderseitige Kommunikation werden zwei Streams erzeugt, deren Teildatenströme kausal voneinander abhängen. Man spricht hier auch von *kombinierten Streams*.
Das verwendete Protokoll hat einen Einfluss auf den Aufbau der Streams. Während bei UDP-Streams lediglich alle während einer Unterhaltung empfangenen gültigen Pakete in Reihenfolge des Empfangszeitpunkts den Stream bilden, muss für TCP-Streams einiges mehr an Vorarbeit geleistet werden. Im zugehörigen RFC 793 heißt es: "A stream of data sent on a TCP connection is delivered reliably and in order at the destination." [Pos81, S. 9] Hierfür ist zum Einen notwendig, übertragene Pakete auf Korrektheit zu untersuchen. Dies ist durch eine Überprüfung der in den Paketen enthalten Prüfsumme möglich. Zum Anderen müssen wegen eventuell aufgetretener TCP-Retransmits mehrfach empfangene Pakete herausgefiltert werden. Zuletzt müssen anhand der Sequenznummern der Pakete die korrekte Reihenfolge und die Vollständigkeit des Streams sichergestellt werden.

Es gilt zu beachten, dass es mehrere korrekte Möglichkeiten zur Implementierung dieser Anforderungen gibt. Insbesondere beim Herausfiltern mehrfach empfangener Pakete bleibt es dem Entwickler überlassen, welches der Pakete er behält. Dies eröffnet Angreifern Raum für Evasion-Angriffe. Behält beispielsweise ein Client stets das letzte mit der gleichen Sequenznummer empfangene Paket, während die Beweismittelextraktion das erste Paket behält, würde ein durch den Angreifer später gesendetes Paket mit veränderten Daten zwar beim Client verarbeitet werden, jedoch nicht in den extrahierten Daten enthalten sein. Die übliche Vorgehensweise zur Vermeidung einer solchen Sicherheitslücke ist das Verhalten des Clients bei der Beweismittelextraktion exakt nachzuahmen. Da sich jedoch die Verarbeitung der TCP-Streams in den TCP-Stacks der Clients unterscheiden kann, ist dies hier keine valide Option. Stattdessen könnte man vor dem Verwerfen eines doppelten Pakets zunächst prüfen, ob es

sich von dem bereits gesichteten unterscheidet. Falls ja, könnte eine Baumstruktur des Streams erzeugt werden, die alle fortan interpretierbaren Varianten enthält.

Erfolgt die Kommunikation auf Grundlage von UDP, liegt die Gefahr bei der Erzeugung eines Streams aus den Paketen darin, dass es, im Gegensatz zu TCP, keinen expliziten Verbindungsaufbau und -abbau gibt. Ohne genaue Kenntnis des genutzten Anwendungsprotokolls kann man nicht eindeutig sagen, wann ein UDP-Stream beginnt und wann er endet. Hier muss im Zweifelsfall auf eine Heuristik zurückgegriffen werden, die abschätzt wann eine Kommunikation zwischen zwei Partnern endet. Dies könnte ein Angreifer jedoch ausnutzen. Schätzt die Heuristik beispielsweise, dass nach einer bestimmten Zeitspanne die Verbindung beendet sei, könnte ein Angreifer diesen Umstand nutzen um Daten über vermeintlich mehrere UDP-Streams verteilt an der Beweismittelextraktion vorbei zu schleusen. Eine Möglichkeit diesen Fall auszuschließen, ist es zusätzlich zur Analyse einzelner UDP-Streams, auch Sequenzen mehrerer aufeinander folgender UDP-Streams in der Analyse zu betrachten.

2.4 Klassifikation von Daten

Liegen die zu untersuchenden Kommunikationsdaten in Form von Streams vor, kann mit der Analyse der enthaltenen Daten begonnen werden. Um eine möglichst hohe Erkennungsrate der Daten zu gewährleisten, müssen diese auf mehreren Ebenen klassifiziert werden. Diese Ebenen lauten: Dateiebene, Darstellungsebene und Protokollebene.
Sie ergeben sich aus dem OSI-Modell. Das Verhalten in den höheren Schichten Anwendungsschicht, Darstellungsschicht und Sitzungsschicht wird durch die Anwendungsprotokolle vorgegeben. Hier findet unter Anderem eine Veränderung der zu übertragenden Daten statt [Bra12, S. 37ff]. Um die Daten klassifizieren zu können, müssen diese Veränderungen rückgängig gemacht werden.
Für ein besseres Verständnis der Notwendigkeit dieser Ebenen werden diese und ihr Bezug zueinander im Folgenden in umgekehrter Reihenfolge ihrer Abarbeitung erklärt.

2.4.1 Dateiebene

Auf der untersten Ebene liegen die Dateien, die als zu extrahierendes Beweismittel ein gewünschtes Ziel bilden. Textbasierte Dateiformate lassen sich anhand ihres verwendeten Alphabets erkennen. Solange alle verwendeten Zeichen auf der Tastatur vorkommen, kann davon ausgegangen werden dass es sich um ein textbasiertes Dateiformat handelt. Aus weiteren Informationen, wie Dateiname oder verwendeten Keywörtern, können weitere Schlüsse über

das zugrundeliegende Format gezogen werden.
Auch bei binären Dateiformaten stellt die Dateiendung ein Indiz für den Typ dar. Eine genauere Erkennung ist typischerweise jedoch anhand ihres Dateiheaders möglich. Eine für das jeweilige Dateiformat spezifische Bytefolge kennzeichnet hierbei den Anfang der Datei. In einigen Fällen wird auch das Ende der Datei anhand eines Dateitrailers entsprechend markiert.
Für die Zuordnung von Dateiheadern und -trailern zu den entsprechenden Formaten gibt es im Internet ganze Sammlungen.[Kes16] Das Beispiel aus Abbildung 2.4 verdeutlicht den vereinfachten Aufbau einer Bilddatei im PNG-Format.

Dateiheader (hex)	Bildinformationen	Dateitrailer (hex)
89 50 4E 47 0D 0A 1A 0A	...	49 45 4E 44 AE 42 60 82

Abbildung 2.4: Vereinfachter Aufbau einer PNG-Bilddatei

Problematisch wird die Erkennung von Dateien, wenn diese nicht in ihrer ursprünglichen Darstellung übertragen werden. Dann ist es unter Umständen mehr nicht möglich mit den oben genannten Methoden eine Klassifikation durchzuführen. Es ist daher vorher notwendig, das Darstellungsformat zu erkennen und anschließend die ursprüngliche Darstellung wiederherzustellen.

2.4.2 Darstellungsebene

Für die Darstellung von Daten existieren eine Vielzahl von Darstellungsformaten, sogenannte *Encodings*. Die meistgenutzten Encodings neben dem Klartextformat sind Base64 und Quoted-printable. Für gewöhnlich verfolgt man bei der Nutzung von Encodings das Ziel, die Nutzung von Nicht-ASCII-Zeichen während der Übertragung von Daten zu vermeiden, um möglichen Konflikten mit protokollspezifischen Steuerzeichen aus dem Weg zu gehen. Zur Klassifikation einer Darstellung, könnte man daher zunächst das verwendete Alphabet überprüfen. Werden ausschließlich Zeichen verwendet, die zu einem bestimmten Encoding gehören, liegt der Verdacht nahe, dass dieses hier verwendet wurde.
Versucht man nun Daten zu dekodieren, ergibt sich ein Problem: In den meisten Fällen führt die Dekodierung von Daten mit einem falschen Algorithmus nicht etwa zu einem Fehler, sondern schlicht zu falschen Daten. Es müssten also mehrere Encodings bei der Suche nach Daten getestet werden.

In der Praxis wird im verwendeten Anwendungsprotokoll definiert, welche Encodings wann genutzt werden. Werden mehrere Encodings erlaubt, kommt in vielen Internetprotokollen der Multipurpose Internet Mail Extensions (MIME)-Standard [FB96] zum Einsatz. MIME gibt Auskunft über das verwendete Encoding während einer Übertragung von Daten.
Eine Auswertung der durch MIME vermittelten Information erlaubt das gezielte Anwenden der geeigneten Dekodierung.

Ein mögliches Angriffsszenario ergibt sich aus der Tatsache, dass Daten mehrfach, oder aber mit einem anderen Encoding als angegeben kodiert sein könnten. Will man dieses abmildern, müsste man mehrfach unterschiedliche Encodings testen, bis man die zugrundeliegenden Dateien erkannt hat.

2.4.3 Protokollebene

Nicht nur für die Erkennung des Encodings, auch zur Lokalisierung der Daten innerhalb eines Streams macht es Sinn das Anwendungsprotokoll zu kennen. Um ein Protokoll eindeutig klassifizieren zu können, muss dieses geparst werden können. Aufgrund der Vielzahl möglicher Protokolle bedeutet das nicht nur einen hohen Aufwand in der Implementierung, sondern auch zur Laufzeit des Programms. Schließlich müsste für eine Klassifikation jeder Stream mit jedem bekannten Protokoll testweise geparst werden.
Eine Möglichkeit diesen Vorgang zu beschleunigen ist die Verwendung einer Heuristik. Sie soll bereits vor dem Parsen anhand bestimmter Eigenschaften des Streams bereits eine Vorsortierung möglicher Protokollkandidaten erreichen. Im einfachsten Fall kann dies über die im Stream verwendeten Ports erreicht werden, z. B. wird HTTP-Verkehr in der Regel über Port 80 übertragen.

Auch ohne die Kenntnis des Protokolls, kann eine Dateiklassifikation von Klartextdaten erreicht werden. Dies hängt aber von den Eigenschaften des Protokolls ab. Für Dateiformate mit einem Dateitrailer reicht es, wenn die Übertragung der Nutzdaten nicht durch Protokolldaten unterbrochen wird. Für Dateiformate ohne Dateitrailer darf zur Wahrung der Korrektheit der Datei zusätzlich nach der Übertragung der Nutzdaten keine weitere Übertragung von Protokolldaten im Stream erfolgen.

Tabelle 2.1 zeigt diese Eigenschaften für einige Protokolle auf.

Tabelle 2.1: Verwendung von Protokoll- und Nutzdaten innerhalb eines Streams

Protokollname	PD vor ND	PD während ND	PD nach ND
HTTP 1.0	ja	nein	nein
HTTP 1.1	ja	nein	ja
SMTP	ja	nein	ja, sitzungsende
TFTP (UDP)	ja	ja, PaketInfos/Blocknummern	ja
FTP	nein	nein	nein

Legende:
PD – **P**rotokoll**d**aten
ND – **N**utz**d**aten

2.5 Entropie und Zufall

Eine weitere Herangehensweise zur Analyse von Daten ist die Nutzung statistischer Verfahren. Sie decken über die Vorkommnisse und Verteilung einzelner Zeichen bzw. Zeichengruppen innerhalb von Daten Muster oder Eigenschaften auf. Durch die Kombination mehrerer Verfahren lässt sich ebenfalls eine Form der Klassifikation realisieren.

Im Folgenden werden zwei konkrete Verfahren vorgestellt, die in Kombination eine grobe Klassifikation unbekannter Datenformate ermöglichen: Shannon Entropie und Chi-Quadrat-Verteilung.

2.5.1 Shannon Entropie

Der Begriff Entropie hat seinen Ursprung in der Thermodynamik. In der Informationstheorie wird er gedeutet als das "[...] Maß für die Ungewissheit [..], die im Mittel durch die Zeichen der Quelle aufgelöst wird."[Wer08, S. 5] Als *Quelle* wären hier die in Binärform nach Beweismitteln zu durchsuchende Daten, als *Zeichen* die einzelnen Bytes zu sehen.
Weniger abstrakt formuliert beschreibt die Entropie die Informationsdichte von Daten. Je mehr Bits für die Beschreibung von Informationen tatsächlich minimal benötigt werden, desto höher die Entropie. Ein Base64-kodierter Text nutzt nur 6 der möglichen 8 Bits pro Byte um den Inhalt darzustellen. Hier ist die Entropie entsprechend niedriger, als z. B. bei einer Zip-komprimierten Datei, wo versucht wird möglichst viele Informationen auf 8-Bits abzubilden.
Für die Darstellung und Berechnung von Entropie existieren viele Verfahren. Das bekannteste hiervon ist die Shannon Entropie, benannt nach seinem Erfinder Claude Shannon.
Er stellte für die Berechnung folgende Formel auf: [Völ14, S. 82]

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i$$

mit p_i als Wahrscheinlichkeit des Auftretens eines Zeichens i

Das Ergebnis seiner Berechnung ist die minimal benötigte Anzahl von Bits pro Zeichen für die Darstellung der in der Eingabe enthaltenen Informationen. Unter der Berücksichtigung, dass bei der Analyse unbekannter Daten ein Byte als einzelnes Zeichen zu betrachten ist, gilt somit:

$$0 <= H <= 8$$

Liegt für Daten eine Shannon Entropie von nahezu *8* vor, kann daraus ein Rückschluss über die mögliche Art der Daten gezogen werden. Eine solch hohe Entropie kommt lediglich in drei Fällen vor:

1. Die Daten sind zufällig. Dieser Fall kann hier vernachlässigt werden, weil die Übertragung von Zufallsdaten im Netzwerk eher ungewöhnlich ist.
2. Die Daten sind komprimiert.
3. Die Daten sind verschlüsselt.

Die beiden letzten Punkte sind bei der Klassifikation von Daten von Interesse. Sie können jedoch alleine durch die Verwendung einer Entropieanalyse nicht voneinander unterschieden werden. Um diese Unterscheidung zu ermöglichen, bedarf es einer weiteren Analyse.

2.5.2 Chi-Quadrat-Verteilung

Im Gegensatz zu Kompressionsverfahren, die lediglich versuchen unnötige Bits zu vermeiden, haben Verschlüsselungsverfahren das Ziel den erzeugten Datenstrom möglichst zufällig aussehen zu lassen. Dieser Umstand lässt sich für eine genauere Klassifikation nutzen. Zufällige Daten lassen sich von nicht-zufälligen Daten anhand der statistischen Verteilung ihrer Zeichen unterscheiden. Eine Messgröße für diese Verteilung stellt die Chi-Quadrat-Verteilung dar. Sie wird durch die folgende Formel berechnet: [KRES10, S. 88], [Knu98, S. 43]

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

mit i als Zeichen i,
O_i als beobachtete Anzahl von i,
E_i als erwartete Anzahl von i

Die so berechnete Größe erlaubt es, mit Hilfe ihres sogenannten Quantils, eine Aussage darüber zu treffen, ob die beobachteten Werte unter Annahme der Erwartungswerte statistisch wahrscheinlich sind. Laut Donald Knuth kann das Vorliegen von Zufallsdaten ausgeschlossen werden wenn das Quantil Q gilt: [Knu98, S. 46f]

$$Q < 0.01 \vee Q > 0.99$$

Die recht komplexe Berechnung des Quantils hängt lediglich von der Zahl der Freiheitsgrade ab. Im hier betrachteten Fall sind dies die möglichen Zeichen, die durch ein Byte dargestellt werden können. Da sich diese nie ändern, ist keine dynamische Berechnung dieser Werte notwendig. Stattdessen können die Werte mithilfe eines Chi-Quadrat-Rechners [Wal16] einfach im Voraus berechnet werden.
Es ergeben sich hierbei die folgenden Grenzen für χ^2, innerhalb derer Daten als zufällig und somit als verschlüsselt betrachten werden sollten:

$$206 < \chi^2 < 311$$

Damit das Verfahren korrekt funktioniert, sollte die Größe der Daten jedoch das Fünffache der Anzahl von möglichen Zeichen nicht unterschreiten. [Knu98, S. 45] Eine höhere Datengröße führt zu einem zuverlässigerem Ergebnis.
Hieraus folgt, dass dieses Verfahren nur dann angewendet werden sollte, wenn die im Stream enthaltenen Daten mindestens eine Größe von $5 * 256 = 1280$ Bytes besitzen. Streams mit einer kleineren Datengröße können so nicht mit ausreichender Zuverlässigkeit klassifiziert werden.

3 Anforderungsanalyse

Aufbauend auf den vermittelten Grundlagen werden in diesem Kapitel die Anforderungen an die zu entwickelte Anwendung definiert.

3.1 Allgemeines Ziel der Anwendung

Die zu entwickelnde Anwendung muss in der Lage sein, unverschlüsselte Netzwerkkommunikation in Form von PCAP-Dateien auf übertragene Dateien hin zu analysieren. Gefundene Dateien müssen automatisiert extrahiert werden. Zusätzlich zu den Dateien sind alle Informationen mit zu extrahieren, die einen Rückschluss auf den Übertragungsweg und die beteiligten Parteien bei der Übertragung einer Datei zulassen.

3.2 Nicht-funktionale Anforderungen

3.2.1 Benutzeranforderungen

Zielgruppe der Anwendung sind interessierte Anwender ohne besondere IT-Kenntnisse. Die Anwendung darf keinerlei Kenntnis über Datenextraktion auf Seiten des Nutzers voraussetzen. Der Zielgruppe wird lediglich die Fähigkeit vorausgesetzt, einen Mitschnitt von Netzwerkdaten mithilfe eines entsprechenden Tools anfertigen zu können. Die Entropie-basierte Extraktion hingegen ist als Option für erfahrenere Anwender zu betrachten.

3.2.2 Systemanforderungen

Auf der Softwareseite fungieren Linux und Python als Grundlage. Linux bietet nicht nur mehr Möglichkeiten zum Mitschnitt von Paketdaten als Windows, es existieren auch mehr Frameworks zur Verarbeitung von PCAP-Dateien. Insbesondere auf Basis von Python existiert eine große Auswahl. Durch die in der Regel mitgelieferte Python Paketverwaltung *pip* ist es ein leichtes, eventuell benötigte Abhängigkeiten nachträglich zu installieren. Es bleibt noch die Wahl zwischen Python 2 und Python 3. Da zu Beginn dieser Arbeit die meisten Linux

Distributionen noch auf Python 2.7 als Standard setzen, wird dieses auch hier vorausgesetzt.

Auf Hardwareseite werden keine besonderen Anforderungen an das System gestellt. Die maximale Größe der zu verarbeitenden PCAP-Datei hängt jedoch von der Größe des Arbeitsspeichers ab. Daher werden keine Garantien über das Verarbeitungspotential der Anwendung gegeben.

3.2.3 Architektur

Bei der Architektur der Anwendung kommen prinzipiell zwei Ansätze in Frage:

1. Die Modellierung als vollständig eigenständige Applikation, die eine vorliegende Eingabedatei verarbeitet.
2. Die Modellierung als Verarbeitungsschicht, die über Pipelining Echtzeiteingabedaten entgegennimmt und verarbeitet.

Wegen der besseren Testbarkeit und dadurch potentiell einfacheren Implementierung wird hier der erste Ansatz verfolgt. Die einzelnen Funktionsbausteine der Anwendung sollten jedoch für eine eventuelle Wiederverwendung in einem Framework modular aufgebaut sein.

3.2.4 Erweiterungsfähigkeit

Für die Verarbeitung der Protokoll-, Darstellungs- und Dateiebene ist ein Plugin-System zu entwickeln. Die Plugins sollen jeweils minimalen Implementierungsaufwand erfordern. Ein erfahrener Nutzer mit Python-Programmiererfahrung soll ein neues Dateiformat-Plugin in weniger als fünf Minuten implementieren können.

3.3 Funktionale Anforderungen

3.3.1 Benutzerschnittstelle

Wie für die meisten Linux Anwendungen üblich, erfolgt die Bedienung über ein Command-line interface (CLI). Dies erleichtert den Einsatz in der Batchverarbeitung. Der Aufbau soll jedoch eine spätere Ergänzung einer grafischen Oberfläche nicht unnötig erschweren.

3.3.2 PCAP-Dateiparsing

Als Eingabeformat dient das vom *libpcap*-Projekt spezifizierte PCAP-Dateiformat. Es hat die größte Verbreitung und gilt zum Zeitpunkt dieser Arbeit als Standard-Exportformat von Wi-

reshark.
Auf Layer 2 Ebene muss mindestens der Ethernet Standard unterstützt werden. Auf den darüber liegenden Schichten müssen IPv4-basierte TCP- und UDP-Pakete verarbeitet werden können.

Als Grundlage für die weiteren Verarbeitungsschritte sollen zu jedem Paket folgende Daten bzw. Meta-Daten eingelesen werden:

- Zeitstempel des Pakets
- Quell-IP-Adresse
- Quell-Port
- Ziel-IP-Adresse
- Ziel-Port
- Nutzdaten
- Prüfsummen (Layer 3 + 4)
- Flags (nur TCP)
- Sequenznummer (nur TCP)

Die für das Parsing notwendigen Routinen müssen nicht selbst implementiert werden. Stattdessen kann für die Realisierung der geforderten Funktionalitäten auf eine PCAP-Parsing-Library bzw. ein entsprechendes Framework zurückgegriffen werden.

3.3.3 Zusammensetzen von Streams

In der hier zu entwickelnden ersten Version sollen lediglich einfache Streams zusammengesetzt und verarbeitet werden. Die Kommunikation wird also nur unidirektional betrachtet. Die Nutzung kombinierter Streams zur bidirektionalen Betrachtung der Kommunikation ist vorerst kein Bestandteil der Anwendung.

Die Gültigkeit der eingelesenen Pakete muss vor dem Hinzufügen zu einem Stream anhand der Prüfsummen sichergestellt werden. Die Regeln, nach denen ein Paket zu einem Stream hinzugefügt wird, entsprechen der jeweiligen Beschreibung von TCP- und UDP-Streams aus dem Grundlagenteil. Im Fall von TCP sind Anfang und Ende eines Streams anhand der Flags auszumachen. Für UDP soll hierfür eine konfigurierbare, zeitbasierte Heuristik zum Einsatz

kommen. So hat es der Anwender selbst in der Hand, zu entscheiden wann ein UDP-Stream zu Ende ist. Abschnittsweise mit Pausen übertragene Daten können so bei Nutzung eines entsprechend hohen UDP-Timeout-Werts erkannt werden.

Auf eine Erkennung und gesonderte Behandlung von potentiellen Angriffsversuchen, wie z. B. den im Grundlagenteil beschriebenen Evasion-Angriff, wird verzichtet. Dies ist daher vertretbar, weil die Wahrscheinlichkeit eines Angriffs zur Umgehung der Beweismittelextraktion eher als gering einzuschätzen ist. Solche Szenarien finden eher mit dem Ziel der Umgehung von Intrusion-Detection-Systemen statt.

3.3.4 Nebenläufigkeit

Die weiterführende Analyse einzelner Streams erfolgt nebenläufig, um einen potentiellen Performancegewinn auf Mehrkernsystemen zu erhalten.

3.3.5 Protokollerkennung

Im ersten Release soll eine explizite Erkennung und Behandlung des HTTP-Protokolls erfolgen. Außerdem sollen über eine generische Protokollverarbeitung der Transfer von Dateien über FTP und proprietäre Protokolle erkannt werden können.
Die Erkennung der Protokolle muss durch eine Port-basierte Heuristik unterstützt werden. Zudem muss für die Abarbeitung der Protokolle eine Priorisierung vornehmbar sein.

3.3.6 Decoder

Zunächst sind nur Base64-Kodierung und Klartext als mögliche Kodierungen vorgesehen. Die Klartext-Kodierung ist priorisiert zu betrachten. Weitere Kodierungen müssen als Bestandteil der jeweiligen Protokolle verarbeitet werden, sofern diese sie nutzen.

3.3.7 Dateiklassifikation

Die Klassifikation von Dateien erfolgt standardgemäß über die Suche von Datei-Headern bzw. -Trailern.
Nach der Extraktion einer Datei wird der Stream direkt nach dem Ende der extrahierten Datei weiter untersucht, um auch konkatenierte Dateien behandeln zu können.

In der ersten Version sollen mindestens folgende Dateiformate erkannt und extrahiert werden:

- Bildformate: JPG, PNG, GIF
- Audioformate: WAV, MP3
- Videoformate: AVI, MPG
- Sonstige Formate: ZIP, PDF

Darüber hinaus muss auf Wunsch des Benutzers eine zusätzliche Extraktion und Klassifikation von Daten anhand der Entropie und Chi-Quadrat-Verteilung erfolgen. Dabei soll eine Unterteilung in drei Gruppen von Dateiformaten erfolgen:

- Verschlüsselte Dateiformate
- Komprimierte Dateiformate
- Klartext Dateiformate

Sollten die Daten nicht die geforderte Mindestgröße von 1280 Bytes besitzen, werden sie ohne weitere Analyse exportiert, jedoch als *unkategorisiert* markiert.

3.3.8 Ausgabeformat

Alle klassifizierten Dateien werden in strukturierter Form auf Dateisystemebene exportiert. Hierzu soll eine Ordnerstruktur entsprechend der Kategorie exportierter Daten, der beteiligten Kommunikationspartner und des Zeitpunkts der Übertragung genutzt werden.
Die mittels Entropieanalyse extrahierten Dateien werden dabei separiert von den Anderen abgelegt, weil diese lediglich erfahrenen Nutzern eine weitergehende manuelle Analyse vereinfachen sollen.

3.3.9 Visuelle Darstellung

Für die in dieser Arbeit zu entwickelnde Version ist noch keine visuelle Darstellung vorgesehen. Die Konzeption der Anwendung sollte jedoch mit Rücksicht auf die spätere Implementierung dieser Funktion erfolgen.

Die spätere Visualisierung ist mitsamt einer grafischen Benutzeroberfläche mit einem beliebigen modernen Browser abzurufen. Die Oberfläche ermöglicht den Start einer neuen Analyse und stellt anschließend die gefunden Dateien im Kontext der stattgefundenen Kommunikationen visuell dar. Die Visualisierung gibt dabei auch einen Überblick über die zeitlichen Aspekte

der Übertragungen. Dem Benutzer müssen auf Wunsch Details über die Übertragung einzelner Dateien angezeigt werden. Abbildung 3.1 dient als mögliches Beispiel für die Visualisierung.

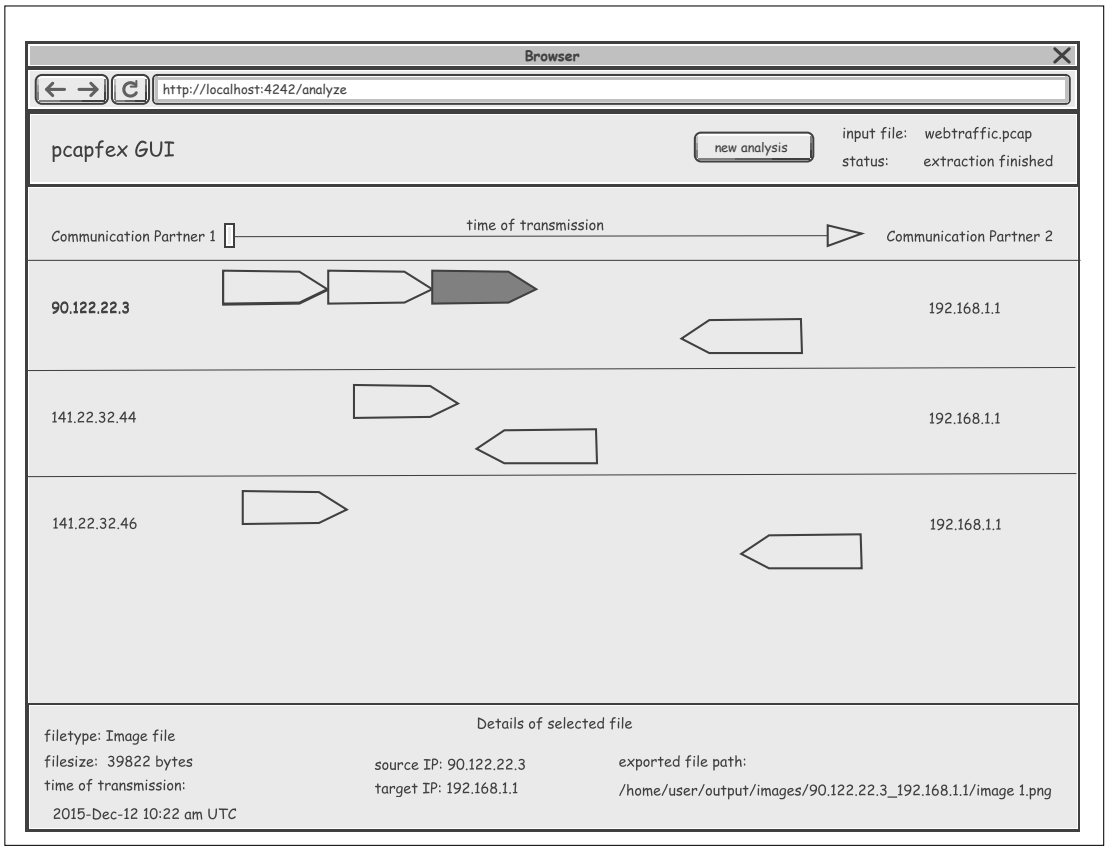

Abbildung 3.1: Mockup für Web-GUI und Visualisierung

4 Testszenarien

Im Folgenden werden Anwendungsszenarien beschrieben, die unterschiedliche Arten von Netzwerkkommunikation beschreiben. Sie dienen der Verifikation der Anwendung während der Entwicklung, zielen jedoch nur auf bestimmte Funktionalitäten ab. Sie sind kein Ersatz für spezifische Modultests. Jedes der Szenarien ist nachzustellen und der dabei entstehende Datenverkehr mitzuschneiden. Anschließend muss die Anwendung in der Lage sein, die geforderten Ziele zu erreichen. Eventuell für eine Verifikation der Anwendung notwendige Inhalte sind zusammen mit den Mitschnitten zu archivieren.

4.1 Download einer Zip-Datei von einer Webseite

Beschreibung

Ein Gerät lädt eine Zip-Datei eigenständig von einer Webseite.

Verwendetes Protokoll

HTTP 1.1

Initialisierung der Verbindung

Das Gerät stellt Anfrage für den Download der Datei an einen Server.

Inhalt der Anfrage

Ein HTTP-GET-Request an den Server mit dem Pfad der Datei sowie einiger HTTP-Header.

Inhalt der Antwort

Eine HTTP-Response-Nachricht mit HTTP-Headern und einer anschließenden Binärübertragung der angeforderten Zip-Datei.

Ziel der Extraktion

Die übertragene Zip-Datei.

4.2 Upload einer Bilddatei auf einer Webseite

Beschreibung

Ein Gerät lädt eine JPG-Datei eigenständig auf einen Webserver.

Verwendetes Protokoll

HTTP 1.1

Initialisierung der Verbindung

Das Gerät stellt Anfrage für den Upload der Datei an den Server.

Inhalt der Anfrage

Ein HTTP-POST-Request an den Server mit HTTP-Headern und einer anschließenden Binärübertragung der hochzuladenden JPG-Datei.

Inhalt der Antwort

Eine HTTP-Response-Nachricht mit HTTP-Headern und Bestätigung des Uploads.

Ziel der Extraktion

Die übertragene JPG-Datei.

4.3 FTP Download einer PDF-Datei

Beschreibung

Ein Gerät lädt ein PDF-Dokument von einem FTP-Server herunter.

Verwendetes Protokoll

FTP (Passiver Modus)

Initialisierung der Verbindung

Das Gerät führt zunächst eine Authentifizierung am Server durch. Anschließend stellt es eine Anfrage zum Download der Datei.

Inhalt der Anfrage

Der Wechsel in den passiven Modus und der anschließende Start der Übertragung per FTP-RETR.

Inhalt der Antwort

Zunächst die Öffnung einer separaten Kommunikationsverbindung und anschließend die Binärübertragung der PDF-Datei innerhalb der neuen Verbindung.

Ziel der Extraktion

Die übertragene PDF-Datei.

4.4 Übertragung einer Audiodatei über ein proprietäres Protokoll 1

Beschreibung

Ein Gerät überträgt eine zuvor aufgezeichnete Audiodatei im MP3-Format an einen Server.

Verwendetes Protokoll

Es wird kein standardisiertes Protokoll verwendet. Nach Aufbau der TCP-Verbindung wird die Audiodatei direkt Base64-kodiert übertragen. Anschließend wird die Verbindung wieder abgebaut.

Initialisierung der Verbindung

Es wird lediglich ein TCP-Handshake durchgeführt.

Inhalt der Anfrage

Die Audiodatei in Base64-kodierter Form.

Inhalt der Antwort

Keine Antwort.

Ziel der Extraktion

Die übertragene MP3-Datei.

4.5 Übertragung einer Audiodatei über ein proprietäres Protokoll 2

Beschreibung

Ein Gerät überträgt eine zuvor aufgezeichnete Audiodatei im MP3-Format an einen Server.

Verwendetes Protokoll

Es wird kein standardisiertes Protokoll verwendet. Nach Aufbau der UDP-Verbindung wird die Audiodatei direkt übertragen. Der Verlust von Paketen wird für dieses Szenario ausgeschlossen, auch kommen die Pakete in korrekter Reihenfolge an.

Initialisierung der Verbindung

Es wird keine Initialisierung durchgeführt.

Inhalt der Anfrage

Die Audiodatei in Binärform.

Inhalt der Antwort

Keine Antwort.

Ziel der Extraktion

Die übertragene MP3-Datei.

4.6 Übertragung einer verschlüsselten Datei über ein proprietäres Protokoll

Beschreibung

Ein Gerät sendet eine per AES-128-CBC verschlüsselte Datei mit einer Größe von etwa 1 MB an einen Server.

Verwendetes Protokoll

Es wird kein standardisiertes Protokoll verwendet. Es benutzt eine TCP-Verbindung. Für die Übertragung wird erst ein 32-Byte langer Header, dann die zu übertragene Datei, dann ein 8-Byte langer Trailer gesendet. Anschließend wird die Verbindung abgebaut.

Initialisierung der Verbindung

Es wird lediglich ein TCP-Handshake durchgeführt.

Inhalt der Anfrage

Die verschlüsselte Datei in Binärform.

Inhalt der Antwort

Keine Antwort.

Ziel der Extraktion

Bei der Dateiheader-basierten Klassifikation soll keine Extraktion erfolgen. Bei der Entropie-basierten Klassifikation soll die gesamte Kommunikation als eine Datei extrahiert werden, also Protokollheader, Datei und Protokolltrailer. Die extrahierte Datei soll als verschlüsselt gekennzeichnet sein.

5 Konzeption

Nachdem alle Anforderungen an die Anwendung definiert wurden kann ein Entwurfskonzept entwickelt werden. Im Folgenden wird das Konzept für die zu implementierende Anwendung erläutert, ohne die Generalisierung durch sprachspezifische Aspekte zu gefährden.
Die Anwendung trägt zudem fortan den Namen *"pcapfex"* (Packet Capture Forensic Evidence Extractor).

5.1 Allgemeiner Aufbau

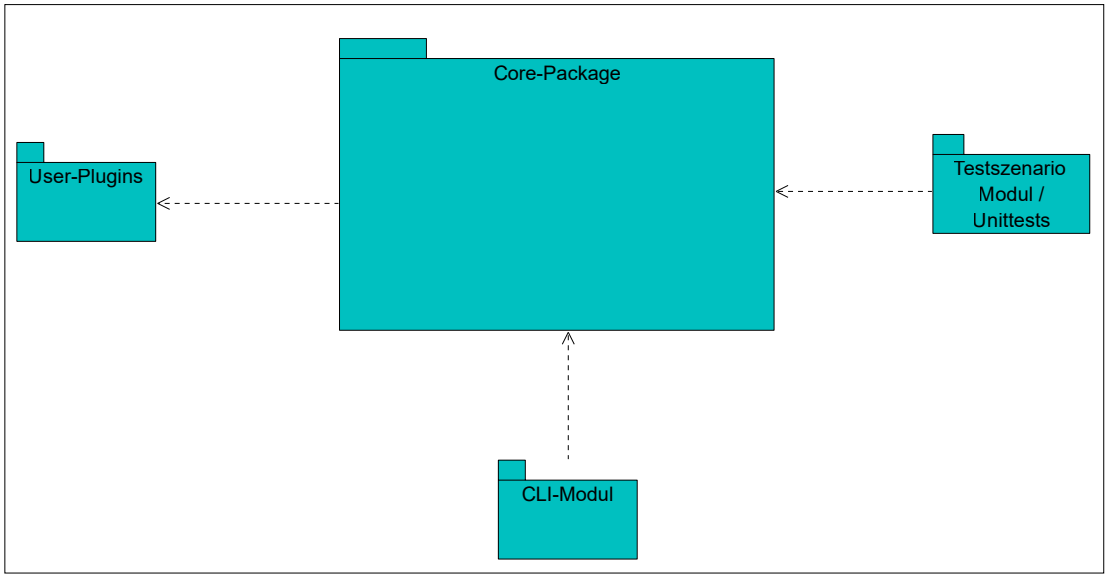

Abbildung 5.1: Übersicht der Packages in *pcapfex*

Abbildung 5.1 enthält eine Grobdarstellung der verwendeten Komponenten bzw. Packages. Der Anwendungskern befindet sich im Core-Package. Es referenziert das User-Plugins-Package und lädt hieraus die jeweiligen Plugins zur Erkennung von Anwendungsprotokollen, Encodings und Dateien. Das CLI-Modul ist im Prinzip lediglich ein einfacher Parser für die Eingaben

aus der Kommandozeile. Neben den für die Benutzerinteraktion notwendigen Ausgaben hält es eine Referenz auf das Core-Package und startet, wenn gefordert, den Analyseprozess. Im Testszenario-Modul befinden sich die notwendigen Daten für die Szenarien aus Kapitel 4. Für jedes der Szenarien findet sich hier ein entsprechender Unittest. Hierzu wird direkt auf die Funktionalitäten des Core-Packages zurückgegriffen.
Das Testszenario Modul ist nicht für die Auslieferung der Anwendung gedacht, es dient lediglich als Unterstützung in der Entwicklung von *pcapfex*.

5.2 Bestandteile des Core-Packages

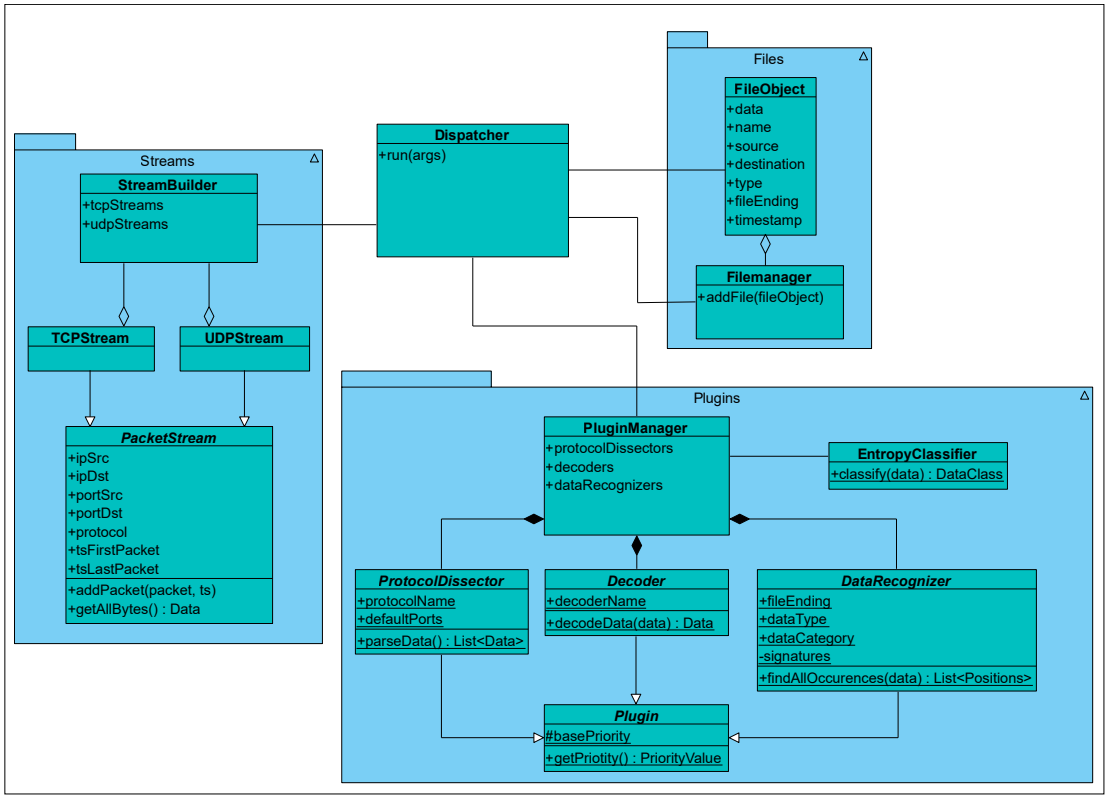

Abbildung 5.2: Subpakete und Klassen des Core-Packages

Abbildung 5.2 zeigt den Inhalt des Core-Packages. Im Zentrum steht der Dispatcher. Er bildet einerseits die Schnittstelle nach außen. Andererseits koordiniert er den gesamten Ablauf der Analyse. Hierfür greift er auf die Funktionalitäten der Subpakete Streams, Plugins und Files

zurück.

Das Streams-Paket beinhaltet die Klassen, die für das Zusammensetzen der TCP- und UDP-Streams notwendig sind. Der StreamBuilder übernimmt hierzu auch das Parsing der PCAP-Datei.

Das Plugins-Paket lädt die User-Plugins zur Laufzeit und hält diese nach Priorität bzw. Heuristik sortiert für die Benutzung durch den Dispatcher bereit. Es beinhaltet für jede Art von Plugin eine abstrakte Klasse, die von den User-Plugins implementiert werden muss.

Das Files-Paket ist für den strukturierten Export zuvor vom Dispatcher gefundener Dateien verantwortlich. Die Dateien werden mithilfe der FileObject Klasse mit den notwendigen Metadaten versehen und anschließend dem FileManager übergeben. Dieser wird nach dem Active Object Pattern implementiert. So können die Dateien asynchron auf die Festplatte geschrieben werden, ohne die weitere Suche nach Dateien zu unterbrechen.

5.3 Benutzerinteraktion

Das CLI-Modul ermöglicht dem Nutzer über die Nutzung von Kommandozeilenargumenten bzw. -parametern die Kontrolle über folgende Aspekte der Analyse:

1. Die zu analysierende PCAP-Datei. Sie wird als erstes Argument über ihren relativen Pfad angegeben.
2. Den Zielordner für die Ausgabe. Er wird optional als zweites Argument übergeben. Wird er nicht spezifiziert wird als Wert *'output'* angenommen.
3. Die Benutzung der Entropieanalyse. Sie wird standardmäßig nicht eingesetzt, kann aber über den Parameter *'-e'* aktiviert werden.
4. Das UDP-Timeout für die Stream-Heuristik in Sekunden. Es hat den Standardwert *120* und kann über den Parameter *'–T <wert>'* verändert werden.

Darüber hinaus hat der Benutzer über den Parameter *'-h'* die Möglichkeit, sich über die möglichen Argumente und Parameter zu informieren.

Während des Programmlaufs wird der Nutzer über den aktuellen Stand der Analyse informiert. Es erfolgen Ausgaben über:

- Einstellungen für die Analyse
- Größe der PCAP-Datei
- Aktueller Fortschritt des Parsings bzw. des Zusammensetzens von Streams
- Anzahl gefundener Streams
- Anzahl gefundener Dateien pro Stream
- Exportierte Dateien mit Pfad
- Eventuell während der Analyse aufgetretene Fehler

5.4 Ablauf eines Programmlaufs

Abbildung 5.3 zeigt den Ablauf des gesamten Prozesses zur Analyse einer PCAP-Datei. Zu beachten gilt hier, dass der Dispatcher für die Analyse der einzelnen Streams auf das Thread Pool Pattern zurückgreift. Die maximale Anzahl der Threads sollte hierbei zur Vermeidung übermäßigem Schedulings mit der Zahl der CPU-Kerne auf dem verwendeten System übereinstimmen. In der Abbildung wurde zugunsten der besseren Übersicht auf eine explizite Darstellung der Thread-Pool-Instanzen verzichtet.

Für die erwähnte Protokollerkennung wird auf die *parseData(data)*-Methode aus den Protocol-Dissector-Plugins zurückgegriffen. Sie liefert bei Erfolg eine Liste der einzelnen Nutzdatenabschnitte bzw. Payloads.
Ähnlich wird bei der Nutzung der Decoder-Plugins vorgegangen. Die *decodeData(data)*-Methode liefert nur dann Daten, wenn es bei der Dekodierung nicht explizit zu einem Fehler kam.
Die eigentliche Suche nach Dateien erfolgt schließlich anhand von regulären Ausdrücken, die Bestandteil der einzelnen DataRecognizer-Plugins sind. Hierzu wird auf die *findAllOccurences(data)*-Methode zurückgegriffen.
Gefundene Dateien werden vom Dispatcher als FileObject gekapselt, mit Metadaten versehen und anschließend dem FileManager zum asynchronen Dateisystemexport übergeben.

Durch die mehrfache Verschachtelung der Plugin-Schleifen beträgt die Laufzeitkomplexität für die Analyse eines Streams $O(n^3)$. Um diesem pessimistischen Fall in der Praxis möglichst selten zu begegnen, ist daher bei der Implementierung der Plugins stets darauf zu achten, Fehlerfälle frühzeitig zu erkennen. Sieht beispielsweise ein Protokoll eine bestimmte Zeichenkette

am Anfang des Streams vor, ist auf diese hin zu überprüfen und im Falle ihres Fehlens die weitere Protokollerkennung mit diesem Plugin direkt abzubrechen.

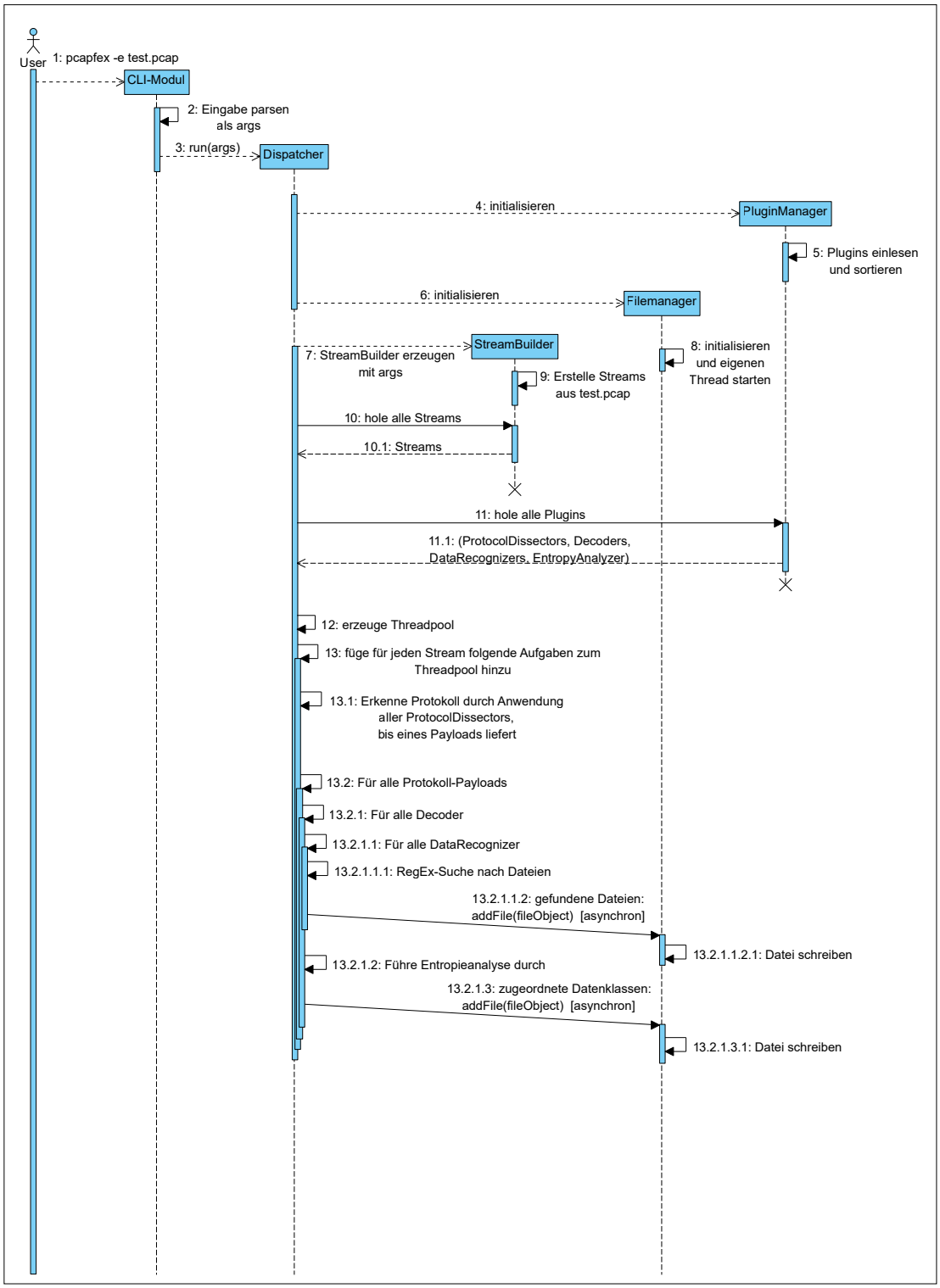

Abbildung 5.3: Sequenz zur Analyse einer PCAP-Datei

5.5 Zusammensetzen der Streams

Das Zusammensetzen der Streams übernimmt der StreamBuilder bei seiner Erzeugung. Er benutzt eine PCAP-Parsing-Library um die einzelnen Pakete aus der PCAP-Datei einzulesen. Abbildung 5.4 zeigt, wie der StreamBuilder die eingelesenen Pakete behandelt, um aus Ihnen die Streams zu formen.

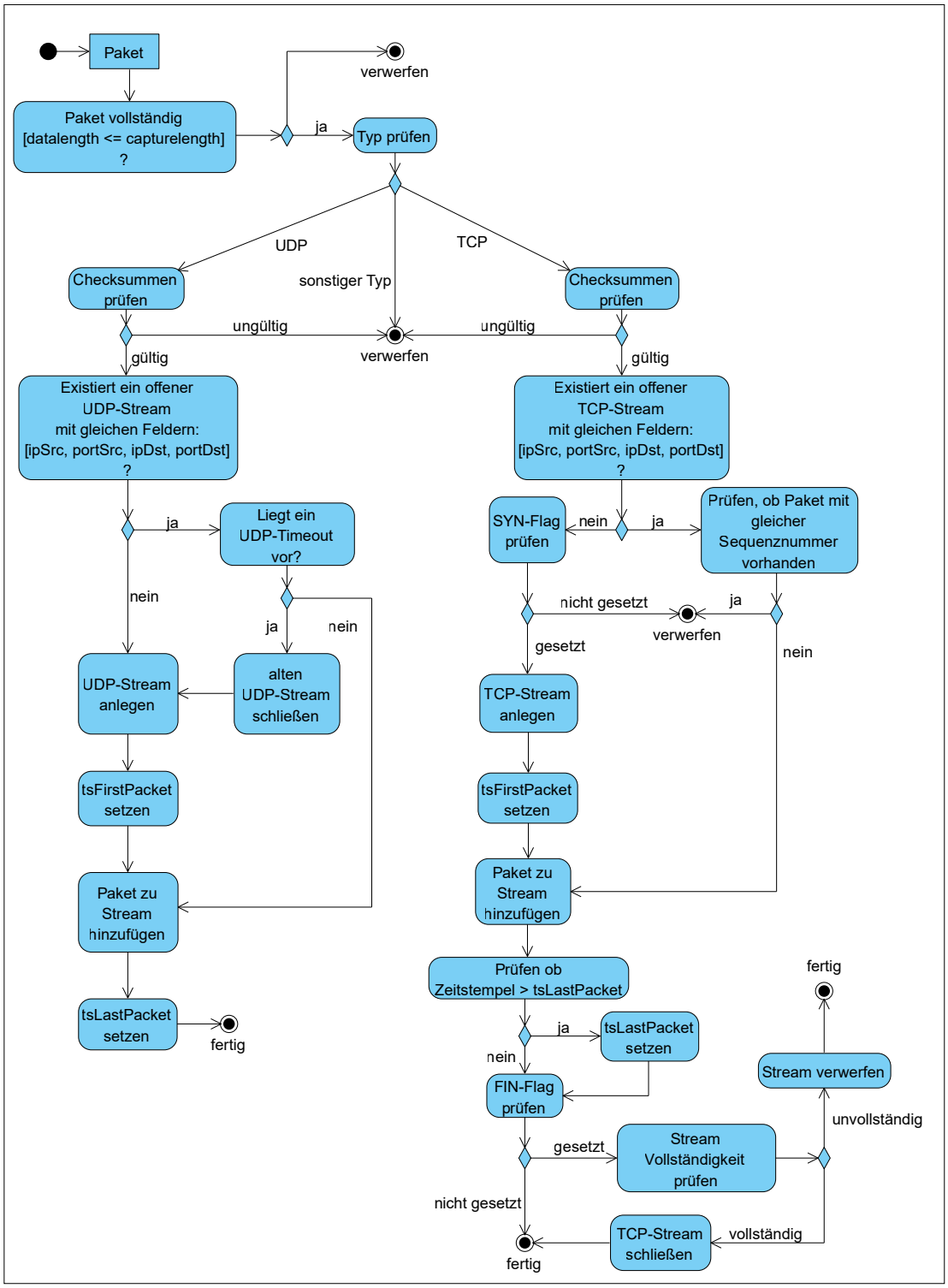

Abbildung 5.4: Erzeugung von Streams anhand der einzelnen Pakete

5.6 Laden von Plugins

Der PluginManager lädt bei seiner Instanziierung die Plugins. Sie liegen je nach Typ in einem der Unterordner:

- plugins/protocol_dissectors
- plugins/decoders
- plugins/data_recognizers

Hierzu iteriert er über alle Quellcodedateien in den jeweiligen Ordnern und lädt diese zur Laufzeit. Die Plugin-Quellcodedateien enthalten jeweils eine Klasse, die eine Subklasse des Typs '*Plugin*' entsprechend der Art des Plugins repräsentiert. Der PluginManager sammelt zunächst beim Laden die Referenzen auf diese Klassen in dem Typ entsprechenden Listen. Anschließend sortiert er die Listen nach der Priorität des Plugins, die von der Methode *getPriority()* geliefert wird.
Die Priorität eines Plugins wird als Zahl dargestellt. Je niedriger sie ist, desto höher die Priorität. Jedes Plugin hat eine Basispriorität < 100. Im Falle der Port-basierten Heuristik wird je bei der Berechnung der Priorität mit der *getPriority()*-Methode des ProtocolDissector-Plugins die Basispriorität um 50 reduziert, sofern der Stream einen Port des Plugins benutzt.

5.7 Durchführen der Entropieanalyse

Für die Entropieanalyse hält der PluginManager den EntropyClassifier vor. Abbildung 5.5 zeigt den Ablauf der Klassifikation.

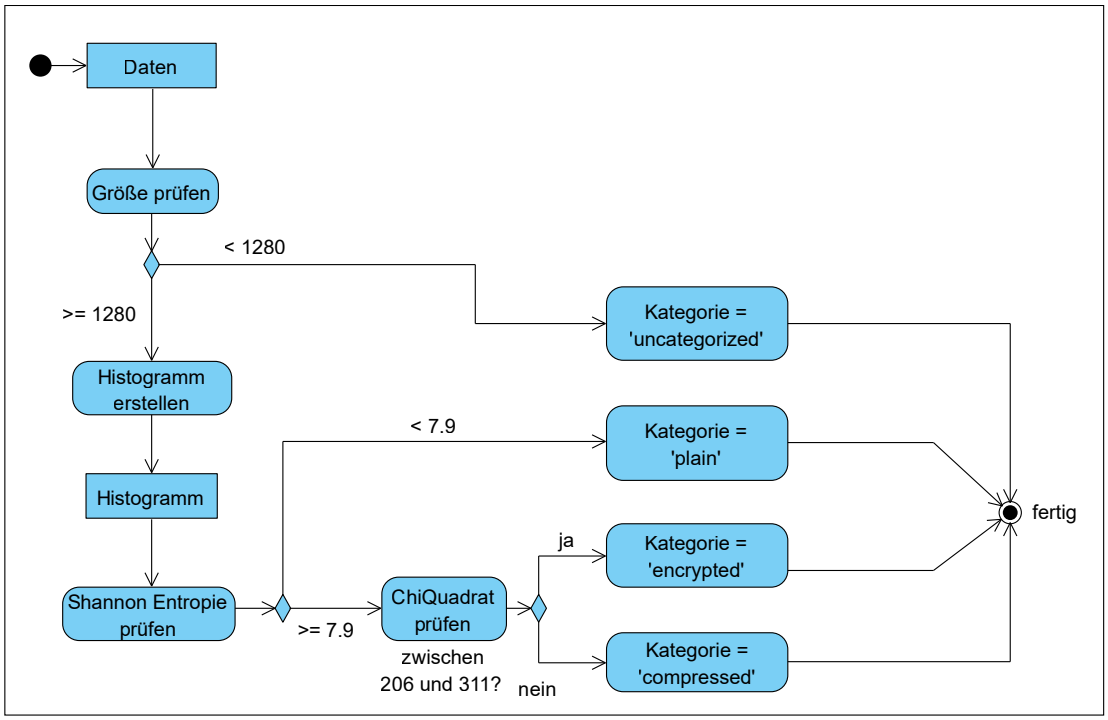

Abbildung 5.5: Datenklassifikation mittels Entropie- und ChiQuadrat-Analyse

5.8 Strukturierter Export

Der FileManager übernimmt den strukturierten Export der gefundenen Beweismittel bzw. Dateien auf Dateisystemebene. Die benötigten Informationen bezieht er hierfür aus den FileObject-Attributen. Das Exportformat ist dabei wie folgt spezifiziert:

<*Zielordner*>/<*type*>/<*source*>_<*destination*>/<*timestamp*>/<*name*>[<*Index*>].<*fileEnding*>

Die hierbei notwendigen Ordner sind vom FileManager anzulegen.
Die Benutzung des Index-Eintrags erfolgt nur dann, wenn bereits eine Datei mit demselben Namen existiert. Dann muss der Index angehängt und solange erhöht werden, bis ein Dateiname gefunden wurde der noch nicht existiert.
Das *timestamp*-Feld ist als UTC-Zeit in der Zeitzone des Rechners darzustellen, auf dem *pcapfex* gerade läuft. Die Umrechnung der Zeit erfolgt dabei unter der Annahme, dass der Mitschnitt

der Daten auf einem Rechner mit UTC-Empfänger erfolgt ist. Daher sind keine weiteren Schaltsekunden zu addieren oder andere Korrekturen vorzunehmen.

Neben den exportierten Beweismitteln legt der FileManager außerdem im Stamm des Zielordners eine Datei im JavaScript Object Notation (JSON)-Format ab, die eine Liste aller Beweismittel mit ihren jeweiligen Attributen enthält. Diese dient der einfachen Anbindung einer Visualisierung an die bestehende Struktur von *pcapfex*. Da dieses Feature jedoch für eine spätere Version von *pcapfex* vorgesehen ist, entfällt die Implementierung im zugehörigem Release dieser Arbeit.

6 Implementierung

Die Implementierung ist auf Grundlage des im Kapitel 5 erstellten Entwurfs erfolgt. Wie jedoch bei Softwareprojekten üblich, musste in manchen Punkten vom Entwurf abgewichen werden. Dieses Kapitel gibt Aufschluss darüber, wie die Implementierung erfolgt ist, welche Abweichungen dabei entstanden sind und warum diese notwendig waren. Außerdem wird die Python-spezifische Umsetzung einiger besonderer Aspekte anhand von Codeausschnitten erklärt.

6.1 Code-Richtlinien

Für eine bessere Übersicht über die verwendeten Packages und Klassen ist die Implementierung nicht nach den üblichen Richtlinien Pythons, sondern in Anlehnung an die Konventionen in Java erfolgt.
Die verwendeten Richtlinien lauten im einzelnen:

- Jede Klasse kommt in eine eigene Datei, außer zu Klassen zugehörigen Enums und Exceptions.
- Jedes Package kommt in einen eigenen Unterordner.
- Klassennamen werden groß geschrieben.
- Variablennamen werden klein geschrieben.
- Nicht-öffentliche Variablen und Methoden fangen mit '_' an.
- Aus mehreren Worten bestehende Bezeichner nutzen die CamelCase-Notation.
- Es werde möglichst sprechende Bezeichner verwendet, um die Notwendigkeit von Kommentaren zu minimieren.
- Alles ist auf Englisch auszudrücken.

6.2 Werkzeuge für die Entwicklung

Als integrierte Entwicklungsumgebung wurde *IntelliJ IDEA Ultimate 15* mit dem vom direkt vom Hersteller angebotenen Python-Plugin auf Basis von *PyCharm* eingesetzt [Jet15]. Neben Projektverwaltung, Syntax-Highlighting und Codevervollständigung bietet es einen umfangreichen Debugger, sowie einen Profiler.

Als Versionsverwaltung wurde Git eingesetzt. Hierzu wurde ein öffentliches Repository auf GitHub eingerichtet, wo *pcapfex* unter einer Open-Source-Lizenz zu beziehen ist [Win16].

6.3 Verwendete PCAP-Parsing-Library

Als PCAP-Parsing-Library wurde *dpkt* verwendet [DS15]. Es arbeitet objektorientiert und bietet die wichtigsten Grundfunktionalitäten für die Verarbeitung TCP/IP-basierter Kommunikation, wie zum Beispiel Prüfsummenbildung. Es unterliegt der BSD-Lizenz, darf somit also auch in Open-Source-Projekten eingesetzt werden. Außerdem wird es zum Zeitpunkt dieser Arbeit aktiv weiterentwickelt.

Listing 6.1 zeigt den Umgang mit *dpkt* zum Parsen einer PCAP-Datei.

```
import dpkt

# Datei simple.pcap öffnen
with open('simple.pcap', 'rb') as pcapFile:

    # Datei parsen und über einzelne Paketeinträge iterieren
    pcapReader = dpkt.pcap.Reader(pcapFile)
    for timeStamp, record in pcapReader:

        # Eintrag als Ethernet Frame einlesen,
        # dpkt erkennt Inhalte und parst automatisch auch Layer 3 + 4
            Protokolle
        eth = dpkt.ethernet.Ethernet(record)
        layer3 = eth.data
        layer4 = layer3.data

        if type(layer4) is dpkt.tcp.TCP:
            print 'TCP-Paket'
            # Verarbeite TCP-Paket
            # ...
        elif type(layer4) is dpkt.udp.UDP:
            print 'UDP-Paket'
            # Verarbeite UDP-Paket
            # ...
```

Listing 6.1: Vereinfachtes Beispiel zum Parsen einer PCAP-Datei mit *dpkt*

6.4 Umsetzung des Plugin-Systems

Python ermöglicht die Implementierung von Plugin-Konzepten mit sehr einfachen Mitteln. Das in Python integrierte *import*-Modul kann genutzt werden um Quelldateien zur Laufzeit in Bytecode zu kompilieren und zu laden.

Listing 6.2 zeigt den Ausschnitt des PluginManagers, der verantwortlich für das Laden der Plugins ist:

```
1  import imp
2  #...
3
4  class PluginManager:
5      # ...
6
7      def __loadPlugins(self, path, targetdict):
8              for pluginfile in os.listdir(path):
9                  if not os.path.isfile(path + pluginfile):
10                     continue
11                 if not pluginfile[-3:] == ".py":
12                     continue
13                 if pluginfile.endswith("__init__.py"):
14                     continue
15
16                 name = pluginfile.split('/')[-1][:-3]
17                 module = imp.load_source(name, path + pluginfile)
18                 targetdict[name] = module.getClassReference()
19                 # ...
```

Listing 6.2: Lademechanismus des PluginManagers

Die *__loadPlugins*-Methode erhält zwei Parameter: Der erste Parameter *path* stellt den Pfad zum Ordner dar, aus dem die Plugins geladen werden sollen. Der zweite Parameter *targetdict* ist eine Referenz auf ein Dictionary, welches anschließend den Namen, sowie eine Referenz auf die Klasse der jeweiligen Plugins halten soll. Die Methode iteriert über alle Dateien im angegebenen Ordner und stellt in den Zeilen 9 bis 14 zunächst sicher, dass es sich um eine gültige Quellcodedatei handelt. Anschließend wird sie in den Zeilen 16 bis 18 mithilfe des *import*-Moduls geladen und zum Dictionary hinzugefügt.

Die einzelnen Plugin-Module werden, wie im Entwurf gefordert, über die Ableitung von abstrakten Klassen dargestellt. Als Beispiel hierfür ist in Listing 6.3 die abstrakte Klasse für DataRecognizer-Plugins dargestellt:

```
1  # -*- coding: utf8 -*-
2  __author__ = 'Viktor Winkelmann'
3
4  from abc import ABCMeta, abstractproperty
5  from Plugin import *
6  try:
7      import regex as re
8      print 'Using concurrency enabled regex module.'
9  except:
```

```
10      print 'Consider installing the \'regex\' module using \'pip install
            regex\' to improve performance on multicore systems.'
11      import re
12
13
14  class DataCategory:
15      IMAGE = "Image file"
16      VIDEO = "Video file"
17      AUDIO = "Audio file"
18      DOC = "Document file"
19      TEXT = "Plaintext file"
20      EXECUTABLE = "Executable file"
21
22      COMPRESSED = "Compressed file"
23      ENCRYPTED = "Encrypted file"
24      UNKNOWN = "Unknown data"
25
26      def __iter__(self):
27          return self.__dict__.__iter__()
28
29
30  class DataRecognizer(Plugin):
31      __metaclass__ = ABCMeta
32
33      @classmethod
34      def getPriority(cls):
35          return cls.basePriority
36
37
38      @abstractproperty
39      def signatures(cls):
40          """ IMPORTANT: Override as Class Property """
41          return NotImplemented
42
43      @abstractproperty
44      def fileEnding(cls):
45          """ IMPORTANT: Override as Class Property """
46          return NotImplemented
47
48      @abstractproperty
49      def dataType(cls):
50          """ IMPORTANT: Override as Class Property """
51          return NotImplemented
```

```
52
53      @abstractproperty
54      def dataCategory(cls):
55          """ IMPORTANT: Override as Class Property """
56          return NotImplemented
57
58      @classmethod
59      def _buildRegexPatterns(cls):
60          regexstr = b''
61          for (fileHeader, fileTrailer) in cls.signatures:
62              if fileTrailer is None:
63                  regexstr += b'(%s.*)|' % (fileHeader,)
64              else:
65                  regexstr += b'(%s.*?%s)|' % (fileHeader, fileTrailer)
66
67          cls._regex = re.compile(regexstr[:-1], re.DOTALL)
68
69      @classmethod
70      def findAllOccurences(cls, data, startindex=0, endindex=0):
71          if not hasattr(cls, '_regex'):
72              cls._buildRegexPatterns()
73
74          if endindex == 0:
75              endindex = len(data)
76
77          return [m.span() for m in cls._regex.finditer(data, startindex,
                endindex)]
```

Listing 6.3: Abstrakte Klasse für DataRecognizer-Plugins

Wie man in den Zeilen 38 bis 56 sieht wird hier auf das Python-spezifische Konzept der abstrakten Klassenvariablen zurückgegriffen. Sie sind das Einzige, worüber eine implementierende Klasse verfügen muss. Die notwendigen Methoden sind bereits alle in der abstrakten Klasse *DataRecognizer* definiert.
Die Methode *findAllOccurences* baut zunächst aus den möglichen Signaturen einer Datei einen einzigen regulären Ausdruck und gibt dann die Positionen aller Funde zurück.
Am Beispiel des Plugins für JPEG-Dateien in Listing 6.4 wird die Einfachheit der Implementierung von Plugins noch deutlicher:

```
import sys

sys.path.append('../..')
from core.Plugins.DataRecognizer import *

def getClassReference():
    return JpegFile


class JpegFile(DataRecognizer):
    signatures = [(b'\xFF\xD8\xFF', b'\xFF\xD9'), (b'.{6}\x4A\x46\x49\x46\
        x00', None)]
    fileEnding = "jpg"
    dataType = "JPEG file"
    dataCategory = DataCategory.IMAGE
```

Listing 6.4: Implementierende Klasse für JPEG-Dateien

Die im Modulscope befindliche Methode *getClassReference()* hilft dem PluginManager dabei die Referenz auf die korrekte Klasse in der Quellcodedatei zu bekommen. Die implementierende Klasse *JpegFile* selbst hält ausschließlich die abstrakten Klassenvariablen, von denen *signatures* die wichtigste ist. Sie ist eine Liste aus Tupeln. Jedes Tupel besteht aus zwei Strings, die jeweils einen regulären Ausdruck für Dateiheader und Dateitrailer darstellen. Hat ein Dateiformat keinen Dateitrailer, so ist dieser *None*.

6.5 Multithreading in Python

Der Umgang mit Threads in Python hängt von der verwendeten Python Implementierung ab. Neben der auf C basierenden Standardimplementierung von Python, *CPython*, gibt es weitere Implementierungen die zumeist schlichtweg auf anderen Sprachen basieren. Dazu gehören unter Anderem *Jython*, *PyPy* und *IronPython*. Nicht alle Implementierungen beherrschen echtes Multithreading. Inbesondere im Fall von *CPython* ist dies ein Problem, weil es die wohl am meisten genutzte Python Variante ist. Es ist in *CPython* zwar möglich mehrere Threads zu erzeugen, sie laufen jedoch nicht parallel. Nur während blockierender Operationen, wie etwa I/O-Zugriffen, kommt es zu echter Nebenläufigkeit. Bei der Abarbeitung von Python-Code wird die Nebenläufigkeit nur durch das Abwechselnde Nutzen von Threads emuliert. Dieses Verhalten liegt am sogenannten Global Interpreter Lock (GIL) [Mai15]. Der GIL ist ein Mutex mit dem Zweck die Speicherverwaltung von *CPython* auf einfache Art threadsafe zu machen. Will ein Thread Python-Code ausführen, muss er zunächst Besitz vom GIL erlangen. Wird

C-Code ausgeführt, kann im Code der GIL wieder freigegeben werden. Natürlich darf dies nur dann geschehen, wenn die durchgeführte Operation an sich threadsafe ist.

Im Fall von *pcapfex* bedeutet der GIL konkret, dass die Suche nach Dateien mit *CPython* zunächst nicht nebenläufig funktioniert. Es gibt jedoch Bemühungen, die es erlauben, die aufwändigste Operation bei der Suche nach Dateien, die Suche nach regulären Ausdrücken (RegEx), nebenläufig zu gestalten. Ein neues RegEx-Modul für *CPython*, welches langfristig das aktuelle Modul *re* abwechseln soll, erlaubt die Freigabe des GIL während der Suche [Bar16]. Es ist weitestgehend kompatibel zur aktuellen Implementierung und kann bereits jetzt heruntergeladen und installiert werden.
In Listing 6.3 Zeile 6 bis 11 sieht man, dass in der Implementierung der DataRecognizer-Plugins einerseits das neue Modul *regex* bevorzugt wird, andererseits der Benutzer eine Empfehlung erhält dieses zu installieren, falls es nicht vorhanden ist.

6.6 Optimierung von Laufzeit und Speicherverbrauch

Bei der Implementierung von *pcapfex* wurde auf zwei Hilfsmittel zurückgegriffen, um die Performance zu optimieren.

Das erste Hilfsmittel ist der in IntelliJ IDEA integrierte Profiler. Er protokolliert bei der Ausführung von Code, wie viel Zeit die einzelnen Methodenaufrufe in Anspruch genommen haben. Außerdem liefert er eine Graphendarstellung, aus der hervorgeht in welchem Kontext die einzelnen Methoden aufgerufen wurden. Abbildung 6.1 zeigt die Ausgaben des Profilers Hiermit lassen sich Stellen im Code lokalisieren, bei denen unter Umständen eine Optimierung Sinn macht. So hat sich während der Entwicklung ergeben, dass alleine durch die Benutzung der Klasse *dict* anstelle eines *OrderedDict* zum Sammeln von TCP-Paketen, sowie den Wegfall einiger unnötiger Iterationen, die Laufzeit um mehr als 90% reduziert werden konnte.

Das zweite Hilfmittel ist das Speicheranalysetool *Pympler* [BHS15]. Es wird in den zu analysierenden Code eingebunden und protokolliert, wie viele Objekte welchen Typs im geprüften Teil des Codes erzeugt wurden. Auch der dadurch verbrauchte Speicher wird angezeigt. Abbildung 6.2 enthält eine Beispielausgabe für den Speicherverbrauch der DataRecognizer-Module. Durch das Verhindern der mehrfachen Erzeugung einiger Objekte im Code, wie etwa einiger Strings und RegEx-Objekte, konnte während der Implementierung der Speicherverbrauch um etwa 30% reduziert werden.

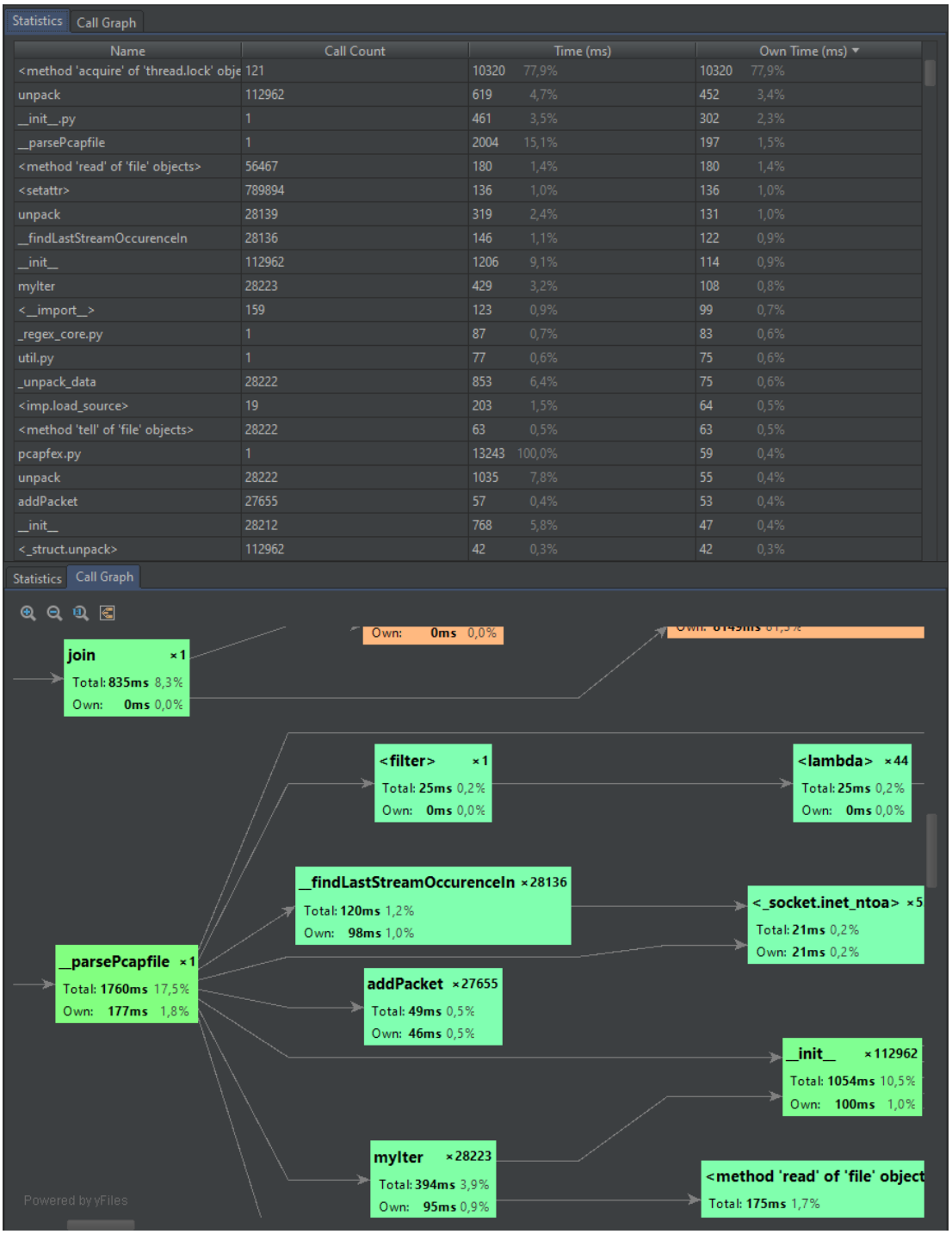

Abbildung 6.1: Ausgaben des Profilers in IntelliJ IDEA

types	# objects	total size
list	2077	2.82 MB
str	2049	90.11 KB
dict	25	4.54 KB
int	156	1.83 KB
tuple	14	560 B
wrapper_descriptor	13	520 B
method_descriptor	13	468 B
listiterator	9	288 B
weakref	4	176 B
<class '_regex_core.GreedyRepeat	5	160 B
<class '_regex_core.Sequence	5	160 B
instancemethod	4	160 B
member_descriptor	4	144 B
getset_descriptor	4	144 B
cell	5	140 B

Abbildung 6.2: Ausgabe von *pympler*

6.7 Abweichungen vom Konzept

Während der Implementierung kam es bei den Testszenarien 4.2 und 4.3 zu unvorhergesehenem Verhalten. Einige Streams wurden verworfen, weil die zugehörigen Pakete laut *pcapfex* falsche Prüfsummen aufwiesen. Da jedoch die an der Kommunikation beteiligten Rechner diese Pakete akzeptiert hatten, musste der Sache nachgegangen werden. Eine nähere Betrachtung mit Wireshark ergab, dass die Prüfsummen tatsächlich falsch waren. Wireshark gibt sogar einen Hinweis, dass hier eventuell das sogenannte *TCP-Checkum-Offloading* schuld sei [wir08]. Hierbei handelt es sich um eine Hardwareunterstützung zur Berechnung der Prüfsummen von zu sendenden Paketen. Das Betriebssystem übergibt nur das rohe Datenpaket an die Netzwerkkarte, diese berechnet die Checksumme eigenständig und versendet das Paket. Zeichnet man den Datenverkehr auf dem sendenden Rechner auf, so fehlen die korrekten Prüfsummen in den Paketen, weil sie auf Betriebssystemebene nicht existieren.
In der Tat erklärte dies die Ursache für Testszenario 4.2, wo ja schließlich auch Daten versendet wurden. In dem Testszenario 4.3 war jedoch der aufzeichnende Rechner gar nicht für das Senden von Paketen verantwortlich. Trotzdem kam es zu falschen Prüfsummen. Auffällig

war hier, dass die Größe der empfangenen Pakete höher war als der Maximum Transfer Unit (MTU)-Wert der verwendeten Internetleitung. Es hätte eine Fragmentierung stattfinden müssen. Davon war jedoch im Paketmitschnitt nichts zu sehen. Eine Recherche ergab, dass auch hier eine Optimierung verantwortlich dafür ist. Durch die sogenannte *Receive Aggregation* werden fragmentierte Pakete unter Nutzung der Möglichkeiten der Hardware beim Empfang zusammengefasst, bevor sie an die Anwendung weitergegeben werden. Auf die Neuberechnung einer Prüfsumme wird dabei verzichtet [MZ08].

Um diesen Problemen zu begegnen wurde das Konzept um die Möglichkeit erweitert, die Validierung der Prüfsummen mit dem Parameter *-nv* zu deaktivieren. Dadurch werden alle Pakete als valide angesehen und weiterverarbeitet. Hierfür wurden das *CLI-Modul* und der *StreamBuilder* entsprechend angepasst.

Im Allgemeinen ist es empfehlenswert, den Mitschnitt von Daten auf einem unbeteiligten Rechner zu erstellen, um nicht auf die Validierung der Pakete verzichten zu müssen.

6.8 Ergebnis der Implementierung

Das Ergebnis der Implementierung ist die Release Version 1.0 von *pcapfex*. Um die Funktionalität der Anwendung zu überprüfen, wurde sie neben den in Kapitel 4 definierten Testszenarien anhand einiger weiterer Paketmitschnitte getestet.
Das System, auf dem getestet wurde, verfügt über die folgenden technischen Daten:

- Intel Core i5-3220M @ 2.6 GHz (Dual-Core mit Hyperthreading)
- 8 GB RAM
- 480 GB SATA3 SSD
- Linux Mint 17.2 64-Bit
- CPython 2.7.6 64-Bit

Die Tests wurden jeweils mit einer Entropieanalyse, jedoch ohne eine Prüfummenvalidierung durchgeführt. Für die Zeitmessung wurde das arithmetische Mittel aus drei Testläufen genommen. Tabelle 6.1 dokumentiert den Ausgang der Testläufe.

Tabelle 6.1: Ergebnisse der Tests mit *pcapfex* 1.0

Testname	Erfolg	Anzahl FP	Anzahl FE	Zeit
Szenario 4.1	ja	0	0	335 ms
Szenario 4.2	ja	0	0	155 ms
Szenario 4.3	ja	0	0	199 ms
Szenario 4.4	ja	0	1	274 ms
Szenario 4.5	ja	0	1	259 ms
Szenario 4.6	ja	1	0	335 ms

Legende:

FP – **f**alsch-**p**ositive Funde

FE – **f**alsche **E**ntropieklassifizierung

Wie man anhand der Ergebnisse sieht, arbeitet *pcapfex* recht zufriedenstellend. Die definierten Ziele der Testszenarien wurden alle erreicht.
Die Entropieanalyse hat in den Testszenarien 4.4 und 4.5 entgegen der natürlichen Erwartung die MP3-Datei als Klartextdaten eingestuft, anstatt als komprimiert. Hier sollte geprüft werden, ob mit einem anderen Grenzwert für die Entropie eventuell noch ein besseres Ergebnis zu erreichen ist.
In Szenario 4.6 sieht man, dass bedingt durch die Verwendete Methode zur Erkennung von Dateiformaten, stets auch mit falsch-positiven Funden zu rechnen ist. Um dem vorzubeugen könnte man die DataRecognizer-Plugins um eine Methode erweitern, die einen Fund zusätzlich auf seine Validität überprüft. Dies wäre jedoch einerseits sehr aufwändig, andererseits könnte es als Einfallstor für Code-Injection-Angriffe genutzt werden. Daher sollte eine Validierung ausschließlich in einem Sandboxprozess durchgeführt werden.

7 Fazit & Ausblick

Mit *pcapfex* ist es tatsächlich gelungen, den Prozess der forensischen Analyse so stark zu vereinfachen, dass auch wenig erfahrene Anwender schnell Ergebnisse erzielen können. Die Ergebnisse sind jedoch zum Teil mit Vorsicht zu genießen. Aktuell werden nur einige bestimmte Dateiformate genau erkannt. Man müsste für jedes bekannte Dateiformat ein Plugin implementieren. Es kann daher nie das Auffinden aller Dateien garantiert werden.
Außerdem kommt es, insbesondere bei größeren Datenmengen, mit hoher Wahrscheinlichkeit zu falsch-positiven Funden. Diese müssen aktuell händisch aussortiert werden, was in der Regel heißt man müsste die einzelnen Dateien in einem geeigneten Programm öffnen und schauen ob es zu Problemen kommt. Da es sich jedoch bei den Dateien auch um Malware handeln könnte, sollte man hier entsprechende Vorsichtsmaßnahmen ergreifen.
Ein weiterer zu berücksichtigender Aspekt bei der Betrachtung der Ergebnisse sind Mehrfachfunde desselben Dateiobjekts. Bei Formaten, die die Einbettung anderer Dateien vorsehen etwa. Enthält zum Beispiel eine PDF-Datei zwei PNG-Dateien, wird *pcapfex* hier drei Dateien exportieren.

Die Entropieanalyse hat sich als gutes Mittel herausgestellt, um erfahrenen Anwendern eine tiefer gehende Analyse zu vereinfachen. Sie ist unabhängig von den Dateiformat-Plugins, jedoch nicht von den Protokoll-Plugins. Es ist daher immer möglich, dass eine über die Entropieanalyse exportierte Datei Protokollbestandteile enthält oder sogar aus mehreren einzelnen Dateien besteht. Dies ist bei der weiteren Analyse stets zu berücksichtigen.

Für die Weiterentwicklung von *pcapfex* sind zwei Pfade vorgesehen, die sich nicht gegenseitig ausschließen.
Der erste Entwicklungspfad ist das Ausbauen und Vervollständigen des aktuellen Ansatzes. Hierzu gehören die Implementierung weiterer Plugins, sowie die bereits im Konzept berücksichtigte grafische Oberfläche inklusive Visualisierung des Datenflusses. Auch die Implementierung von kombinierten Streams würde den aktuellen Ansatz sinnvoll erweitern. So ist es zum Beispiel aktuell nicht möglich, den Namen einer über das HTTP-Protokoll heruntergeladenen Datei zu

bestimmen. Der Name ist schließlich nur im Request enthalten, nicht in der Response-Nachricht. Durch die Nutzung kombinierter Streams wäre auch der Name der Datei bestimmbar.
Der zweite Entwicklungspfad befasst sich mit der Redefinition der Methodik zum Auffinden und klassifizieren von Dateiobjekten. Es gilt eine Methodik zu finden, die die Eigenschaften der beiden bisherigen Methoden Header/Tailer- und Entropieanalyse miteinander kombiniert. Deep-Learning-Methoden könnten hier eine sinnvolle Basis bilden. So könnte bei korrekter Anwendung einerseits die Klassifikation von Dateien zuverlässiger erfolgen als bisher, andererseits könnte man durch Mustererkennung auf die händische Implementierung weiterer Plugins zur Dateierkennung verzichten.

Welcher der beiden Entwicklungspfade als erstes eingeschlagen wird, ist nebensächlich. In jedem Fall kann abschließend gesagt werden, dass mit *pcapfax 1.0* durchaus eine brauchbare Grundlage für ein forensisches Analysetool geschaffen wurde, für das sich eine eine Weiterentwicklung unter dem Open-Source-Aspekt lohnt.

Literaturverzeichnis

[Bar16] Matthew Barnett. mrab-regex. https://bitbucket.org/mrabarnett/mrab-regex, 2016. [Online; abgerufen 21. Februar 2016].

[BHS15] Jean Brouwers and Ludwig Haehne and Robert Schuppenies. Pympler. http://pythonhosted.org/Pympler/, 2015. [Online; abgerufen 22. Februar 2016].

[Bra12] Christian Braun. *Computernetze kompakt*. Springer Verlag, 2012.

[DS15] Dug Song, others. dpkt. https://github.com/kbandla/dpkt, 2015. [Online; abgerufen 01. November 2015].

[FB96] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME). RFC 2045, September 1996. Update of RFC 822.

[Har15] Guy Harris. Libpcap file format. https://wiki.wireshark.org/Development/LibpcapFileFormat, 2015. [Online; abgerufen 08. Dezember 2015].

[Jet15] JetBrains. Intellij idea. https://www.jetbrains.com/idea/, 2015. [Online; abgerufen 01. November 2015].

[Kes16] Gary Kessler. File signatures table. http://www.garykessler.net/library/file_sigs.html, 2016. [Online; abgerufen 13. Februar 2016].

[Knu98] Donald Knuth. *The Art of Computer Programming, Volume Two, Seminumerical Algorithms*. Addison-Wesley, 1998.

[KRES10] Udo Kuckartz and Stefan Rädiker and Thomas Ebert and Julia Schehl. *Statistik: Eine verständliche Einführung*. VS-Verlag, 2010.

[Mai15] Wolfgang Maier. Python wiki - globalinterpreterlock. https://wiki.python.org/moin/GlobalInterpreterLock, 2015. [Online; abgerufen 21. Februar 2016].

[MZ08] Aravind Menon and Willy Zwaenepoel. Optimizing tcp receive performance. pages 85–98, 2008. USENIX 2008 Annual Technical Conference.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[Sou15] Open Source. Tcpdump & libpcap. http://www.tcpdump.org, 2015. [Online; abgerufen 08. Dezember 2015].

[Vac13] John R. Vacca. *Computer and Information Security Handbook (Second Edition).* Elsevier Inc., 2013.

[Völ14] Horst Völz. *Grundlagen und Inhalte der vier Varianten von Information: Wie die Information entstand und welche Arten es gibt.* Springer-Verlag, 2014.

[Wal16] John Walker. Chi-square calculator. http://www.fourmilab.ch/rpkp/experiments/analysis/chiCalc.html, 2016. [Online; abgerufen 15. Februar 2016].

[Wer08] Martin Werner. *Information und Codierung, 2. Auflage.* 2008.

[Win16] Viktor Winkelmann. pcapfex. http://vikwin.github.io/pcapfex/, 2016. [Online; Ergebnis dieser Arbeit].

[wir08] wireshark.org. Wireshark wiki - tcp checksum verification. https://wiki.wireshark.org/TCP_Checksum_Verification, 2008. [Online; abgerufen 20. Februar 2016].

Glossar

Active Object Pattern Ein Entwurfsmuster für Nebenläufigkeit. Das Active Object sammelt angeforderte Aktionen (Methodenaufrufe) in einer synchronisierten Queue und führt diese anschließend in einem separaten Thread aus. Es wird insbesondere bei I/O-intensive Operationen verwendet. 25

AES-128-CBC AES (Avanced Encryption Standard) ist ein modernes symmetrisches Verschlüsselungsverfahren. Es arbeitet in Blöcken und unterstützt Schlüssel mit einer Länge von 128, 192 oder 256 Bit. CBC (Cipher Block Chaining) ist ein Modus, bei dem jeder Block vor dem verschlüsseln mit seinem bereits verschlüsselten Vorgänger per XOR verknüpft wird. 21

Base64 Ein Kodierungsverfahren, bei dem 8-Bit Daten in einer ASCII-kompatiblen 6-Bit Darstellung abgebildet werden. Aus den 6 Bits ergibt sich ein Zeichenalphabet mit einer Länge von 64. 7

Code-Injection Ein Angriff, bei dem eine Sicherheitslücke in der Dateneingabe eines Programms ausgenutzt wird. Über die Lücke wird fremder Programmcode eingeschleust und im Kontext des angegriffenen Prozesses ausgeführt. 45

Command-line interface (CLI) Eine Mensch-Maschine-Schnittstelle, bei der der Mensch der Maschine Befehle über Tastatureingaben gibt. 13

Deep-Learning Eine Kategorie des maschinellen Lernens. Sie gilt aktuell als eine der vielversprechendsten Kategorien im Bereich der künstlichen Intelligenz. Sie umfasst unter Anderem die Nutzung neuronaler Netze als Klassifikator. 46

Drift Die maximal entstehende Ungenauigkeit einer Uhr in einem gegebenem Zeitraum. 4

Evasion-Angriff Eine Technik zur Umgehung von Netzwerksicherheitseinrichtungen. Beruht darauf, dass das Angriffsziel Daten anders verarbeitet als die Sicherheitseinrichtung. 5

Global Interpreter Lock (GIL) Ein Mutex-Objekt in einigen Python-Implementierungen, welches das Ausführen von Python-Bytecode auf einen Thread zur Zeit beschränkt. 40

JavaScript Object Notation (JSON) Ein Datenformat für die Serialisierung von Objekten. Entgegen der Namensgebung wird es auch in anderen Programmiersprachen benutzt. 33

Maximum Transfer Unit (MTU) Die maximale Paketgröße, die über ein Netzwerk geroutet werden kann. Sollen größere Pakete übertragen werden, müssen diese fragmentiert werden. 43

Multipurpose Internet Mail Extensions (MIME) Ein Standard für die Repräsentation von Nachrichten im Internet. Er wurde ursprünglich für Emails entwickelt, wird jedoch auch in vielen anderen Internetprotokollen wie z. B. HTTP eingesetzt. 7

OSI-Modell Ein Referenzmodell für die Kommunikation in Netzwerken. OSI steht für Open Systems Interconnection. Es beschreibt 7 Schichten, die von den Daten bei der Kommunikation durchlaufen werden. Sie lauten von 1 bis 7: Bitübertragungsschicht, Sicherungsschicht, Vermittlungsschicht, Transportschicht, Sitzungsschicht, Darstellungsschicht und Anwendungsschicht. 6

Packet Capture (PCAP) Eine Schnittstelle zum Mitschneiden von Netzwerkverkehrsdaten. 3

Quantil Eine Messgröße in der Statistik, die einen Schwellwert für die Korrektheit einer Aussage darstellt. Ein bekanntes Beispiel für ein Quantil ist der Median-Wert. Er entspricht dem mittleren Quantil. 11

Quoted-printable Ein Kodierungsverfahren, bei dem Nicht-ASCII-Zeichen durch ein Gleichheitszeichen mit anschließender Hexadezimaler Darstellung des Zeichens ersetzt werden. Andere Zeichen bleiben in ihrer ursprünglichen Darstellung erhalten. 7

TCP-Retransmit Die erneute Zusendung eines TCP-Pakets. Dieser Vorgang kann sowohl explizit, durch erneute Anforderung vom Empfänger, als auch implizit, durch einen Timeout auf Senderseite, angestoßen werden. 5

Thread Pool Pattern Ein Entwurfsmuster für Nebenläufigkeit. Es wird ein Pool bestehend aus einer bestimmten Anzahl von Threads angelegt. Außerdem verwaltet der Thread Pool eine synchronisierte Queue für Aufgaben. Jeder Thread holt sich ständig neue Aufgaben aus der Queue und arbeitet diese ab. 26

Universal Time Coordinated (UTC) Die internationale Basis zur Zeitmessung. Sie wurde 1967 eingeführt und gilt seitdem als Standard. 4

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 3. März 2016 Viktor Winkelmann