JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Alexander Lemmé, BSc

Submitted at
Institute of Networks and Security

Supervisor
Dr. Michael Sonntag

February 2018

Extension of an existing P2P-Client for Evidence Collection

Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Network and Security

Abstract

Collecting evidence within P2P networks has become an important part of modern conservation of evidence. Especially in the area of copyrighted material, the legal situation is very different because of various definitions, what constitutes a copyright infringement. As a rule, however, the "pure" participation in a P2P network is not illegal in any way as long as no copyrighted material is publicly offered and distributed.

Modern solutions, in the form of software or services, usually provide good evidence that is generally accepted by courts. However, some problems can arise with closed systems. These range from badly documented evidence to false accusations only due to the participation of a P2P network.

This master thesis provides a prototype which solves the described problems in an open and comprehensible way. An existing BitTorrent client is expanded with a plugin. Only Open Source software is used in the implementation and all steps of the proofing system are implemented openly and comprehensibly.

The evidence collection is done by downloading parts of files directly from observed targets and make use of shared meta data by the BitTorrent protocol. So public sharing of files can be proven and in addition the download progress of a target can be captured by downloaded parts of files and "available" messages.

Zusammenfassung

Das Sammeln von Beweisen innerhalb von P2P-Netzwerken ist ein wichtiger Bestandteil der modernen Beweissicherung geworden. Gerade im Bereich von urheberrechtlich geschütztem Material, ist die Gesetzeslage aufgrund verschiedenster Definitionen, was ein Urheberrechtsverstoß ist, sehr verschieden. Im Regelfall ist aber die reine Teilnahme an einem P2P-Netzwerk in keinster Weiße illegal, solang kein urheberrechtlich geschütztes Material öffentlich angeboten und verteilt wird.

Moderne Lösungen, in Form von Software oder Dienstleistungen, liefern meist gute Beweise die von Gerichten im Allgemeinen akzeptiert werden. Allerdings können bei geschlossenen Systemen einige Probleme entstehen. Diese reichen von schlecht dokumentierten Beweisen bis zu falschen Beschuldigungen, nur aufgrund der Teilnahme eines P2P-Netzwerkes.

Diese Masterarbeit soll einen Prototyp zur Verfügung stellen, welcher die beschriebenen Probleme auf eine offene und nachvollziehbare Art löst. Dabei wird ein bestehender BitTorrent Client mit einem Plug-In erweitert. Bei der Umsetzung kommt nur Open Source Software zum Einsatz und alle Schritte der Beweissicherung sind offen und nachvollziehbar implementiert.

Zur Beweissicherung werden Teile von Dateien von den überwachten Zielen heruntergeladen und gleichzeitig auch alle Metadaten vom genutzten BitTorrent Protokoll aufgezeichnet. Damit kann einem Ziel nicht nur das öffentliche zur Verfügung stellen von Dateien nachgewiesen werden, sondern auch ein Downloadverlauf des Ziels anhand von den geladenen Daten und sogenannten „available Messages“ erfasst werden.

Contents

List of Figures

List of Tables

Listings

1. Introduction

What is peer-to-peer (P2P) filesharing and why do we need to prove something with evidences? P2P means data is exchanged between peers of a network, without using a server between them. Evidence collection is necessary because filesharing is a significant part of the European aggregate internet traffic and makes up to 8.44%[18] of it. But not all of the traffic is used in a lawful purpose.

1.1. Motivation

Today peer-to-peer(P2P) filesharing is a widely used technology, to exchange large amounts of data. In contrast to a normal download, files will be shared between independent peers, using their own resources. So from one point of view there is no need for the initial uploader to share data and insert resources, because the swarm of peers will provide all data. This is why companies, organizations and also individuals employ this technology to share for example software, updates or distribution images. On the other hand there is no single point where the sharing can be stopped, once it is spread out in the internet. This is problematic for copyright protected files or files with illegal content. To enforce laws on illegal file sharing activities, evidence must be collected. To be used in court, this evidences must meet a certain standard, which are described in section 2.1. Current solutions, described in detail in chapter 3, for this problem are

- inadequate to meet the standard (screen shots/logging from client).
- closed source solutions from companies (the mode of operation is unknown).

To solve to this problem, this thesis will try to offer a prototype implementation of an open and automated solution. This prototype implementation will be discussed in chapter 4.

1.2. Definitions

In order to understand the following sections some terms have to defined.

P2P: Short term for peer-to-peer. P2P in computing or networking describe a scenario where different participants of a system communicate directly which each other, without the need of a third party, like a server, between them.

Peer: A participant of a P2P-filesharing-network.

Download: Receive files, or parts of them, from connected peers.

Upload: Send files, or parts of them, to connected peers.

Shared file: Content files which are shared in a P2P-network.

Piece: Shared files are chopped in small pieces. Each piece can be verified by a hash.

Block: Pieces are split up in blocks. It is the smallest unit transmitted and can not be checked on its own.

BitTorrent-protocol: A specific P2P-filesharing-protocol exchanging blocks between peers.

BitTorrent-client: Software to join a P2P-filesharing-network working with the BitTorrent-protocol. There exists a wide range of BitTorrent-clients, some cover only the basic functions, other provide a large number of additional tools.

Torrent-file: Describes the shared files, like name and length, the pieces with the according hash values, and all including meta-data. They are necessary to communicate with other peers.

Torrent: A torrent in a BitTorrent-client describes the state of a download. This means is it started or stopped, which percentage is done yet, or what parts or files need to be downloaded and which will be skipped.

Swarm: Set of all active peers of one torrent.

Chain of custody: A chronological documentation showing the collection, timing and storage of evidence.

1.3. Objective

The objective of this thesis is a prototype implementation of a software for evidence collection. For simplicity an existing P2P-client should be extended. To guarantee the chain of custody and a high grade of automation there are several requirements.

Download:

- Storage of the complete data traffic (including meta-data).
- Comparison of downloaded data with verified original files of the download via hash value.
- Which files or parts of them where made publicly available?
- History of the files shared by a peer and the progress of a download.
- Prevent extended P2P-client to upload data from shared files.

Evidence:

- Exact documentation including history for the chain of custody.
- Integration of an external time source with time synchronization.
- Possibility to digitally sign the collected and packaged evidences including a thirdparty timestamp.

Features:

- Lookup of IP addresses in Geo-IP databases to exclude them from observation.
- Blacklist/Whitelist for IP addresses.
- Selection for interesting files.
- Simulating normal behavior of an P2P-client.

1.3.1. BitTorrent-protocol

There is a wide selection of various P2P-filesharing-protocols available. Some of them are historic, like the first big P2P-filesharing-network Napster, others are still in use by a significant amount of users, like BitTorrent, eDonkey, Gnutella2, compared in table 1.1 and many more. Current P2P-protocols are decentralized and so independent of a central server. Also they are usually standardized, so many P2P-clients of different vendors exits for these protocols. This thesis will focus on the BitTorrent-protocol.

Due to the wide distribution and the fact that it is often used for illegal filesharing, the BitTorrent protocol is probably the best choice. In addition in this protocol files are not distributed as a whole, but in small pieces. This makes it possible to use upload capacities of distributed users. Thus many peers (some unconsciously) itself become distributors of files, which in some case may be illegal. By taking advantage of these uploads, peers will be known by their IP address if data is downloaded. This information can be used in court, when the upload was illegal. A complete description of the functionality of the BitTorrent-protocol can be found in section 2.2.

	Chunking Sheme	**Peer Finding**	**Hash Distribution**	**File Search**	**Usage**
Gnutella	peer negotiate the chunk size at runtime, 64Kb by default	decentralized	no chunk level hashing, file hash used as ID	yes	-
eDonkey	devided into 9500kB parts, each 53 chunks	hybrid, originally centralized via servers	MD4 hashing at part level	yes	2,59%
BitTorrent	devide file into fixed size pieces (16kB up to >16MB)	hybrid, originally centralized via trackers	SHA hashing at piece level	no	17,36%

Table 1.1.: P2P protocols in comparison [22] and usage 2013 Europe fixed access [17]

2. Theoretical background

This section describes the main terms and technologies which are used in this thesis.

2.1. Collecting evidence

To proof something in court, evidence is needed. To gather useful evidence specific standards must be met.[19]

- **Admissible**: Everything collected has to be done on a legal basis.
- **Authentic**: The evidence should cover a specific incident and nothing else.
- **Complete**: Everything found, collected, or evaluated has to be present in an evidence, not only the damaging or unburdening parts.
- **Reliable**: Collecting, handling, and evaluating should ensure veracity and authenticity ("Chain of Custody").
- **Believable**: Should be comprehensible and understandable by laymen.

2.1.1. Definitions of evidence

There is no general universal definition of evidence but there are several different definitions according to different circumstances and countries. The Austrian law does not have a definition of evidence, it is more of a list of what can be brought up as proof in court:

> *The Austrian concept of an active judge, however, goes along with the judge's duty to do case-management and especially to induce a truthful fact-finding using judicial discretion. While only five means of proof (documents, witnesses, expert opinions, evidence by inspection and the examination of parties) are explicitly listed the Austrian civil procedure code, there is no numerus clausus regarding the means of evidence. Evidence may be freely assessed by the judge.*[15]

In contrast, the Encyclopedia Britannica[1] defines evidence as follows:

> *Evidence, in law, any of the material items or assertions of fact that may be submitted to a competent tribunal as a means of ascertaining the truth of any alleged matter of fact under investigation before it.*[14]

This two examples show how different several definitions could be, even when they are very general definitions of evidence. In this thesis digital evidence will be created and therefore a specific definition is needed. *Eoghan Casey* defines digital evidence in his book *Digital Evidence and Computer Crime* as follows:

> *"Digital evidence is defined as any data stored or transmitted using a computer that support or refute a theory of how an offense occurred or that address critical elements of the offense such as intent or alibi.'*[6]

This definition, combined with the chain of custody, is used as a guiding principle to ensure the quality of the evidence collected within the prototype implementation of this thesis.

2.1.2. Evidence collection in P2P-networks

In computer forensics the evidence collection on P2P-network has to consider several points. Beside the normal requirements on evidence there are some challenges and problems, which have to be solved. This special requirements results out of the unique design of a P2P-network.

[1]https://www.britannica.com/

In contrast to normal client-server connections, in P2P-network a client has to deal with various other peers, and does not even get whole files from a single participant but rather small parts only. In consequence, without having complete data reliable evidence has to be collected. In addition in P2P-networks each peer shares files, so uploading files is part of the sharing, which has to be considered too.

The limitation of the evidence collection in P2P-networks is that it can only provide IP-addresses, port numbers, and other meta data. To match this information to a physical computer without the help of internet service providers can be hard to impossible. To find a responsible human being is even more difficult. So this thesis focuses on gathering digital evidence only, not on identifying real persons or computers.

Uploading and downloading illegal content

In P2P-network files will be shared between different clients. To verify these files for investigation, at least some parts of them, have to be downloaded. It is very important to not upload any copyright protected content as investigator. The problem is that the shared files can be illegal in different ways. If they are copyright protected and not allowed to be shared, uploading them is illegal. According to the judgment of the European Court of Justice the downloading copyright protected files from an illegal source is illegal as well[8]. Or the content itself is criminal, for example child pornography, then the processing of the file above is illegal.

Process evidence

Normally a P2P-network is based on messages sent between clients. To get evidence the gathered messages musts be processed. This can be made on different levels. Either to get parts of files and meta data or to recover whole files and the complete communication. What is necessary depends on the validation mechanism of the protocol and the required information. It is very important to stick to the protocol standard, to guarantee a correct and reliable evidence.

Validation of evidence

All data collected as evidence has to be validated in one or another way. It is important that this is done on the very first level, because if an error is found on a later processing or evaluating level, the evidence gets useless. In P2P-networks there are usually two types to validate:

- **Peer**: Is the peer the one who participates in a P2P-network? Normally peers are identified by their IP-address and port number, but a peer can hide this information behind a proxy, for example *TOR*[2].
- **Data**: Can a piece of data be assigned to a file? Mostly there are protocol mechanism to verify parts of a shared files by hash values. If not, at least the complete downloaded file has to be validated.

Indirect/Direct monitoring

Monitoring a P2P-network can be done in two ways. Which way to choose depends on what should be investigated.

- **Indirect monitoring**: This method means that a monitor passively joins a swarm of a P2P-network and gathers information about other peers in the swarm. A peer in a swarm shares different information about itself, like IP addresses or port numbers, depending of the protocol, to connect to it. This information is processed and evaluated. The problem with indirect monitoring is, that you cannot prove the information you get.
- **Direct monitoring**: This means that a monitor actively joins a swarm. In addition to the information of the indirect monitoring, actual network traffic is available. A monitor acts like a normal client except that it stores the whole communications with other peers, so shared files can be compared with their originals and attributed to specific peers. It is an open legal question if the whole file is needed for comparison or verifiable parts are sufficient.

[2]www.torproject.org

Processing of collected data

Just collecting traffic and meta data in P2P-networks is no evidence at all. To prove a specific crime the collected information has to be processed. In case of file sharing, the transferred messages are usually split up in small chunks and therefore has to be reassembled into pieces, which can be verified. Also meta data of one message on its own has no special value until all of them are combined and investigated for conclusions.

Results and evidentiary value

As a result of evidence collection in P2P-networks there is a proof that a specific peer, identified by IP address and port numbers at a concrete time, had a connection to the observing party. Also by evaluating the content of the connection a public sharing of even small parts of files can be verified. This implies that if a target shares a file, it is, at least partly, stored on it. This can be concluded from a list of sent and received messages, as well as the recovered files. In addition the evaluation of the meta data can be used as circumstantial evidence. If done right, the chain of custody was applied in the collection process.

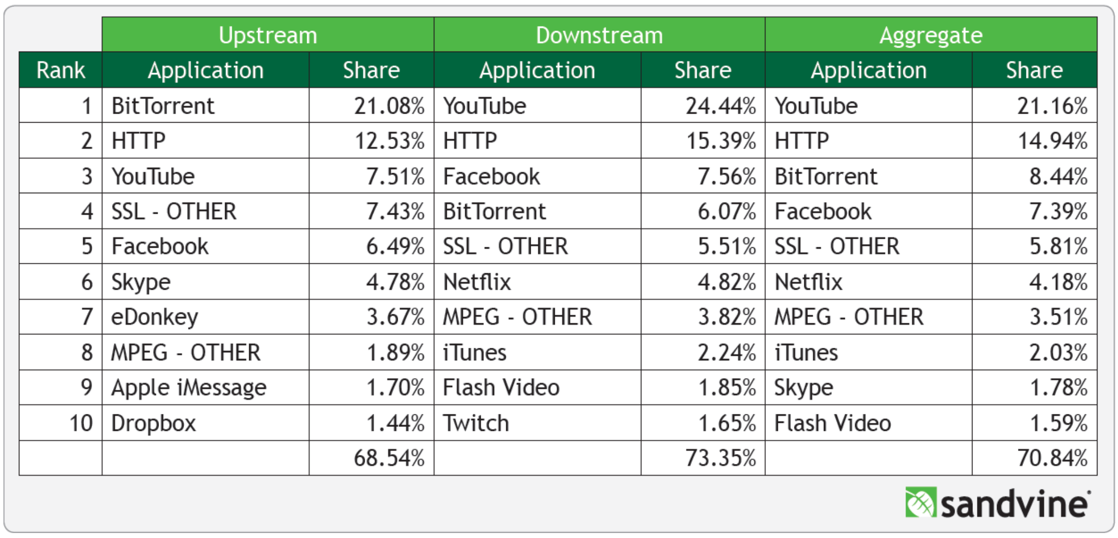
In the end there is still the problem that a IP address has to be matched to a natural person or corporate entity. As most of the IP addresses are assigned dynamically, the institution behind the IP at the given time has to be found. But even when the computer or network device in question is found, the actual user of it has to be identified.

2.2. BitTorrent

This section describes the BitTorrent-protocol on a level that is needed for this thesis. It contains information about the history, but focuses on the functionality of BitTorrent.

2.2.1. History

The BitTorrent protocol was invented by Bram Cohen in April 2001. It was originally developed for the bootlegger-online-community etree[3]. The challenge was to share big files over the internet without the use of expensive, centralized file servers. In July 2001 the first implementation was published. The protocol itself got final in the year 2008. Nowadays it is the biggest P2P-filesharing-protocol and was in 2015 responsible for 8.44% fixed access internet traffic, shown in figure 2.1.



	Upstream		Downstream		Aggregate	
Rank	Application	Share	Application	Share	Application	Share
1	BitTorrent	21.08%	YouTube	24.44%	YouTube	21.16%
2	HTTP	12.53%	HTTP	15.39%	HTTP	14.94%
3	YouTube	7.51%	Facebook	7.56%	BitTorrent	8.44%
4	SSL - OTHER	7.43%	BitTorrent	6.07%	Facebook	7.39%
5	Facebook	6.49%	SSL - OTHER	5.51%	SSL - OTHER	5.81%
6	Skype	4.78%	Netflix	4.82%	Netflix	4.18%
7	eDonkey	3.67%	MPEG - OTHER	3.82%	MPEG - OTHER	3.51%
8	MPEG - OTHER	1.89%	iTunes	2.24%	iTunes	2.03%
9	Apple iMessage	1.70%	Flash Video	1.85%	Skype	1.78%
10	Dropbox	1.44%	Twitch	1.65%	Flash Video	1.59%
		68.54%		73.35%		70.84%

Figure 2.1.: Top 10 Peak Period Applications - Europe, Fixed Access 2015[18]

[3]www.etree.org

2.2.2. Functionality

The BitTorrent-protocol is designed to share big files in small pieces with many other peers. Unlike other P2P-protocols that enable users to search and share different files, BitTorrent does not create a network for all files. For every download, which can contain an arbitrary number of files, a swarm is created, where only the specified files are provided. Peers start without any pieces of the shared files. After time they download more and more pieces, but also upload already downloaded pieces to other peers. In BitTorrent-swarms these peers are called **leecher**. When a peer has finished a download, or is the origin of the torrent, it is called a **seeder**. A *seeder* uploads shared data as long as it stays in the swarm. An overview of the whole BitTorrent functionality is shown in figure 2.2[3].

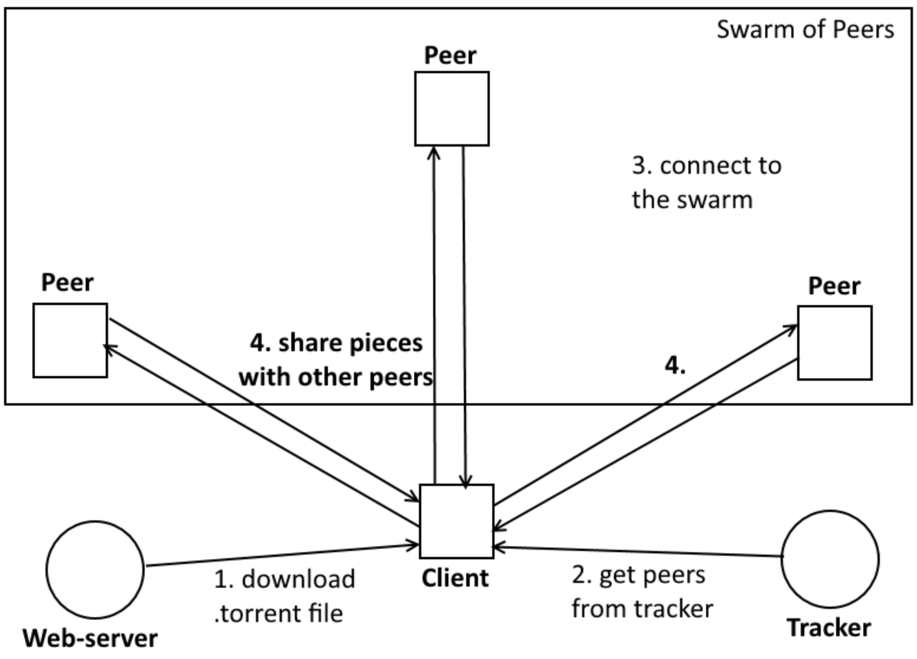

Swarm of Peers
Peer
3. connect to the swarm
Peer
4. share pieces with other peers
4.
Peer
Client
1. download .torrent file
2. get peers from tracker
Web-server
Tracker

Figure 2.2.: BitTorrent Procedure[12]

2.2.3. Data structure

The BitTorrent-protocol shares big files in small pieces, so there is a need of a data structure. Files of one torrent are split up in pieces of fixed size (except the last one) that each can be verified with a SHA1 hash. This pieces are still too long, so when shared with BitTorrent, a piece is split up into several blocks of fixed size. The intersection points of the pieces do not consider different files, all files are treated as one pile of binary data. Figure 2.3 shows a torrent, which consists of six files, split up in 16 pieces, each of 4 blocks. Also the borders of the files do not line up with the one from the pieces.

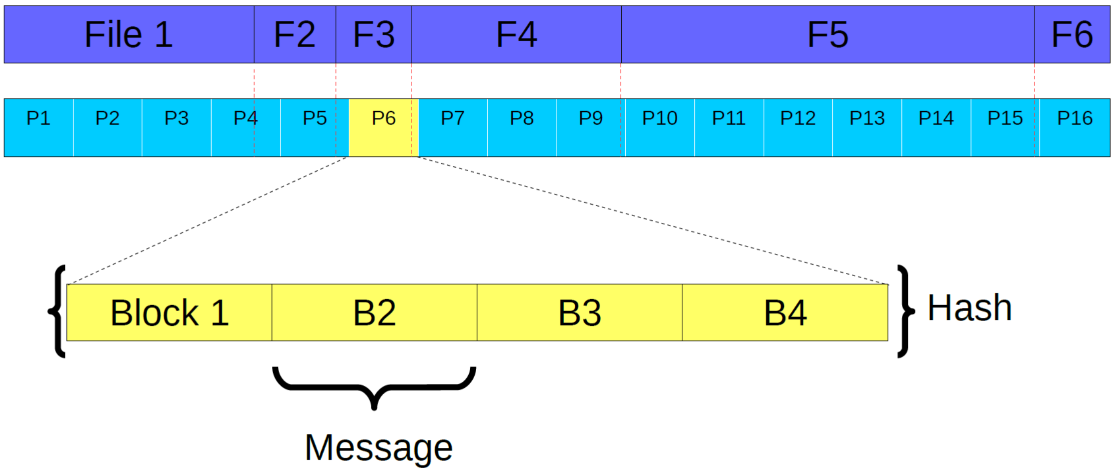


Figure 2.3.: A torrent is divided in files, pieces and blocks

2.2.4. .torrent file content

The .torrent file provides all necessary information to join a swarm. Usually this file is downloaded from the web and opened in a BitTorrent-client which handles the download. All data in a .torrent file is a dictionary, containing at least the keys listed below.

- **announce** contains the announce URL of the tracker
- **info** dictionary that describes the shared files in this torrent
  - **piece length** number of bytes in each piece
  - **pieces** concatenation of 20-byte SHA1 hash value of each piece
  - **name** filename, dictionary if more than one file
  - **length** filelength, dictionary if more than one file

2.2.5. Tracker

The tracker is a web service that provides lists of peers and statistics about a torrent, but is not directly involved in the data transfer. It responds to HTTP-GET requests which contains all information to connect to the swarm of this torrent, like a list of peers to start sharing data. The necessary information for the tracker, to help peers find each other, are the following:

- **info_hash** 20-byte SHA1 hash value of the info dictionary in the .torrent file.
- **peer_id** arbitrary 20-byte string, there are no guidelines for creation.
- **port** the client is listening on.

The tracker's HTTP-response is of type "text/plain" and contains a dictionary with a lit of peers and other information about the health and the state of the torrent. The important fields of this dictionary are the following:

- **peers** dictionary or binary encoded list of peers
  - **peer_id** self selected 20-byte string identifies a peer
  - **ip** peer's IP-address as string, could be IPv4(dotted quad), IPv6(hexed) or even a DNS-name
  - **port** peer's port number as integer

- **complete** number of seeders as integer
- **incomplete** number of leechers as integer

2.2.6. Peer Protocol

The peer protocol is a TCP based protocol for the exchange of pieces which are listed in the .torrent file. A connection is initiated via a handshake and then continued with the exchange of BitTorrent-messages. Beside the actual data exchange, the peer protocol also maintains the state, described in section 2.2.6, and meta information, like available pieces, of a torrent.

Handshake

Every connection between two peers starts with an initial handshake. The initiator of a connection first sends their handshake. The recipient checks via the info hash if it can serve the requested torrent. If so, he responds with its own handshake and a connection is established. The handshake contains following information:

- **pstrlen** single byte, containing the string length of **pstr**.
- **pstr** string identifier of the protocol ("BitTorrent protocol").
- **reserved** 8-bytes, can be used to change the behavior of the protocol. In current implementations all zeroes.
- **info_hash** 20-byte SHA1 hash value of the info dictionary in the .torrent file.
- **peer_id** arbitrary 20-byte string, usually the same that is announced to the tracker.

Messages

The BitTorrent-protocol is implemented with unique messages. With these messages, shown in table 2.1, the connection between two peers is maintained and the data exchange takes place.

ID	Name	Description
-	**keep-alive**	Keeps the connection open, because a connection will be dropped after a certain period of time without activity.
0	**choke**	Choke a connected peer.
1	**unchoke**	Unchoke a connected peer.
2	**interested**	Register interest of pieces from a connected peer.
3	**not interested**	Cancel interest of pieces from a connected peer.
4	**have**	Inform other peers a piece has been successfully downloaded and verified via the hash.
5	**bitfield**	Sent after the handshaking sequence is completed, downloaded pieces are indicated with a 1.
6	**request**	Request one or more blocks of a piece.
7	**piece**	Contains the actual data of one or more blocks.
8	**cancel**	Cancel a block request.

Table 2.1.: Messages of the BitTorrent-protocol

State

A peer must maintain state information for each connection that it has with other peers. There are three states:

- **NOT_INTERESTED**: Initial idle state.
- **INTERESTED&CHOKED**: Client registered interested in pieces of a remote peer but waits on unchoke message from them.
- **INTERESTED&UNCHOKED**: Client can request and download pieces from the remote peer.

Initially a client is in *NOT_INTERESTED* state. A client has to send an *interested* message to a remote peer to register interest in pieces. He is now *INTERESTED&CHOKED*. If the remote peer is willing to communicate, he sends an *unchoke* message. The client now is *INTERESTED&UNCHOKED* and can send *requests* for pieces until either the remote peer *chokes* the client again or the client has lost interest (for example he finished a download). A full state diagram is shown in figure 2.4.

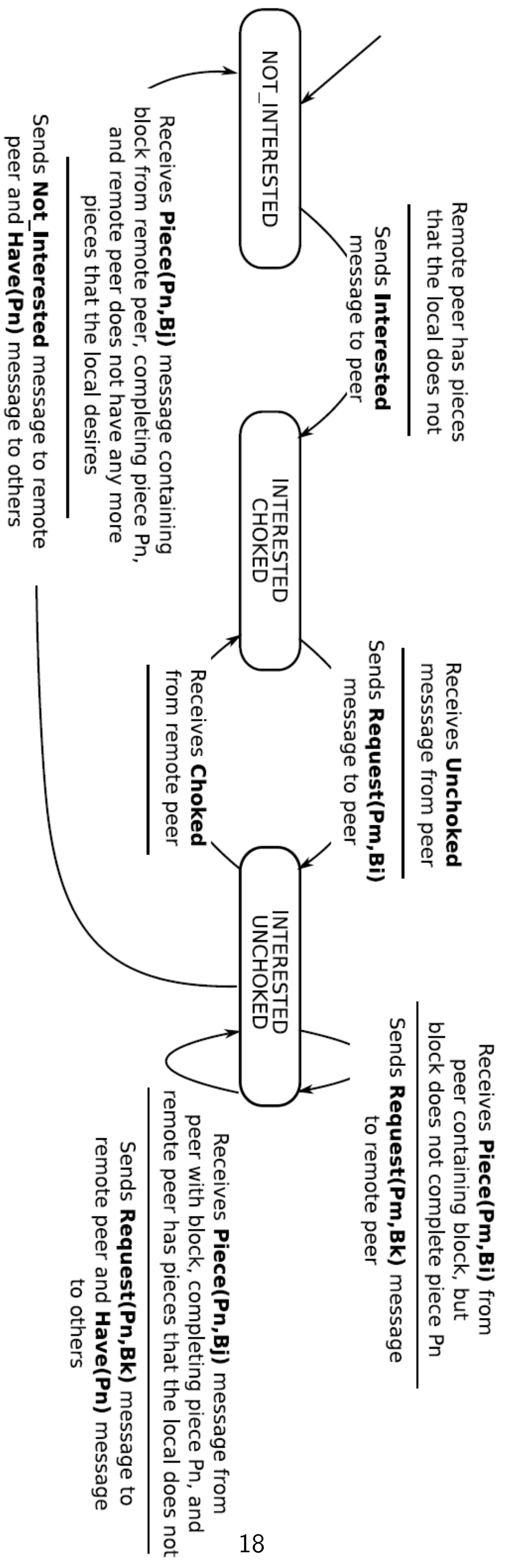

NOT_INTERESTED
Remote peer has pieces that the local does not
Sends **Interested** message to peer
INTERESTED CHOKED
Receives **Piece(Pn,Bj)** message containing block from remote peer, completing piece Pn, and remote peer does not have any more pieces that the local desires
Sends **Not_Interested** message to remote peer and **Have(Pn)** message to others
Receives **Unchoked** messsage from peer
Sends **Request(Pm,Bi)** message to peer
Receives **Choked** from remote peer
INTERESTED UNCHOKED
Receives **Piece(Pm,Bi)** from peer containing block, but block does not complete piece Pn
Sends **Request(Pm,Bk)** message to remote peer
Receives **Piece(Pn,Bj)** message from peer with block, completing piece Pn, and remote peer has pieces that the local does not
Sends **Request(Pn,Bk)** message to remote peer and **Have(Pn)** message to others

Figure 2.4.: State diagrams for connection regarding downloads from remote peers[13]

3. Current Solutions

This chapter shows an overview of existing approaches to collect evidence. The differences and problems of this approaches will be shown and discussed. It also presents some exemplary cases with reference to the approaches.

3.1. Existing approaches

Evidence collection could be done in various ways. Some of them are very cheap and simple and result in circumstantial evidence which relies on witnesses or other evidence. Other approaches are able to prove actions so this evidence can be seen as a fact, like plain network communication. But these approaches are more complex and cost more money. In today's evidence collection three main approaches can be found.

3.1.1. Screenshots

The simplest and easiest solution is to make screenshots during the operation of a BitTorrent-client. Most clients can show all necessary information to a user like IP-addresses, the shared files and the process of uploading pieces of a file. One big problem with this approach is that there is no link to the actual data transferred by the client. For example uploading a readme-file as part of a torrent is nothing special and does not interfere with the copyright of a music track but cannot be excluded via a screenshot. In addition screenshots always show only parts of a full picture, there can be scroll bars, different tabs with information or they show only information a user wanted to be visible on them. Another big problem with screenshots as evidence is that it is a simple picture, which can be tampered very easily. Even worse, as long as it shows only a computer screen, it can be easily constructed with software out of nowhere. The problem becomes even worse when screenshots are printed out as evidence for use in court, because there is no chance to identify tampering.

3.1.2. Client-logging

Another approach is to use the logging ability of some BitTorrent-clients. This could generate really useful evidence, because the information is provided by the client software itself. Theoretically, all information is available and written to a log file. In reality the logs of a BitTorrent-client exist for developers and not for the creation of evidence which can be used in court. So normally those logs shows too little and/or the wrong information to prove a specific action on a torrent.

3.1.3. Specialized software and services

Like the software developed in this thesis there exists specialized software to extract evidence from BitTorrent-traffic. These programs or services are provided by companies specialized on evidence collection. When doing correct and comprehensible evidence collection, there is no problem at all with such kind of service. The problem is that most companies and organizations do not disclose their process and therefore it is difficult to understand how evidence was collected. It is therefore often the interpretation of the court whether a "proof" is accepted. Although most companies are acting with conscience, several cases have happened in the past where companies have unconsciously collected false evidence, like in the example case described in section 3.2.3. In the worst case, some companies even partially created it themselves, which happened in 2014 in the RedTube copyright infringement affair in Germany[5]. Also, semi-legal methods such as honeypots with real content were used, like 2007 where a anti piracy gang set up their own video download portal to trap people into uploading copyright protected material[1].

3.2. Exemplary lawsuits

This section introduces three different cases from Germany to show the complexity of evidence collection. It covers a lawsuit as positive and one as negative example for the use of screenshots, and one case that points out the possible fallibility of closed-source solutions.

3.2.1. Screenshots as failure

LG Hamburg, Judgment from 14.03.2008 - Az. 308 O 76/07[10]

Summary: ”The subject-matter of the proceedings is an objection by the plaintiff to the defendant on the basis of the public access to two music recordings of a popular German artist group in a file sharing system via the Internet connection of the defendants.

The applicant claims that he has the exclusive right to exploit the disputed music recordings. On 11.07.2006 at 17:10:13 (CEST) at the found IP address altogether 170 audio files were made available for download by means of a file sharing software, based on a P2P-protocol, including files with the disputed music recordings. The IP address had been assigned to the defendant at the time of the contested date. The applicant did not allow such use of his recordings.”[7]

The case was dismissed.

Excerpt from the decision reasons: ”Although the applicant has claimed that the company's online investigators have determined that the literal title of the controversial music has been made available for download over the period in question, and has provided proofs of the company, he has also submitted a public prosecutor's information to the effect that this IP address has been assigned to the defendant during the period in question. The prints produced by the company itself are, however, not a suitable means for the proper conduct of the investigations. The head of the investigation service, who had been appointed by the applicant as a witness for the investigation, could not say anything about the investigations. Rather, he merely said that the investigations had been made by a student who was now living in L. again. He then presented the results of his investigations and checked the results on the screen for plausibility. He had not been part of the investigation himself and he had not listened to the music files."[7]

Annotation: This case is a negative example how to do computer forensic. Printouts of screenshots and logs provided by the applicant can not be used as evidence without a witness that can prove its reliability. The fact that such a witness who collected the evidences was not available in court and could not explain his approach on his own lead to the rejection of the case.

3.2.2. Screenshots as success

LG Köln, Judgment from 31.10.2012 - Az. 28 O 306/11[11]

Summary: "The applicants are among the leading German phonogram producers. They are the owners of exclusive rights to numerous pieces of music. In so-called online file-sharing-software music pieces as MP3 files are offered by the respective parties for download. In this way, every user of the exchange can download pieces of music from the computers of the offerer. The applicants thus suffer substantial damage annually.

The applicants have therefore commissioned a company to investigate such copyright infringements. These determined that a total of 5080 audio files were made available for download via an internet connection, which could be traced to the defendant, based on the IP address, by means of the exchange software software BearShare."[2]

The defendant is convicted.

Excerpt from the decision reasons: "The applicants have shown with the help of the screenshots and the testimony of the witnesses that on 19.08.2007 at 11:12:31 over an internet connection, at that time the found IP address was assigned, which from the screenshots with the names of music tracks have been made publicly available.... There are no indications of any doubts as to the correctness of the classification. In the board's view, it is clear from the findings of the evidence that the infringement originated from the Internet connection of the defendant."[2]

Annotation: The applicants succeeded in court even with only screenshots as evidence. In contrast to the case described before, the applicants could show the full process of evidence collection, including witnesses. This shows how important it is how to collect evidence and substantiate it with comprehensibility and witnesses.

3.2.3. False-Positives

LG Berlin, Judgment from 03.05.2011, Az. 16 O 55/11[9]

Summary: "In this present case, LG Berlin had to decide on the legal dispute between an investigator of IP addresses on the one hand and the company issuing file-sharing warnings on the basis of such determined IP addresses. It became clear that the complaining company had worked unreliably when determining IP addresses."[16]

The case was actually not about the question if the investigator provided reliable evidence itself, but about an injunction. The applicants argue that the defendant contacted the applicant's contract partners and attempted to persuade them to terminate their contractual relationship with the applicant based on untrue facts and on secret inhouse information. These allegedly untrue facts about the reliability about the findings of IP addresses turned out to be true.[9]

Annotation: The interesting part of this case is not the verdict itself but the admission of the applicant that can be found in it. It became clear that the applicant's company had worked unreliably when determining IP addresses. In fact, they produced false positives which must not happen ever.

4. Prototype implementation

This chapter describes the prototype implementation. It is structured into the requirements for the implementation, the choice of the extendable BitTorrent-client with its plugin system and the structure and workflow of the prototype. Furthermore there is a short section about the viewer for the collected evidence.

4.1. Requirements

In this section the objectives of this thesis, shown in section 1.3, are transformed into specific requirements for the implementation of a prototype application, regarding the choice of an extensible BitTorrent-client as well as technical specification to deploy the chain of custody.

4.1.1. Extensible BitTorrent-client

The prerequisites by which an BitTorrent-client is selected are listed in the following list:

- **Open source**: To provide open software, the extended client has to be open source software as well. With this precondition it is guaranteed that the whole process of evidence collection is reproducible and can be owned. In addition the whole source code is available and can be customized for own needs.
- **Full BitTorrent functionality**: To get full advantage of the BitTorrent-protocol

and its powerful extensions it is required that the extended client supports and gives access to at least the official extensions described in the accepted BEPs.

- **Up to date & Ongoing development**: To have access to the latest function and security updates the client has to be in ongoing development. So a widely used, up to date client is preferred. To keep up with the latest versions there should be an easy way to patch the application.

4.1.2. Data traffic

The data traffic between the extended client and other peers is the only source for evidence. To extract good and reasonable evidence the client has to implement the following requirements:

- **Storage of complete data traffic**: In order to obtain the most accurate and comprehensible results possible, all traffic must be stored including the shared downloaded data as well as protocol traffic.
- **Verification of shared data**: To guarantee the consistency and completeness of the recorded shared data, a comparison of the data downloaded with the original file has to be performed, for example based on hash values.
- **Availability & history of shared data**: To keep track of the progress of a download at a peer, it is required to store the files made available and the completeness of a download over time provided by the BitTorrent-protocol.
- **Prevent data upload**: To not run the risk of providing shared data on its own, the extended client has to prevent upload of non protocol data.

4.1.3. Evidence

The collection of evidence is the main objective of this thesis, so it is important to extract and handle evidence with care. Evidences have to fulfill some specific standards described in chapter 2.1. To achieve these standards the following requirements apply to the extended client:

- **Reliable time source**: To be independent of the operating system the extended client is running on and its system-clock an integration of an external time source is required. Accurate timestamps are necessary to identify physical computers behind dynamically assigned IP addresses through their internet service providers.
- **Evidence documentation**: The collected evidence has to fulfill the chain of custody and therefore all processed data, actions on them, and all derived meta data have to be documented.
- **Evidence extraction**: The extraction of evidence based on the stored traffic has to be reasonable and reproducible. Only hard evidence are extracted, there must not be guessing or possibilities mentioned as evidence.
- **Tamperproof evidence**: To prevent any tampering with the results there must be the possibility to digitally sign the collected and packaged evidence including a timestamp.

4.1.4. Features

To automate the process of evidence collection and make the extended client easier to use there are some additional non-functional requirements:

- **Geolocation**: Lookup IP addresses in GEO-IP databases to exclude them from observation. This is required to collect evidence only from IP addresses within legal access. Countries to observe can be selected independently.
- **Blacklist/Whitelist**: Exclude or include IP addresses from observation. This filters have to be manually configurable.
- **Tor filter**: Exclude Tor-exit-nodes from observation, because there is no useful evidence to collect.

- **File selection**: Observe only files in a specific torrent interesting for evidence collection.
- **Do not attract attention**: Act as normal BitTorrent-client which supports all expected functionality, except upload of shared files.

4.2. Vuze

Vuze(formerly known as Azureus) is a widely used BitTorrent-client. The original software Azureus was first released in 2003 on SourceForge.net as playground for the Standard Widget Toolkit (SWT), a graphical widget toolkit written in Java maintained by the Eclipse foundation. Although some feature parts of the Vuze has a proprietary license the core is released under GNU General Public License, Version 2[4] and therefore open source.[21]

For the extendable BitTorrent-client Vuze is the client of choice because it is open source and because of its powerful plugin system described in the next section.

4.2.1. Vuze plugin system

The BitTorrent-client Vuze provides a plugin system to extend the normal BitTorrent functionality. A plugin is a Java application which can interact with Vuze with the Plugin-API and is otherwise only limited to the possibilities of Java. Plugins are provided from Vuze itself as well as from the community. Some of the main functions are implemented with plugins like the official DHT implementation. There is also support to write own plugins and make them public available.

The Plugin-API can be used to add UI elements to existing views or create views on their own. But even more important, it is also possible to interact with the underlying BitTorrent core. This interaction is provided via listeners and resource managers. This interface even provides methods to manipulate the actual BitTorrent-messages. This

[4]www.gnu.org/licenses/gpl-2.0.html

functionality is needed to reach the requirements described in section 4.1.

To develop a Vuze plugin the *pluginapi.jar*, provided at `dev.vuze.com/`, is needed and the *Plugin* interface has to be implemented. The plugin system provides optional internationalization as well, in forms of *.properties* files for each supported language. To deploy a plugin to Vuze it has to be deployed as a JAR file including a *plugin.properties* file which defines it as Vuze plugin. This file contains all information for Vuze to load and identify the JAR file as a plugin:

- **plugin.class**: Class containing the *Plugin* interface
- **plugin.id**: Internal representation of the plugin
- **plugin.name**: User readable name of the plugin
- **plugin.version**: The version of the plugin
- **plugin.langfile**: Location of the language files

This plugin can now being installed in an arbitrary Vuze instance supporting at least the version of the *pluginapi.jar* used in the plugin. A plugin can be installed for all users or only for individual ones.[4]

PluginInterface

The entry point for a plugin is the *PluginInterface*, not to confuse with the *Plugin* interface. Plugins only need to implement one particular method, shown in listing 4.1 to get the *PluginInterface*, which provides access to all components of Vuze, like user interface, downloads or networking.[4]

Listing 4.1: Plugin.java[4]

```java
public interface Plugin {
    void initialize(PluginInterface plugin_interface)
            throws PluginException;
}
```

Types of running code

There are three types of code that plugins can run:

- Code executed on plugin initialization.
- Code executed by listener methods.
- Code running in a separate thread.

This means, that all code after initialization is triggered by listeners or is executed in a separate thread. Threading has to be handle with care, because the user interface and the Vuze core are running on different threads. Adding threads without syncronization can lead to unexpected behavior.[4]

4.3. Process of evidence collection

The implemented process of evidence collection results in several steps derived from the requirements. These steps are split up in functional parts in therms of what is implemented within the prototype beside the normal BitTorrent-functionality. This section describes the overall process a user will experience, referring to the subsections where the actual implementation of the parts are described.

An overview of the processing of evidence in this implementation is shown in figure 4.1 but is also described in more details as follows. The first step in this process of evidence collection is the **1.time synchronization**. It is triggered at application startup, to have reliable timestamps when evidence collection starts. In order to start the actual process a torrent has to be added to the client like usual. Within the application the added torrent starts to look up peers and provides them for **2.target picking**. Targets are selected by various attributes, like country or specific IP-ranges. When selected as target, the peer is added to the referring torrent in the application. The target is now treated as a normal peer and is in the state of **3.target observation**. So the target interacts normally with the client and additionally all activities will be recorded by the application. After collecting enough data, specified by some criteria, the target is dropped as a peer in the client. The collected data of the target is then processed in the step of **4.extract & store evidence**. In this step evidence is extracted from the raw traffic data, and also meta data is processed to get a better picture of the target. To fulfill the chain of custody the collected evidence is digitally signed to render it tamper proof.

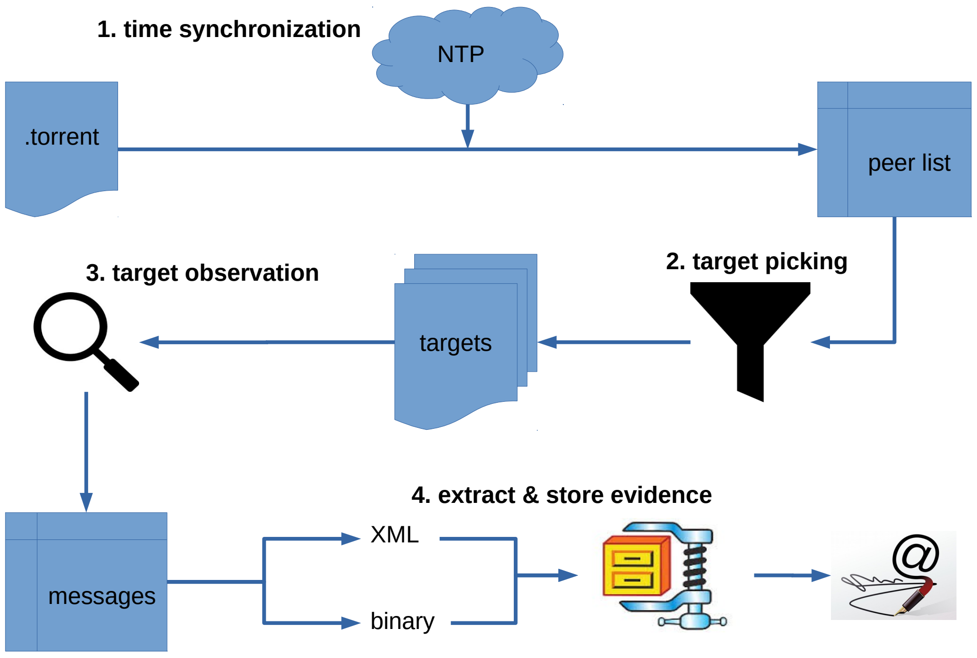


Figure 4.1.: Process of evidence collection

4.3.1. Time synchronization

The local system clock of a computer can be set to an arbitrary value. So this clock is not reliable and does not fulfill the requirements for automatic documentation of evidence. To get trustworthy timestamps and timings the application supports the

Simple Network Time Protocol (SNTP) specified in the RFC4330[5]. SNTP allows to synchronize local clocks over a network with several specific sources.

To get an exact, system independent timestamps the application implements timing as follows:

1. A time source has to be selected in the setting, shown in figure 4.7
2. On initialization or refresh of the time source a request is sent to the specified NTP-timeserver.
3. With the response the local time-offset is computed.
4. This local time-offset is stored and added to every timestamp created within the plugin.
5. After a configurable timeout the time source will be refreshed.

As a remark, SNTP does not recognize any time zones, instead it works only with Coordinated Universal Time (UTC). The presentation of the time in the correct timezone is up to the underlying operating system the application is running on. Therefore it doesn't matter which timeserver is picked, as long as it is reliable and trustworthy. In consequence the time-correcting offset works timezone independent and can be used on any UTC based operating system.

4.3.2. Target picking

First of all, a source for targets is needed, so the torrent selected for observation has to be added to the application. Since the administration of torrents is still a task of Vuze, the observed torrent is added as usual by the client. So all kinds of torrent sources, like magnet links, are supported as long as Vuze can handle them. When the observed torrent is added, it appears in the list of available torrents in the start screen of the plugin, shown on the left in figure 4.2. Before starting the process of evidence collection there is the possibility to select the interested files of the torrent.

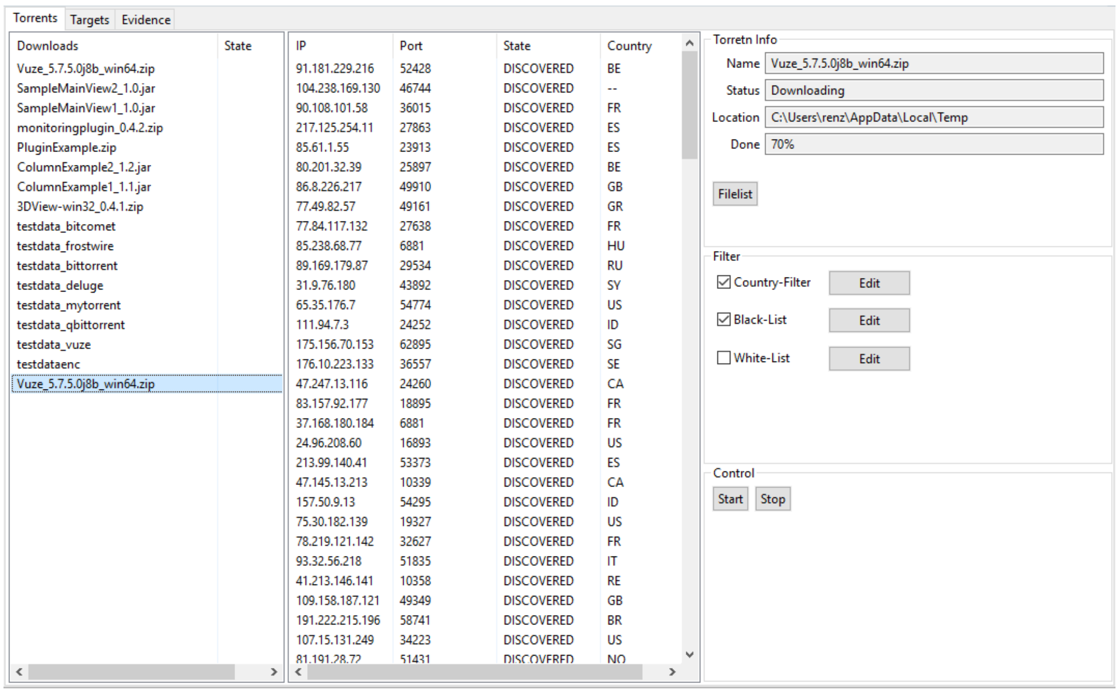
[5]tools.ietf.org/html/rfc4330

When the torrent is started, the *PeerManger* of Vuze starts to collect peers from all available sources. This sources, like trackers or DHT, are not restricted in any way and are configurable in Vuze as a normal functionality of a BitTorrent-client. There is one *PeerManger* per torrent, which handles the discovering as well as the management of the peers. The usual result of a discovered peer would be that it is added to the torrent immediately. This is the first point where the prototype intercepts the usual procedure in the BitTorrent-protocol. The peers, added to the observed torrent, should meet certain criteria and have to be filtered before added to the torrent as targets. Therefore the application listens on *PEER_DISCOVERED*-events from the PeerManager, which are thrown every time a peer is discovered by a specific source. This means a peer can be discovered multiple times, between different sources. A newly discovered peer is added to the peer list available for target picking, shown in the center of figure 4.2. Only if a peer from this list passes the filters described below, or is selected by hand it is added as a target to the system. To prevent Vuze from adding newly discovered peers as targets to the torrent the application also listens on *PEER_ADDED*-events and intercepts additions of peers not selected as target. This filtering and managing of targets has to be done beside Vuze within the application-plugin and is located in the class *PeerFetcher*. It handles all needed listeners on the PeerManager and the storing of the discovered peers.

Filters

By connecting to a BitTorrent swarm the application gets supplied with possible peers via various sources, like a tracker, or DHT. Manually picking all interesting peers as a target would require a huge effort, so there is the possibility to automatically include or exclude peers as a targets by generalized criteria.

This mechanism is implemented in the listener of *PEER_DISCOVERED*-events. When a new peer is discovered it will be immediately selected as a target if it passes all the enabled filters. Since this filtering process depends on the discovery time of a peer, it could happen that when filters are changed during runtime, a peer that would be previously selected as target, will now not be selected after such a change. So there is no retroactive selection. The available filters are:



Downloads	State
Vuze_5.7.5.0j8b_win64.zip	
SampleMainView2_1.0.jar	
SampleMainView1_1.0.jar	
monitoringplugin_0.4.2.zip	
PluginExample.zip	
ColumnExample2_1.2.jar	
ColumnExample1_1.1.jar	
3DView-win32_0.4.1.zip	
testdata_bitcomet	
testdata_frostwire	
testdata_bittorrent	
testdata_deluge	
testdata_mytorrent	
testdata_qbittorrent	
testdata_vuze	
testdataenc	
Vuze_5.7.5.0j8b_win64.zip	

IP	Port	State	Country
91.181.229.216	52428	DISCOVERED	BE
104.238.169.130	46744	DISCOVERED	--
90.108.101.58	36015	DISCOVERED	FR
217.125.254.11	27863	DISCOVERED	ES
85.61.1.55	23913	DISCOVERED	ES
80.201.32.39	25897	DISCOVERED	BE
86.8.226.217	49910	DISCOVERED	GB
77.49.82.57	49161	DISCOVERED	GR
77.84.117.132	27638	DISCOVERED	FR
85.238.68.77	6881	DISCOVERED	HU
89.169.179.87	29534	DISCOVERED	RU
31.9.76.180	43892	DISCOVERED	SY
65.35.176.7	54774	DISCOVERED	US
111.94.7.3	24252	DISCOVERED	ID
175.156.70.153	62895	DISCOVERED	SG
176.10.223.133	36557	DISCOVERED	SE
47.247.13.116	24260	DISCOVERED	CA
83.157.92.177	18895	DISCOVERED	FR
37.168.180.184	6881	DISCOVERED	FR
24.96.208.60	16893	DISCOVERED	US
213.99.140.41	53373	DISCOVERED	ES
47.145.13.213	10339	DISCOVERED	CA
157.50.9.13	54295	DISCOVERED	ID
75.30.182.139	19327	DISCOVERED	US
78.219.121.142	32627	DISCOVERED	FR
93.32.56.218	51835	DISCOVERED	IT
41.213.146.141	10358	DISCOVERED	RE
109.158.187.121	49349	DISCOVERED	GB
191.222.215.196	58741	DISCOVERED	BR
107.15.131.249	34223	DISCOVERED	US
81.191.28.72	51431	DISCOVERED	NO

Torretn Info

Name	Vuze_5.7.5.0j8b_win64.zip
Status	Downloading
Location	C:\Users\renz\AppData\Local\Temp
Done	70%

Filter

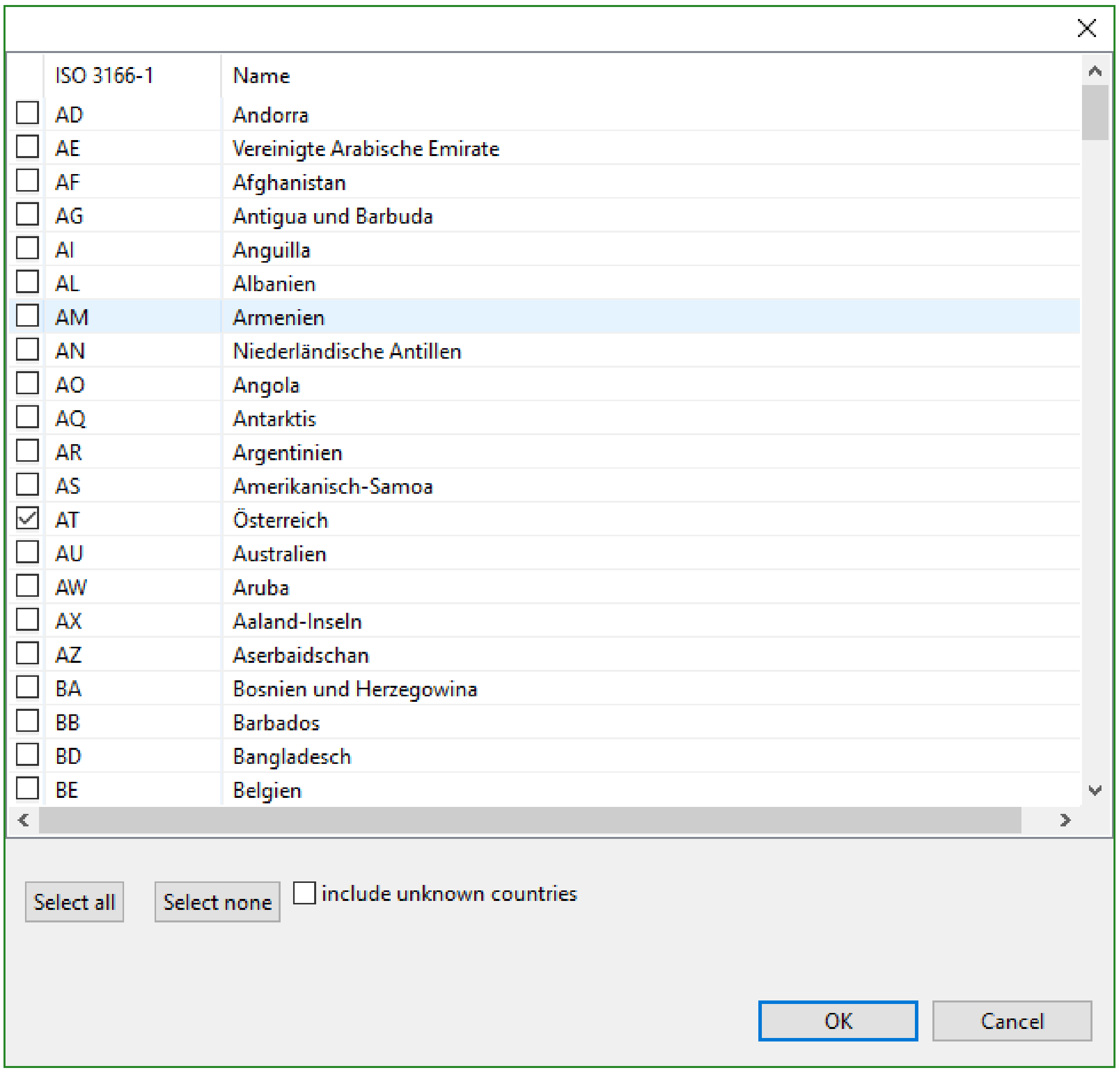
- [x] Country-Filter
- [x] Black-List
- [ ] White-List

Control

Figure 4.2.: Start screen of the prototype

- **Country filter**: This filter, shown in figure 4.3, selects only peers where the IP address can be matched to a given country based on a geolocation service. The filter can be applied to multiple countries at once. IPs which can not be matched to a specific country can be either included or excluded automatically.
- **Blacklist filter**: With this filter peers with specific IP addresses can be explicitly excluded as targets. So local or already covered addresses can be excluded.
- **Whitelist filter**: With this filter specific IP addresses can be explicitly included as targets. So a specific target can be further investigated.
- **Tor filter**: As a filter activated by default, it excludes all Tor exit nodes addresses since no useful evidence can be expected at all. A list of all active Tor exit nodes is provided directly from the Tor project at `check.torproject.org/exit-addresses`. This list is fetched periodically in a configurable period of time, see figure 4.7, and discovered peers can be matched against it.

With these filters configured and enabled the target picking takes place automatically, nevertheless the whole peer list is visible to the user and he still can pick targets manually bypassing the filters. If a target is picked, it is added to the according torrent as a peer and the peer observation starts.



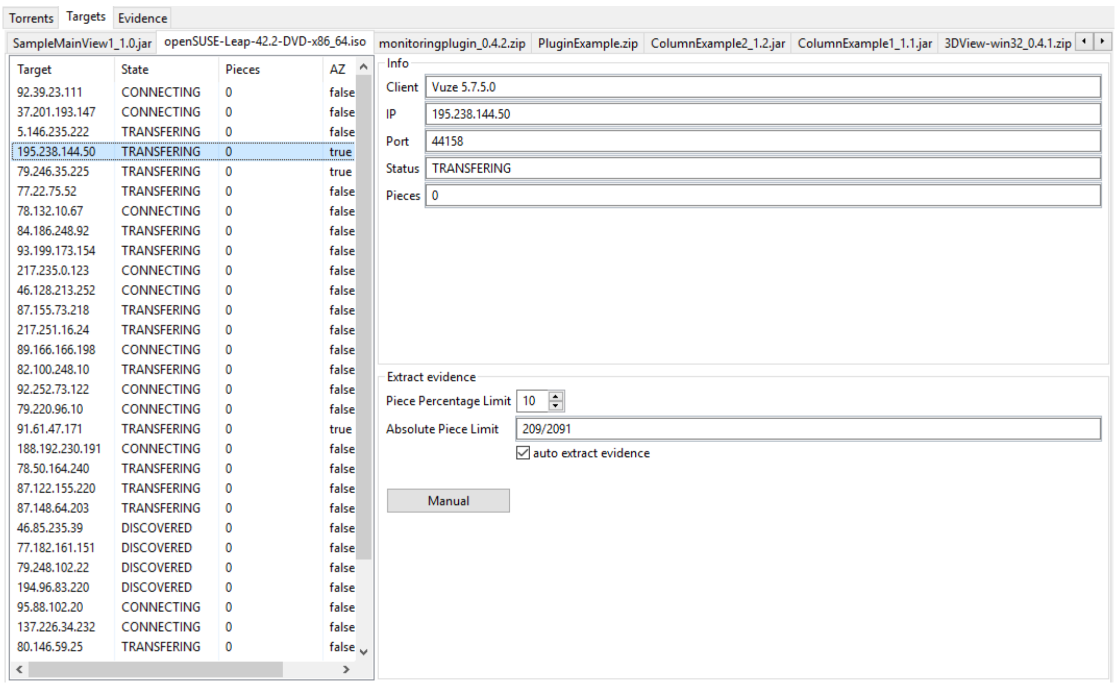
	ISO 3166-1	Name
☐	AD	Andorra
☐	AE	Vereinigte Arabische Emirate
☐	AF	Afghanistan
☐	AG	Antigua und Barbuda
☐	AI	Anguilla
☐	AL	Albanien
☐	AM	Armenien
☐	AN	Niederländische Antillen
☐	AO	Angola
☐	AQ	Antarktis
☐	AR	Argentinien
☐	AS	Amerikanisch-Samoa
☑	AT	Österreich
☐	AU	Australien
☐	AW	Aruba
☐	AX	Aaland-Inseln
☐	AZ	Aserbaidschan
☐	BA	Bosnien und Herzegowina
☐	BB	Barbados
☐	BD	Bangladesch
☐	BE	Belgien

☐ include unknown countries

Figure 4.3.: Country filter: countries are defined by ISO 3166-1

4.3.3. Target observation

When a peer passed all filters and becomes a target, it is added to the observed torrent and therefore will be automatically observed. So only target peers will be added, there are no peers which will not be observed. Each target is now treated by Vuze as a normal peer and it follows the usual process of filesharing in BitTorrent, described in section 2.2.6. The only exception to this is, that all outgoing and incoming messages of the protocol will be recorded to process them afterwards. Additionally outgoing data messages will be dropped to fulfill the requirement of not uploading data. How this is implemented is described below. Figure 4.4 shows the observed target with their current state and the piece count, which have already be downloaded. This piece count is the metric to decide when enough data is available as evidence.



Target	State	Pieces	AZ
92.39.23.111	CONNECTING	0	false
37.201.193.147	CONNECTING	0	false
5.146.235.222	TRANSFERING	0	false
195.238.144.50	TRANSFERING	0	true
79.246.35.225	TRANSFERING	0	true
77.22.75.52	TRANSFERING	0	false
78.132.10.67	CONNECTING	0	false
84.186.248.92	TRANSFERING	0	false
93.199.173.154	TRANSFERING	0	false
217.235.0.123	CONNECTING	0	false
46.128.213.252	CONNECTING	0	false
87.155.73.218	TRANSFERING	0	false
217.251.16.24	TRANSFERING	0	false
89.166.166.198	CONNECTING	0	false
82.100.248.10	TRANSFERING	0	false
92.252.73.122	CONNECTING	0	false
79.220.96.10	CONNECTING	0	false
91.61.47.171	TRANSFERING	0	true
188.192.230.191	CONNECTING	0	false
78.50.164.240	TRANSFERING	0	false
87.122.155.220	TRANSFERING	0	false
87.148.64.203	TRANSFERING	0	false
46.85.235.39	DISCOVERED	0	false
77.182.161.151	DISCOVERED	0	false
79.248.102.22	DISCOVERED	0	false
194.96.83.220	DISCOVERED	0	false
95.88.102.20	CONNECTING	0	false
137.226.34.232	CONNECTING	0	false
80.146.59.25	TRANSFERING	0	false

Info

Client	Vuze 5.7.5.0
IP	195.238.144.50
Port	44158
Status	TRANSFERING
Pieces	0

Extract evidence

Piece Percentage Limit	10
Absolute Piece Limit	209/2091

[✓] auto extract evidence

Figure 4.4.: Target observation

In Vuze each peer has its own *Connection* over which messages can be sent and received. This Connection consists of two message queues, the *IncomingMessageQueue* and *OutgoingMessageQueue*. To each of them a listener is registered to intercept the messages sent or received. This listeners are:

- **IncomingMessageQueueListener**: The only relevant method is *messageReceived*. It provides every valid incoming message. Invalid messages, which can not be decoded, will be dropped by Vuze automatically. Every message is stored in the target's incoming message list. Additionally incoming handshakes will be inspected to recognize other Vuze clients, because they have to be treated in a special way because of Vuze BitTorrent extensions.
- **OutgoingMessageQueueListener**: A method, called *messageAdded*, indicates that a messages is about to be added to the OutgoingMessageQueue and is ready for sending. If it returns *false* the messages will be dropped and is not added to the queue and guaranteed not to be sent. This enables the possibility to tamper with or drop messages, which is used to prevent the application to send data in the form of *piece*-messages. In contrast the method *messageSent* provides the already sent messages, which were queued in the outgoing message list of the target, but has no mechanism to check the receipt. This is used to collect all sent messages of the client to the target.

Each collected messages is stored with a timestamp, the internal Vuze representation *Message*, as well as the binary data of the message as *ByteBuffer*. This messages are stored separately in an incoming and outgoing message list which are processed in the next step of evidence extraction.

To collect meaningful evidence, enough data has to be collected. To have a good measurement of having enough data the pieces a target has downloaded are counted. When this piece count exceeds a user defined threshold, or is stopped manually, the target is removed from the torrent and is further processed outside the BitTorrent functionality. The threshold is defined as a percentage of the available pieces in a torrent, which depends on the piece size and the selected interesting files. It depends on type and size of an interesting file, how many pieces are necessary for reasonable evidence collection and is up to the user to decide. This threshold can be set in the same tab as the target observation and is shown in figure 4.4 in the bottom right box.

4.3.4. Extract evidence

When a target's observation has finished, the next step in the process is to extract evidence from the collected data and information. This evidence includes hard facts, in the form of the data traffic, as well as information derived from the meta data, like the history of a target. To get these two types the collected data has to be processed in two different ways. For shared data traffic, the pieces and messages have to be reassembled and verified, which is done in the *MessageReassembler*. For meta data the protocol messages have to be analyzed, which is done in the *HistoryExtractor*.

Message reassembling

To be able to verify downloaded data, it is necessary to reconstruct data at least on piece level to compare the hash value with the one in the torrent file. Dependent on the amount of reconstructed data it can be possible to reconstruct fully downloaded files, which then can be verified with the md5-hash located in the file dictionary in the torrent file. This reconstruction of data is done by the application itself and not by Vuze. This is because there is no possibility to distinguish the origin of the pieces from different targets and keep them separated. So it is guaranteed that all targets are treated separately and and no false positive is created.

As described in section 2.2.3 and shown in figure 2.3 the structure of a torrent is split up in files, pieces and blocks. So the blocks, as the only unit shared in the messages, have to be reassembled to pieces, which then can be reassembled to files, if all pieces of a file are available. This structure splits up the reassembling process into two stages:

Reassemble messages to pieces

To reassemble the messages some operating numbers are required:

- $torrentSize$ the overall size in bytes of the shared data in a torrent, computed and provided by Vuze. newpage

- $pieceSize$ the size in bytes of a single piece except the last one, provided by the torrent file.
- $pieceCount$ the amount of pieces of a torrent, computed by the following formula:
$$pieceCount = \lceil\frac{torrentSize}{pieceSize}\rceil$$
- $lastPieceSize$ the size in bytes of the last piece, which is irregular, of a torrent. It is computed by the following formula:
$$lastPieceSize = torrentSize - pieceSize * (pieceCount - 1)$$

With these numbers all information is available to start reassembling the messages, shown in figure 4.5. First a map *pieceMap* is created, where we can link partly restored pieces to their index. As shown in listing 4.2, each collected message of type *BT_PIECE*, which holds the shared data in form of blocks, is processed. To insert a block into the right piece and position into the piece, the key of the piece and the offset has to be extracted of the message. Because of Vuze BitTorrent extensions, Vuze clients has extended meta data, so this has to be considered when reassembling pieces. If a piece is not available in the *pieceMap* yet, it is created. The last piece of a torrent is treated specially because of its different length. Now all information and objects are available to store the actual binary data into a piece.

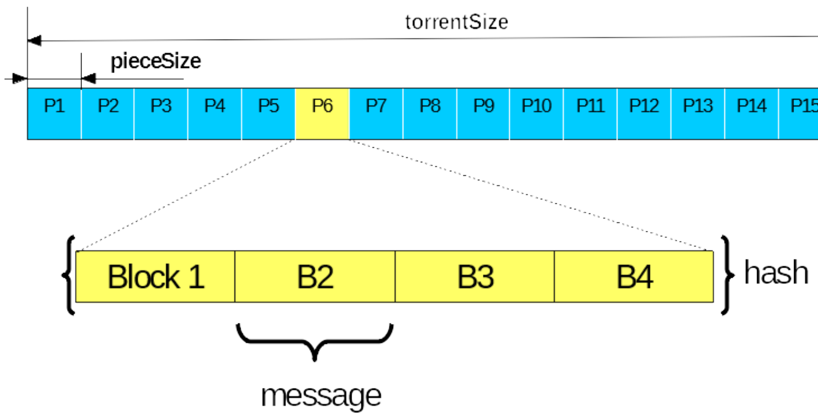


Figure 4.5.: Reassemble messages to pieces

Listing 4.2: reassemble messages to pieces

```
for (MyMessage message : incomingMessageList) {
  if (message.getMsg().getID().equals("BT_PIECE")) {
    metaData = extractKey(message.getMsg().getPayload());
    key = metaData.getF1();
    offset = metaData.getF2();
    if (!pieceMap.containsKey(key)) {
      //handle last piece
      if (key == torrent.getPieceCount() - 1) {
        piece = new MyPiece(key,
                            lastPieceSize,
                            message.getTime());
      } else {
        piece = new MyPiece(key,
                            pieceSize,
                            message.getTime());
      }
      pieceMap.put(key, piece);
    } else {
      piece = pieceMap.get(key);
    }
    piece.addData(message.getTime(),
        offset,
        Arrays.copyOfRange(message.getData(),
                            metaLength,
                            m.getData().length));
  }
}
```

When finished processing all messages, the reassembled pieces can be verified with the hash values from the torrent file, see section 2.2.4. To get these hashes the *pieces* string is convertet into a bytearray with the piece number as index and the hash values as entries, so it can be easily accessed in the application.

For further processing of the pieces we only consider complete pieces, all others are useless, as we cannot verify their integrity. A piece is consider as complete when all blocks of it have data stored inside. For each complete piece a SHA1 hash value is computed and compared with the referring hash value from the torrent file. If successful the piece is added to the *pieceList*, which is returned at the end for further processing.

Reassemble pieces to files

To reassemble the files some numbers have to be known for each different file. This is because the borders of the files inside a torrent do not line up with the borders of the pieces. These numbers are:

- $fileSize$ in bytes, provided by the torrent file.
- $byteStartRange$ the byte a file starts, initially 0.

$$byteStartRange_n = byteEndRange_{n-1}$$

- $byteEndRange$ the byte a file ends.

$$byteEndRange = byteStartRange + fileSize$$

- $startPiece$ the index of the piece which contains the $byteStartRange$.

$$startPiece = \lfloor \frac{byteStartRange}{pieceSize} \rfloor$$

- $endPiece$ the index of the piece which contains the $byteEndRange$.

$$endPiece = \lfloor \frac{byteEndRange}{pieceSize} \rfloor$$

With these numbers, pieces can now be written to each file as shown in figure 4.6. Files are represented as *RandomAccessFile* to write data to them at arbitrary positions, according to which pieces are available. Knowing all files and numbers, the application can now iterate over the files and fill it with data from the pieces.

For each file the $startPiece$ and the $endPiece$ is known. Now all pieces belonging to the file will be copied into the file, with three exceptions:

- **First piece**: An offset where the data of the file starts has to be computed and only data after this offset is copied.
- **Last piece**: An offset where the data of the file ends has to be computed and only data before this offset is copied.
- **Empty piece**: If a piece is not available, because it was not downloaded or can not be verified, the file as a whole can not be restored and will be marked as incomplete.

If all pieces of a file are processed and valid the file is marked as complete and can be used as evidence.

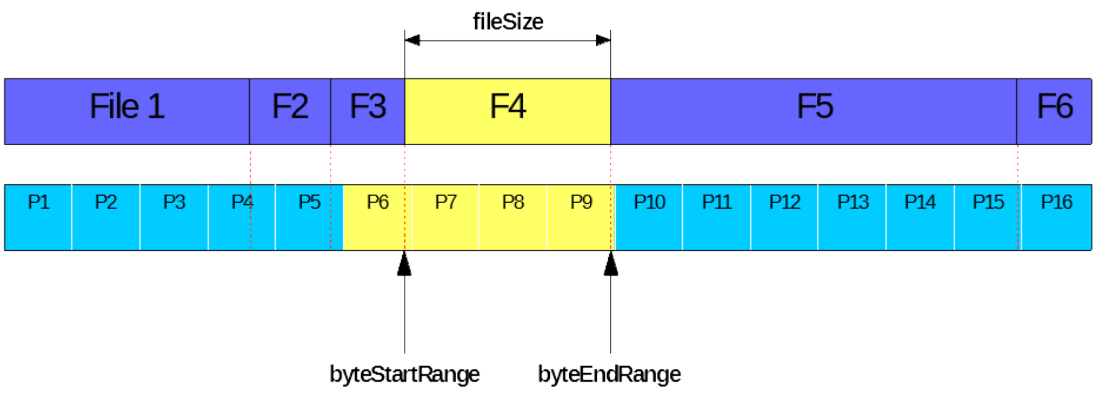


Figure 4.6.: Reassemble pieces to files

History extraction

Within the history extraction the messages of the BitTorrent-protocol, which tells something about the history of a target, will be inspected. These messages are basically for managing the requests for specific pieces. So they contain information what piece or pieces a peer announces to the swarm and can be downloaded from it. With this information, in addition to the timing information, it is possible to reconstruct the sharing history of a torrent for a target. As a reminder, these elements are no hard facts, because there is no way to prove them. A tampered client can basically send what ever they want, but it might be suspicious and monitors detect such misbehaving client and put them on block lists.

There are several messages worth to look at. As they are a essential part of the BitTorrent-protocol to work, they are always available. The messages considered for the history are:

- **bitfield**: Message sent immediately after the handshake, so it appears only once. It indicates all pieces a target has when the observation starts. It contains a binary array where every bit represents the availability of a piece.

  bitfield: □□■■□■□□ $= 00110100_2 = 1A_{16}$

  indicates that pieces number 3, 4, and 6 are available.

- **have**: Message sent when a piece was successfully downloaded and verified via the hash value. Different clients implement this message in different ways. Beside the implementation according to the standard, for example there is the strategy to only advertise pieces the receiving peer does not have, which results in less traffic overhead.

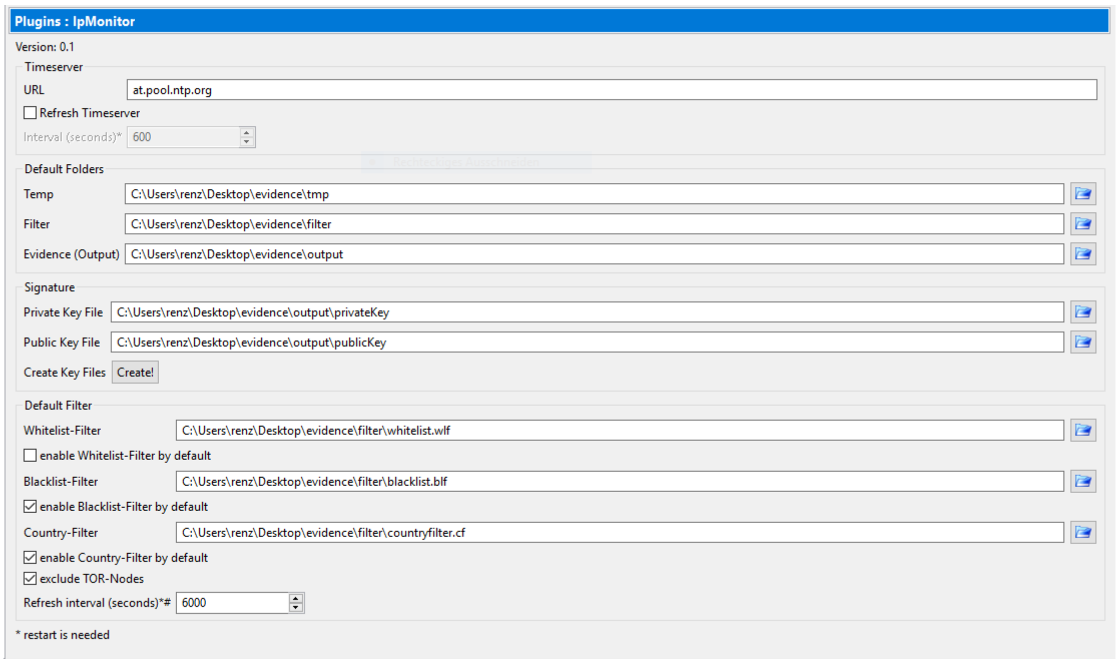
- **haveAll/haveNone**: These two messages are part of the *Fast Extension*[6]. To reduce overhead traffic and these messages just indicates that a target has all or no pieces at all.

To create the sharing history of a target the collected messages are filtered for the

[6]bittorrent.org/beps/bep_0006.html

messages described above and sorted by time. Starting with the bitfield message a reference bitfield as bytearray for the targets pieces is created and pre-filled. Unless the target has not downloaded all pieces yet, for every further message the bitfield array is updated with the new pieces and stored seperatly per message. Additionally a percentage and the according timestamp is stored, so the changes of shared files over time is visible in pieces and percentage.

With this information the download progress of a target can be observed but the origin of the pieces of this progress can not be determined. So when a target keeps advertising new pieces over time, it is an indication that the target is still actively downloading parts of the torrent. But to really prove the public availability of the announced pieces, they has to be downloaded and verified.


Plugins : IpMonitor
Version: 0.1

Timeserver
URL: at.pool.ntp.org
☐ Refresh Timeserver
Interval (seconds)*: 600

Default Folders
Temp: C:\Users\renz\Desktop\evidence\tmp
Filter: C:\Users\renz\Desktop\evidence\filter
Evidence (Output): C:\Users\renz\Desktop\evidence\output

Signature
Private Key File: C:\Users\renz\Desktop\evidence\output\privateKey
Public Key File: C:\Users\renz\Desktop\evidence\output\publicKey
Create Key Files: Create!

Default Filter
Whitelist-Filter: C:\Users\renz\Desktop\evidence\filter\whitelist.wlf
☐ enable Whitelist-Filter by default
Blacklist-Filter: C:\Users\renz\Desktop\evidence\filter\blacklist.blf
☑ enable Blacklist-Filter by default
Country-Filter: C:\Users\renz\Desktop\evidence\filter\countryfilter.cf
☑ enable Country-Filter by default
☑ exclude TOR-Nodes
Refresh interval (seconds)*#: 6000

* restart is needed

Figure 4.7.: Plugin settings

4.3.5. Store evidence

After completing the evidence extraction the result has to be stored. This is achieved in a compact, tamper proof and human readable form. The output of this step is a zip file per target and an according digital signature. Inside the zip file the results are stored in two different ways. First the actual downloaded binary data and second the meta data, history and messages as XML files.

Binary data

The downloaded binary data itself would not be very useful without context, but the reassembled files, even if not completed, are usable. So the reassembled files are stored in a folder like they would be stored regularly by a BitTorrent client. The completeness of the files and which parts are missing can be looked up in the *info.xml* about the target. This storage of binary data is not usable for all kinds of file formats, like encrypted containers, but can be very useful for formats where even small parts can contain usable, independent information like pictures or audio.

XML

To provide evidence in a human readable way and in a structured form it can easily processed by other programs it is stored in the XML file format. There are three different XML files which provide information about the target itself and a summary about the collected data, the observed share history of the target, and all single messages sent or received by the application.

The implementation makes use of a standard Java library, the Document Object Model (DOM) parser, which is an in-memory representation of the XML file. This enables the possibility to create the XML file without a specific order and change elements until it is written to a file. For creating such a DOM a *DocumentBuilder* provides a new *Document* where first a root element has to be added and afterwards normal elements and attributes can be appended. When the DOM is finished, a *Transformer* transforms it into an actual XML file.

Header

The header, described in listing 4.3, of all XML files are the same. It contains the target's IP, peerId and date, so it can be matched over different files of one target.

Listing 4.3: Example of header in each XML file

```xml
<info>
  <header>
    <peer>
      <ip>123.123.123.213</ip>
      <id>2d417a353734402d3830505559596467344e7868</id>
      <date>Mon Sep 25 21:01:01 CEST 2017</date>
      <client>Vuze 5.7.5.0</client>
    </peer>
    <torrent>
      <name>torrentName</name>
      <infohash>epk51MxXilxn55tubi6MbHFEd4k=</infohash>
    </torrent>
  </header>
  <file-list>
    <file complete="true"
          name="README.txt"
          size="771␣bytes">
      <progress piece-count="1" total-piece-count="1"/>
      <start-time>Mon Sep 25 21:01:05 CEST 2017</start-time>
      <end-time>Mon Sep 25 21:01:07 CEST 2017</end-time>
    </file>
  </file-list>
  <time-server offset="-0.39248037338256836"
               time-server="at.pool.ntp.org"/>
</info>
```

info.xml

In the info.xml file all relevant information about the target and the observation are stored. Beside the header this includes information about the used timeserver and tor exit node list provider as well as the torrent and its containing files. To these files additional meta data are added, which contains start and end of observation and how many pieces were downloaded of that specific file. The XML shema definition (XSD) is shown in appendix A.1.

history.xml

The history.xml contains the collected data from the history extraction. The history is sorted ascending by date and shows all history related messages as elements. Each element contains the date, the causing message and the change in percent and as bitfield. The XSD is shown in appendix A.2.

messages.xml

The message.xml file provide all messages collected in observation. Starting with a summary of the amount and type of messages, the incoming and outgoing messages are shown as a list. Each message has description and a timestamp when the message was first recognized. Depending on the message type it can also has a payload of binary data, which is encoded in Base64[7]. The XSD is shown in appendix A.3.

Pack

After generating all binary and XML files showed in table 4.1 all elements are packed and compressed in a ZIP file. This is necessary to have all evidence in one file, which can then be signed afterwards and easily transferred. In the end this ZIP file must contain all information to reconstruct the process of evidence finding.

This step is implemented with the standard Java library and uses a *ZipOutputStream* to compress the targets folder into itself, so that all files, including the generated ZIP file, related to a target are in the same folder.

[7]tools.ietf.org/html/rfc4648

Name	Description
.torrent	The torrent file used for observation
info.xml	Summary and general information about the observation
history.xml	Share history of the observed target
messages.xml	Collected in and outgoing messages
Torrent folder	Contains the downloaded files

Table 4.1.: Content of packed evidence

Sign

The final step of the process of evidence collection is to digitally sign the collected and compressed evidence to fulfill the chain of custody. This has to be done to prove the integrity and authenticity of the evidence. So it can be verified that there has been no tampering of the data and the signer is the collector of the evidence.

The implementation uses the Java Security API for generation and verification of signatures.[20] As this application is a prototype there is only one cryptographic system implemented to sign data, namely RSA.

Key Files: To use the signing ability of the application a key pair, consisting of a private and a public RSA key, has to be provided. In the settings of the application, shown in figure 4.7, there is a possibility to select a private and public key file. If no key pair is provided there is also the possibility to create a new RSA key pair with the length of 2048 byte and the pseudo-random number generation algorithm SHA1PRNG.

Sign Data: To sign the evidence the generated ZIP file is passed to the sign method. There a *Signature* object is created with the algorithm SHA1withDSA. Before use the *Signature* is initialized with the private key. Now the input ZIP file is read byte wise and updates the *Signature*. When the file was read completely the signature is now complete in the referring object[20]. As a last step the signature is written to a file in the directory containing the evidence.

With this step the collection and storing of evidence is finished. There is now a ZIP file a specific target of a specific torrent containing all collected information and messages in binary as well as human readable form. This file is also tamper proof with the belonging signature and can now be delivered for further use.

4.4. Evidence viewer

To show and verify the collected evidence the application provides a very simple evidence viewer. To show the evidence the generated ZIP file of a target has to be selected in the file tree. Then the evidence is opened and shown on the right side of the viewer. It consists of three parts shown in figure 4.8.

4.4.1. ZIP file verification

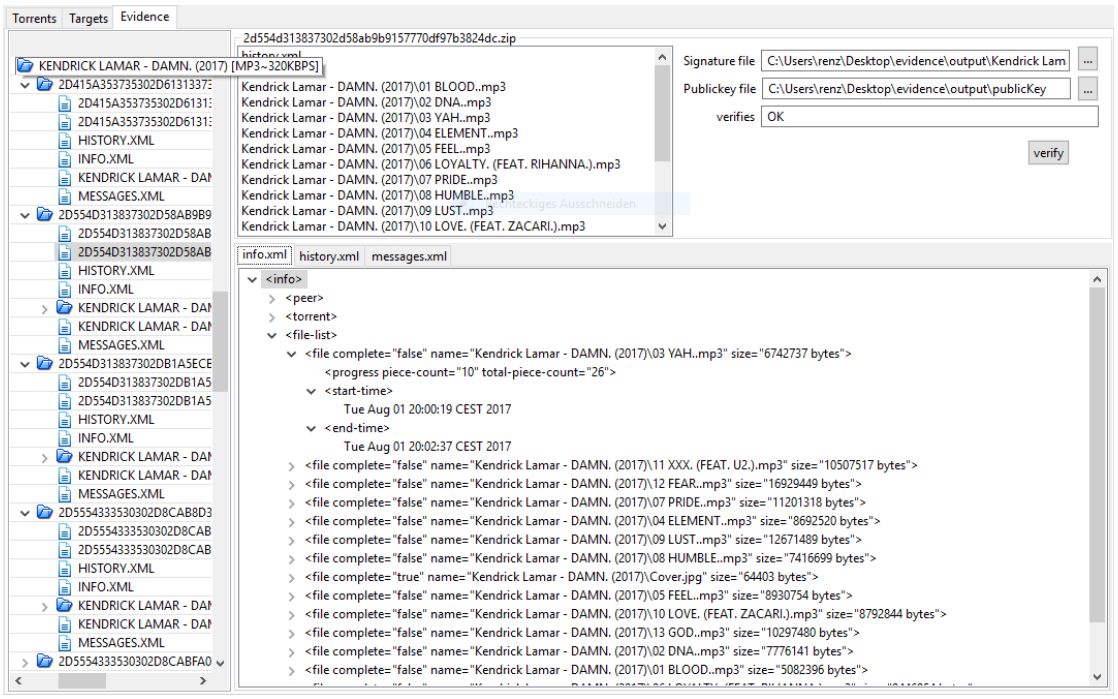
On the top left the ZIP viewer is displayed. It just shows the folders and files of the ZIP file. The verification is located on the top right of the evidence viewer. It provides a small tool to match the signature file of the ZIP file with a file containing the matching public key. So tampering with the packed evidence can be discovered easily.

The implementation of the signature verification is similar to the signing step described in section 4.3.5. To verify the signature the public key is needed. So a *Signature* object is created with the SHA1withDSA algorithm and afterward initialized with the public key. Then the *Signature* is updated byte wise with the ZIP file. In the end the computed signature hast to match the one in the signature file. If this comparison fails it is possible that the evidence file has been compromised or a false public key is used.

4.4.2. XML viewer

In the bottom the content of the generated XML files are shown. The tabs only display the generated files, even if other XML files are present in the ZIP file. It provides a short overview of the evidence and a simple way to look deeper into the specific XML files.

Because a XML file can be seen as a tree structure the viewer is a simple implementation of a *TreeViewer* from the JFace[8] UI toolkit provided by Eclipse.



```
KENDRICK LAMAR - DAMN. (2017) [MP3~320KBPS]
  2D415A353735302D61313373
    2D415A353735302D61313
    2D415A353735302D61313
    HISTORY.XML
    INFO.XML
    KENDRICK LAMAR - DAN
    MESSAGES.XML
  2D554D313837302D58AB9B9
    2D554D313837302D58AB
    2D554D313837302D58AB
    HISTORY.XML
    INFO.XML
    KENDRICK LAMAR - DAN
    KENDRICK LAMAR - DAN
    MESSAGES.XML
  2D554D313837302DB1A5ECE
    2D554D313837302DB1A5
    2D554D313837302DB1A5
    HISTORY.XML
    INFO.XML
    KENDRICK LAMAR - DAN
    KENDRICK LAMAR - DAN
    MESSAGES.XML
  2D5554333530302D8CAB8D3
    2D5554333530302D8CAB
    2D5554333530302D8CAB
    HISTORY.XML
    INFO.XML
    KENDRICK LAMAR - DAN
    KENDRICK LAMAR - DAN
    MESSAGES.XML
  2D5554333530302D8CABFA0
```

```
2d554d313837302d58ab9b9157770df97b3824dc.zip
history.xml
Kendrick Lamar - DAMN. (2017)\01 BLOOD..mp3
Kendrick Lamar - DAMN. (2017)\02 DNA..mp3
Kendrick Lamar - DAMN. (2017)\03 YAH..mp3
Kendrick Lamar - DAMN. (2017)\04 ELEMENT..mp3
Kendrick Lamar - DAMN. (2017)\05 FEEL..mp3
Kendrick Lamar - DAMN. (2017)\06 LOYALTY. (FEAT. RIHANNA.).mp3
Kendrick Lamar - DAMN. (2017)\07 PRIDE..mp3
Kendrick Lamar - DAMN. (2017)\08 HUMBLE..mp3
Kendrick Lamar - DAMN. (2017)\09 LUST..mp3
Kendrick Lamar - DAMN. (2017)\10 LOVE. (FEAT. ZACARI.).mp3
```

Signature file: C:\Users\renz\Desktop\evidence\output\Kendrick Lam

Publickey file: C:\Users\renz\Desktop\evidence\output\publicKey

verifies: OK

```
<info>
  <peer>
  <torrent>
  <file-list>
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\03 YAH..mp3" size="6742737 bytes">
      <progress piece-count="10" total-piece-count="26">
      <start-time>
        Tue Aug 01 20:00:19 CEST 2017
      <end-time>
        Tue Aug 01 20:02:37 CEST 2017
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\11 XXX. (FEAT. U2.).mp3" size="10507517 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\12 FEAR..mp3" size="16929449 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\07 PRIDE..mp3" size="11201318 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\04 ELEMENT..mp3" size="8692520 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\09 LUST..mp3" size="12671489 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\08 HUMBLE..mp3" size="7416699 bytes">
    <file complete="true" name="Kendrick Lamar - DAMN. (2017)\Cover.jpg" size="64403 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\05 FEEL..mp3" size="8930754 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\10 LOVE. (FEAT. ZACARI.).mp3" size="8792844 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\13 GOD..mp3" size="10297480 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\02 DNA..mp3" size="7776141 bytes">
    <file complete="false" name="Kendrick Lamar - DAMN. (2017)\01 BLOOD..mp3" size="5082396 bytes">
```

Figure 4.8.: Evidence viewer with opened and verified evidence file

[8]wiki.eclipse.org/JFace

5. Testing

This chapter contains tests that verify the functionality of the prototype. It describes the different tests scenarios, preconditions, and what the tests are about. The concrete test descriptions, results, and outcome of each tests are presented independent for each test.

The tests should cover the complete functionality of the prototype. So a single test will start by going through the complete process of evidence collection and simulates one observation of a target. As a result, there should be the correct and complete output of the collected evidence. Nevertheless, the tests should also cover the different parts of the evidence collection process:

- **Time synchronization**: Offset of the system clock in comparison with an external time source is computed and applied.
- **Target picking**: Filters detect the specified target only. This includes all the different filters: Tor/Country/Whitelist filter.
- **Target observation**:The messages are intercepted and stored correctly, as well as no shared data is uploaded by the prototype.
- **Extract evidence**: Show that the different pieces and files are reassembled correctly and verified by the provided hash values.
- **Store evidence**: The XML-files containing the metadata and analysis of the history are generated correctly.
- **Pack & Sign**: The collected evidence is packed and signed in an single archive and the signature verifies.

5.1. Test setup

In order to reproduce and compare the tests a fixed test setup is needed. This setup consists of a few preconditions about the configuration of the prototype as well as the participants in the test environment. The actual tests vary in the different specialized torrent used. How they differ of each other will be described in the according section of the tests in section 5.2.

5.1.1. Configuration

There are a few preconditions set which are the same for all tests. These are settings in the configuration of the plugin prototype. In general shared files of a torrent, which are not interesting for evidence collection, should be excluded in the torrent. Also the amount of pieces downloaded from a target should be adapted to the specific observed torrent. To keep the tests comparable, all files are selected for observation and the pieceCount is set to 10%.

Timeserver

As external time source `at.pool.ntp.org` is set. This URL links to a pool of time-servers provided by the NTP Pool Project[9]. It will automatically provide the closest available timeserver to use with NTP.

Signature keyfiles

The keypair of private and public key are created by the prototype itself, so they are RSA keys with a keysize of 2048 bit. After creation they stay the same for all tests. Internally the keys are stored in binary format, but for readability they are shown in listing 5.1 and 5.2 in the Privacy-enhanced Electronic Mail (PEM) format.

[9]http://www.pool.ntp.org

Listing 5.1: privateKey.pem

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAnbHLyhhiNTCHhxQCOF0+93N1SUdTSwLON5Xy2EuIKUQ4GUhx
7vqKiP2hZCdjOPg4cvCTcDWe/U8M1WKkEdgDzeCzfBXVznFtGUTYPxBnJFkDbPy+
XKyRLF+YI/4b6/65t/RhvI/IuHNBALA9f25gisSsuvcVOqMc2HvE4EhtZwF18ifL
RbA7UW18cPpcbA5y666OYviZbLUMoc5VGf81qwiTSzKLwhWq+Sv+yV6O18uSERpR
YTozSGv1JJ+XDGIhhp/VvhFBSmVxpFuIABJAfjLReVX9wXP/LD11Hwmxr+F8IG3F
z8ktUb3j4tS4MzWTrdtjyTwQufoTHV17u5TJfQIDAQABAoIBAGYNw3I115DOqFb9
1vX9OpvQB851r2zzEtwcWb4rG6KBOMD8Bmf/kppXchQr/WXIFIgkY3aR7SPH/ek2
3jGCF+wsCFw8kVL6Sv24dS5TAj0esj3dT6EQOR97enibBhzzi+OF75/R1gxJ4n58
kEAhAa+s2pBhCUcaereLWf1AQjUPUDxRGXWMpT8MBL9unQGdBbgwIT/S3GMTFDIm
QCGVuzKzMywBgPnHE+SOvS0o1mW0QX5TnROfFeQaPji6tMEDuEt9m1xoliwhqak9
EkIA7k5VTW8evhwLTYegQ/5MPsR3w4FFWeMe7zk2iHxTJwuoGpbozV74s7kLeAwo
WTwkFw0CgYEA7YTmDq4b3u5Y7X9baLhkkIjjq8k7zUmAc0mIzqKRboxIEI+LxPKx
fRFRfvKjoG1ysPBUWmS9ZkxSA58kyM5IA+F+nQopEna5FFIyz2RgBBQYF+9jaVro
H3suO1LNbrM/ayx48AFLhdLSMk6AxoO2wDLUXwHNTjMbHroKTXbx2/MCgYEAqfbi
kG+JbI6qniU/65s31ZIII+j4Srdp77KEbAIPmxEzT3T5JHSG24MTFI/XXnIx7Vjg
TxpNxaTdJ3I4OhZiDj8Iiqn44wF5fs8BxkwQTB/RLEEcii7f+vr1Qv46rIB72N+S
9Ix/iQ/cos3uOBVNQgcZjyi7DihzTdeRUVuYUM8CgYBtM2PeLfhMZ4LINI+dZuF9
fiJGkc9306pF1vIaJq38iHnSwIx9YSNvH+47Em1eTdOkO+pcYOKfTMmMNrffxs9f
UWY35zr47TKsgBoyNbDbTy3Am2Q2RQBhsO2RgsFGewzWRm0X01CYp1PSozsbieUU
uonfYGG26VI3r0sbUGtkNwKBgHNLI8uU7exh6WmYKTFBPPHyu0OZdPzySLmiJrRG
6KpvU5g96Fgzd9vmoYcXArkCtybjmF3woPtG6AK/N6cn0eKKHdg6jywmru03raN/
q7qIBtP3Y03VmHjfzopgfRrRmbG3kDDw9E7c5LXH7iY7RQpaWJtFbQagp/8REXHY
KY3/AoGBAOiCYqPOFuMeiBVnynHfWmKIudX06+SLD2/9oRpWhaEPwjeUL2DgFPJ0
X9QsmzZZGj0sWW22wKsEyfIIexcLdgQfdQZrpbfRPczIABMv8cJ5aoU641wnnk/B
e5A1zzHNsKVQ1Fnpwkui366A3cuWHRHtLrhVeEo9oCcBfI1OxIfQ
-----END RSA PRIVATE KEY-----
```

Listing 5.2: publicKey.pem

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnbHLyhhiNTCHhxQCOF0+
93N1SUdTSwLON5Xy2EuIKUQ4GUhx7vqKiP2hZCdjOPg4cvCTcDWe/U8M1WKkEdgD
zeCzfBXVznFtGUTYPxBnJFkDbPy+XKyRLF+YI/4b6/65t/RhvI/IuHNBALA9f25g
isSsuvcVOqMc2HvE4EhtZwF18ifLRbA7UW18cPpcbA5y666OYviZbLUMoc5VGf81
qwiTSzKLwhWq+Sv+yV6O18uSERpRYTozSGv1JJ+XDGIhhp/VvhFBSmVxpFuIABJA
fjLReVX9wXP/LD11Hwmxr+F8IG3Fz8ktUb3j4tS4MzWTrdtjyTwQufoTHV17u5TJ
fQIDAQAB
-----END PUBLIC KEY-----
```

Filters

The configuration for the different filters are shown in table 5.1.

Filters	**Values**
Whitelist	containing the target's IP-address
Blacklist	containing the source's IP-address
Country	enabled and Austria (AT) as only country selected
Tor	Tor exit nodes are excluded and the list is refreshed every 10 minutes

Table 5.1.: Filter configurations

5.1.2. Test environment

To stick as close as possible to the real world, the tests should take place over the internet and not in a controlled local network. The reason about this decision is, that modern BitTorrent clients are rather good with peer finding algorithms and recognize local peers. In consequence there is no public IP-address to pin down. Instead of solving this problem by a complex network setup, it is easier and closer to reality to use the internet with different providers.

To run the different tests in this environment a special torrent file is needed as well as at least three participants (source, target and observer) to meet the minimum requirements for a BitTorrent swarm, as they are described in detail below.

Torrent file

For each test there is a special prepared torrent available. These torrent files are distributed to all three participants to join the BitTorrent swarm. The data shared by these torrents are described in the specific test sections, but are all freely available and allowed to share in public. But nevertheless the torrent files shall not be shared outside of this test environment, to not monitor third parties within the tests.

Source

The source, or in terms of BitTorrent the seeder, is needed to provide a data source for the target to download. It is a simple Windows machine connected to the internet with fully working installation of Vuze. In theory Vuze could also act as a tracker for the test-torrents, but this functionality is not needed in this tests. Finding other peers works also via DHT so a specific tracker is not required, and for testing purpose the peer source does not matter.

Target

The machine which is observed in the tests. To act as a target, a fully working BitTorrent client, Vuze, is installed. Vuze is not configured in any special way, just with its default settings. So it simulates an ordinary user in a BitTorrent swarm. It starts with the torrent file added to Vuze for download. So in the beginning of each test there is no shared data on the target machine stored. The only thing available is the torrent file to be observed for the given test.

Observer

The machine running the prototype application to test. It is a machine with the prototype plugin installed in Vuze. As the target, it only has the torrent files to observe without any data. The observer is the only one, where direct access is possible, like it would be in real world. So the tests will be started from the observer, and and will take place until enough data has been collected.

5.2. Test scenarios and results

As mentioned before, the different tests are about different torrent with special characteristics. So the tests cover special cases and evaluate the usability of the resulting evidence. The prepared torrent files vary in two ways, both variables influences the way to use the prototype:

- **File size**: The specific sizes of the shared files. This affects the possibility of downloading a complete file as evidence.
- **File count**: The amount of files shared within a torrent. This affects the amount of data needed to gather evidence for different files.

5.2.1. Scenario 1: Multiple files with small size

In this scenario the observed torrent will contain a bunch of relative small (100kB up to 2MB) files. This simulates shared music albums if they are not packed into one file, which is a common use-case. This test scenario should ensure that the implemented prototype works correct and provides useful evidence for this scenario.

As test result, it is expected that the collected evidence are correct and verified. This also includes that the observed peer is identified correctly. The collected data has to be reassembled and verified. Because of the small sizes of the shared files, there should also be at least one file that is completely restored. As the target starts download with the beginning of the test, the history should contain correct data too.

Setup

The scenario setup is defined by the torrent shown in table 5.2, which contains a README file and a selection of various Java libraries up to 2 megabyte of size, taken from the Eclipse IDE. The information about the participating BitTorrent clients are shown in table 5.3.

Attribute	**Value**
Name	testData
Hash	2CBAD1A1 68BB8F41 DFA5A570 6072C7B2 0DEA7F27
Total Size	216,11MB
# of Pieces	1729
Piece Size	128kB
Files	107

Table 5.2.: testData.torrent

Participant	**IP**	**ISP**	**OS**	**Client**
Observer	46.124.119.153	T-Mobile Austria	Windows 10	Vuze 5.7.5.0
Target	93.82.156.213	A1 Telekom Austria	Windows 10	Vuze 5.7.5.0
Source	91.119.65.135	UPC Austria	Windows 7	Vuze 5.7.5.0

Table 5.3.: participants of testData.torrent

Result

The test run has been completed without any errors and produced the expected evidences in form of XML files and downloaded data files inside a signed ZIP package. The test result will be discussed step by step with an interpretation in the end.

The target observation started at 17:54 on 11th of November 2017 with the BitTorrent handshake which is shown in listing 5.6 and listing 5.7. The observation was triggered by passing the special prepared filters when the target peer was discovered. The observation stopped at 18:15 as planned after 10% (172 pieces/21MB) of the torrent was downloaded.

As counted in the message summary in listing 5.5 1380 BT_PIECE messages were reassembled. Altogether 64 files were partly restored and one file was downloaded and reassembled completely. The info.xml, shown in listing 5.4, identifies the *commons-io-2.2.jar* as completed file, which was in addition verified manually in listing 5.3 to demonstrate the correct reconstruction of files from received messages.

Listing 5.3: commons-io-2.2.jar sha1 comparison

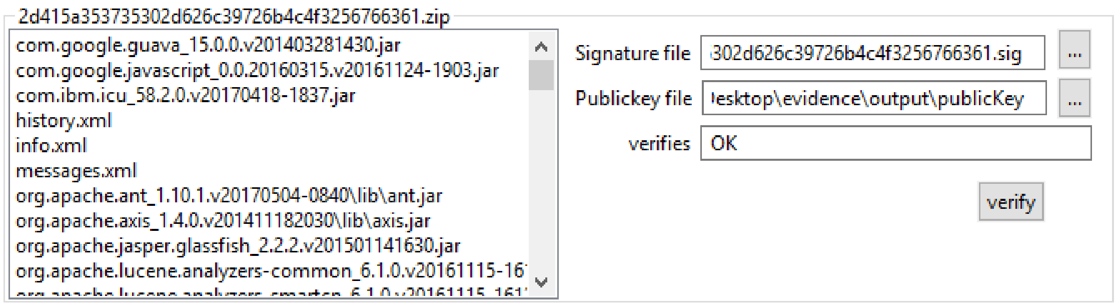
```
Local:
fciv.exe -sha1 .\commons-io-2.2.jar
83b5b8a7ba1c08f9e8c8ff2373724e33d3c1e22a

Remote:
https://repo1.maven.org/maven2/commons-io/
    commons-io/2.2/commons-io-2.2.jar.sha1
83b5b8a7ba1c08f9e8c8ff2373724e33d3c1e22a
```

The messages.xml file provides a summary of the collected messages, shown in listing 5.5. The sent and received messages indicated a normal BitTorrent traffic with the only difference that the observer does not share any pieces, as expected.

From the previous mentioned messages the history.xml was created. Listing 5.8 shows the initial completeness state of the target with 1,27% by receiving a BT_BITFIELD message. This value is updated by received BT_HAVE messages or when the observer successfully downloaded a piece (<*piece_downloaded index="49"/*>). In the time period of observation the target completed the torrent to 11,16%. The fact, that the target started at 1,27% and not with zero or a low percentage can be explained by the peer finding processes of BitTorrent. First, a peer joining a swarm has to distributed to other peers in the swarm via different sources, like tracker or DHT, which are updated periodically in the range of minutes. The second reason is, that if a peer just joined a swarm it has nothing to offer, so it denies requests.

As last step the created and reassembled files are packed into the *2d415a353735302d626c39726b4c4f3256766361.zip* file, named according to the peerId, along with the according signature file created with the private key, shown in listing 5.2. Figure 5.1 shows the correct verification of the created package with the signature file.



```
2d415a353735302d626c39726b4c4f3256766361.zip
com.google.guava_15.0.0.v201403281430.jar
com.google.javascript_0.0.20160315.v20161124-1903.jar
com.ibm.icu_58.2.0.v20170418-1837.jar
history.xml
info.xml
messages.xml
org.apache.ant_1.10.1.v20170504-0840\lib\ant.jar
org.apache.axis_1.4.0.v201411182030\lib\axis.jar
org.apache.jasper.glassfish_2.2.2.v201501141630.jar
org.apache.lucene.analyzers-common_6.1.0.v20161115-16

Signature file   302d626c39726b4c4f3256766361.sig
Publickey file   esktop\evidence\output\publicKey
verifies         OK
```

Figure 5.1.: Signed package verifies with public key

The test was a success, because the results are as expected. One file could be restored completely and also many other files could be restored partly by verified pieces. All single steps of evidence extraction were working correct and without any problems.

With the collected evidence it can be proven, that the target made the files, marked in listing 5.4 with a piece count of one or above, public available. In addition listing 5.8 shows that the target was still downloading pieces from other sources, which are later downloaded and successfully verified by the observing client.

Listing 5.4: testData info.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<info>
  <header>
    <peer>
      <ip>93.82.156.213</ip>
      <id>2d415a353735302d626c39726b4c4f3256766361</id>
      <date>Sat Nov 11 17:54:50 CET 2017</date>
      <client>Vuze 5.7.5.0</client>
    </peer>
    <torrent>
      <name>testData</name>
      <infohash>LLrRoWi7j0HfpaVwYHLHsg3qfyc=</infohash>
    </torrent>
  </header>
  <file-list>
     ...
    <file complete="true"
          name="...\commons-io-2.2.jar"
          size="173587␣bytes">
      <progress piece-count="2" total-piece-count="2"/>
      <start-time>Sat Nov 11 17:57:24 CET 2017</start-time>
      <end-time>Sat Nov 11 17:59:49 CET 2017</end-time>
    </file>
     ...
    <file complete="false"
          name="org.eclipse.jst.j2ee_1.2.0.v201704202045.jar"
          size="1889386␣bytes">
      <progress piece-count="4" total-piece-count="15"/>
      <start-time>Sat Nov 11 17:56:25 CET 2017</start-time>
      <end-time>Sat Nov 11 18:15:07 CET 2017</end-time>
    </file>
     ...
  </file-list>
  <time-server offset="-0.4631321430206299"
               time-server="at.pool.ntp.org"/>
  <tor-exit-node-list
          url="https://check.torproject.org/exit-addresses"/>
</info>
```

Listing 5.5: testData messages-summary

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <header>...</header>
  <messages-summary>
    <total-message-count>3183</total-message-count>
    <in-message-list-summary>
      <size>1626</size>
      <message-type count="1" id="AZ_HANDSHAKE"/>
      <message-type count="1" id="BT_BITFIELD"/>
      <message-type count="1" id="AZ_PEER_EXCHANGE"/>
      <message-type count="228" id="BT_HAVE"/>
      <message-type count="3" id="AZ_HAVE"/>
      <message-type count="1" id="BT_HANDSHAKE"/>
      <message-type count="10" id="BT_ALLOWED_FAST"/>
      <message-type count="1" id="BT_UNCHOKE"/>
      <message-type count="1380" id="BT_PIECE"/>
    </in-message-list-summary>
    <out-message-list-summary>
      <size>1557</size>
      <message-type count="1" id="AZ_HANDSHAKE"/>
      <message-type count="1" id="BT_BITFIELD"/>
      <message-type count="1" id="BT_INTERESTED"/>
      <message-type count="172" id="BT_HAVE"/>
      <message-type count="1382" id="BT_REQUEST"/>
    </out-message-list-summary>
  </messages-summary>
  <in-message-list>...</in-message-list>
  <out-message-list>...</out-message-list>
</messages>
```

Listing 5.6: excerpt testData in-message-list

```
<in-message-list>
  <message id="BT_HANDSHAKE">
    <date>Sat Nov 11 17:55:03 CET 2017</date>
    <time>1510419303321</time>
    <description>
      BT_HANDSHAKE of dataID: ...
    </description>
    <payload>...</payload>
  </message>
  <message id="AZ_HANDSHAKE">
    <date>Sat Nov 11 17:55:03 CET 2017</date>
    <time>1510419303326</time>
    <description>AZ_HANDSHAKE from ...</description>
    <payload>...</payload>
  </message>
  <message id="BT_BITFIELD">
    <date>Sat Nov 11 17:55:03 CET 2017</date>
    <time>1510419303602</time>
    <description>BT_BITFIELD</description>
    <payload>...</payload>
  </message>
  <message id="BT_UNCHOKE">
    <date>Sat Nov 11 17:55:04 CET 2017</date>
    <time>1510419304861</time>
    <description>BT_UNCHOKE</description>
  </message>
   ...
  <message id="BT_PIECE">
    <date>Sat Nov 11 17:55:05 CET 2017</date>
    <time>1510419305488</time>
    <description>
      BT_PIECE data for piece #49:0-&gt;16383
    </description>
    <payload>...</payload>
  </message>
   ...
</in-message-list>
```

Listing 5.7: excerpt testData out-message-list

```
<out-message-list>
  <message id="AZ_HANDSHAKE">
    <date>Sat Nov 11 17:55:03 CET 2017</date>
    <time>1510419303371</time>
    <description>AZ_HANDSHAKE from ...</description>
    <payload>...</payload>
  </message>
  <message id="BT_BITFIELD">
    <date>Sat Nov 11 17:55:03 CET 2017</date>
    <time>1510419303371</time>
    <description>BT_BITFIELD</description>
    <payload>...</payload>
  </message>
  <message id="BT_INTERESTED">
    <date>Sat Nov 11 17:55:03 CET 2017</date>
    <time>1510419303622</time>
    <description>BT_INTERESTED</description>
  </message>
  <message id="BT_REQUEST">
    <date>Sat Nov 11 17:55:04 CET 2017</date>
    <time>1510419304913</time>
    <description>
      BT_REQUEST piece #49:0-&gt;16383
    </description>
    <payload>...</payload>
  </message>
   ...
</out-message-list>
```

Listing 5.8: testData history.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sharehistory>
  <header>...</header>
  <history>
    <entry date="Sat␣Nov␣11␣17:55:03␣CET␣2017">
      <message type="BT_BITFIELD">Bitfield update</message>
      <bitfield>...</bitfield>
      <percent>1.2724117987275883%</percent>
    </entry>
    <entry date="Sat␣Nov␣11␣17:55:07␣CET␣2017">
      <message type="BT_HAVE">
        <piece index="219"/>
      </message>
      <bitfield>...</bitfield>
      <percent>1.2724117987275883%</percent>
    </entry>
    <entry date="Sat␣Nov␣11␣17:55:11␣CET␣2017">
      <piece_downloaded index="49"/>
      <bitfield>...</bitfield>
      <percent>1.3302486986697513%</percent>
    </entry>
     ...
    <entry date="Sat␣Nov␣11␣18:15:33␣CET␣2017">
      <message type="BT_HAVE">
        <piece index="1588"/>
      </message>
      <bitfield>...</bitfield>
      <percent>11.16252168883748%</percent>
    </entry>
  </history>
</sharehistory>
```

5.2.2. Scenario 2: Single, large file

In this scenario the observed torrent will contain one big shared file of not quite 400MB in size. This simulates the use-case of sharing films, games, or other large data. In real-life such data usually already is packed into one single file, so in the end this scenario still fits for the use-case. This test scenario should ensure that the implemented prototype works correct and provides useful evidence for this scenario.

As test result it is expected that the collected evidence are correct and verified. This also includes that the observed peer is identified correct. The collected data has to be reassembled and verified. Because of the big sizes of the shared file, only the README file can be restored. As the target starts download with the beginning of the test, the history should as well contain reasonable data.

Setup

The scenario setup is defined by the torrent described in table 5.4, which contains a README file and the movie Big Buck Bunny provided by the Blender Foundation[10] licensed under the Creative Commons Attribution 3.0[11]. Th information about the participating BitTorrent clients are shown in table 5.5.

Attribute	Value
Name	bigBuckBunny
Hash	7A9939D4 CC578A5C 67E38B6E 6E2E4C6C 71447789
Total Size	397,44 MB
# of Pieces	1590
Piece Size	256 kB
Files	2

Table 5.4.: bigBuckBunny.torrent

[10] www.blender.org
[11] creativecommons.org/licenses/by/3.0/

Participant	**IP**	**ISP**	**OS**	**Client**
Observer	46.124.119.153	T-Mobile Austria	Windows 10	Vuze 5.7.5.0
Target	93.82.156.213	A1 Telekom Austria	Windows 10	Vuze 5.7.5.0
Source	91.119.65.135	UPC Austria	Windows 7	Vuze 5.7.5.0

Table 5.5.: participants of bigBuckBunny.torrent

Result

The test run has been completed without any errors and produced the expected evidences in form of XML files and two downloaded data files inside a signed ZIP package. The test result will be discussed step by step with an interpretation in the end.

The target observation started at 17:55 on 11th of November 2017 with the BitTorrent handshake which is shown in listing 5.12 and listing 5.13. The observation was triggered by the pass of the special prepared filters when the target peer was discovered. The observation stopped at 18:32 as planned after 10% (157 pieces/40MB) of the torrent was downloaded from the target by the observer.

The received 2520 BT_PIECE messages are reassembled into 157 pieces. As the *README.txt* file is in one single piece, which was downloaded, it was recovered completely, as shown in 5.11. The rest of the downloaded pieces belongs to the second file of the torrent *big_buck_bunny_720p_h264.mov*, which was in addition partly compared with the original file, shown in listing 5.9

Listing 5.9: big_buck_bunny_720p_h264 sha1 comparison

```
Piece #15 & #16 of downloaded torrent:
fciv.exe -sha1 .\big_buck_bunny_piece15_16_downloaded.mov
b5006d3f2eb77dd534ea107e63430d5dcdf2dd15

Corresponding part of original file
offset = 15 pieces(3840kB), length = two pieces (512kB):
fciv.exe -sha1 .\big_buck_bunny_piece15_16_original.mov
b5006d3f2eb77dd534ea107e63430d5dcdf2dd15
```

The messages.xml file provides a summary of the collected messages, shown in listing 5.11. The sent and received messages indicated a normal BitTorrent traffic with the only difference that the observer does not share any pieces, as expected.

From the previous mentioned messages the history.xml was created. Listing 5.14 shows the initial completeness state of the target with 0,81% by receiving a BT_BITFIELD message. This value is updated by received BT_HAVE messages or when the observer successfully downloaded a piece. In the time period of observation the target completed the torrent to 10.62%. The fact, that the target started at 0,81% and not with zero or a low percentage can be explained by the peer finding processes of BitTorrent, described in scenario 1.

As last step the created and resembled files are packed up in the *93.82.156.213.zip* file, named according to the peerId, along with the according signature file created with the private key, shown in the previous section. As in the first scenario the signature verifies with the provided package and public key.

The test run completed successfully and so are the results. All single steps of evidence extraction were working correct and without any problems. As expected the single *README* file could be restored. Only the distribution of the downloaded pieces of the movie file varies so much, that with nearly 10% of the video file no video player was able to even show parts of the movie.

With the collected evidence it can be proven, that the target made the files *big_buck_bunny_720p_h264.mov* and *README.txt* public available. In addition listing 5.14 shows that the target was still downloading pieces from other sources, which are later downloaded and successfully verified by the observing client.

Listing 5.10: bigBuckBunny info.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<info>
  <header>
    <peer>
      <ip>93.82.156.213</ip>
      <id/>
      <date>Sat Nov 11 17:55:50 CET 2017</date>
      <client/>
    </peer>
    <torrent>
      <name>bigBuckBunny</name>
      <infohash>epk51MxXiIxn44tubi5MbHFEd4k=</infohash>
    </torrent>
  </header>
  <file-list>
    <file complete="true"
          name="README.txt"
          size="771␣bytes">
      <progress piece-count="1" total-piece-count="1"/>
      <start-time>Sat Nov 11 17:55:57 CET 2017</start-time>
      <end-time>Sat Nov 11 17:56:33 CET 2017</end-time>
    </file>
    <file complete="false"
          name="big_buck_bunny_720p_h264.mov"
          size="416751190␣bytes">
      <progress piece-count="157" total-piece-count="1590"/>
      <start-time>Sat Nov 11 17:56:33 CET 2017</start-time>
      <end-time>Sat Nov 11 18:32:09 CET 2017</end-time>
    </file>
  </file-list>
  <time-server offset="-0.4631321430206299"
               time-server="at.pool.ntp.org"/>
  <tor-exit-node-list
          url="https://check.torproject.org/exit-addresses"/>
</info>
```

Listing 5.11: bigBuckBunny messages-summary

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <header>...</header>
  <messages-summary>
    <total-message-count>5379</total-message-count>
    <in-message-list-summary>
      <size>2694</size>
      <message-type count="1" id="AZ_HANDSHAKE"/>
      <message-type count="1" id="BT_BITFIELD"/>
      <message-type count="1" id="AZ_PEER_EXCHANGE"/>
      <message-type count="159" id="BT_HAVE"/>
      <message-type count="1" id="BT_HANDSHAKE"/>
      <message-type count="10" id="BT_ALLOWED_FAST"/>
      <message-type count="1" id="BT_UNCHOKE"/>
      <message-type count="2520" id="BT_PIECE"/>
    </in-message-list-summary>
    <out-message-list-summary>
      <size>2685</size>
      <message-type count="1" id="AZ_HANDSHAKE"/>
      <message-type count="1" id="BT_BITFIELD"/>
      <message-type count="1" id="BT_INTERESTED"/>
      <message-type count="157" id="BT_HAVE"/>
      <message-type count="2525" id="BT_REQUEST"/>
    </out-message-list-summary>
  </messages-summary>
  <in-message-list>...</in-message-list>
  <out-message-list>...</out-message-list>
</messages>
```

Listing 5.12: excerpt bigBuckBunny in-message-list

```
<in-message-list>
  <message id="BT_HANDSHAKE">
    <date>Sat Nov 11 17:55:54 CET 2017</date>
    <time>1510419354428</time>
    <description>
      BT_HANDSHAKE of dataID: ...
    </description>
    <payload>...</payload>
  </message>
  <message id="AZ_HANDSHAKE">
    <date>Sat Nov 11 17:55:54 CET 2017</date>
    <time>1510419354429</time>
    <description>
      AZ_HANDSHAKE from ...
    </description>
    <payload>...</payload>
  </message>
  <message id="BT_BITFIELD">
    <date>Sat Nov 11 17:55:54 CET 2017</date>
    <time>1510419354608</time>
    <description>BT_BITFIELD</description>
    <payload>...</payload>
  </message>
  <message id="BT_UNCHOKE">
    <date>Sat Nov 11 17:55:55 CET 2017</date>
    <time>1510419355772</time>
    <description>BT_UNCHOKE</description>
  </message>
   ...
  <message id="BT_PIECE">
    <date>Sat Nov 11 17:55:57 CET 2017</date>
    <time>1510419357017</time>
    <description>
      BT_PIECE data for piece #1589:0-&gt;16383
    </description>
    <payload>...</payload>
  </message>
   ...
</in-message-list>
```

Listing 5.13: excerpt bigBuckBunny out-message-list

```
<out-message-list>
  <message id="AZ_HANDSHAKE">
    <date>Sat Nov 11 17:55:54 CET 2017</date>
    <time>1510419354479</time>
    <description>
      AZ_HANDSHAKE from ...
    </description>
    <payload>...</payload>
  </message>
  <message id="BT_BITFIELD">
    <date>Sat Nov 11 17:55:54 CET 2017</date>
    <time>1510419354479</time>
    <description>BT_BITFIELD</description>
    <payload>...</payload>
  </message>
  <message id="BT_INTERESTED">
    <date>Sat Nov 11 17:55:54 CET 2017</date>
    <time>1510419354631</time>
    <description>BT_INTERESTED</description>
  </message>
  <message id="BT_REQUEST">
    <date>Sat Nov 11 17:55:56 CET 2017</date>
    <time>1510419356423</time>
    <description>
      BT_REQUEST piece #1589:0-&gt;16383
    </description>
    <payload>AAAGNQAAAAAAAEAA</payload>
  </message>
   ...
</out-message-list>
```

Listing 5.14: bigBuckBunny history.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sharehistory>
  <header>...</header>
  <history>
    <entry date="Sat␣Nov␣11␣17:55:54␣CET␣2017">
      <message type="BT_BITFIELD">Bitfield update</message>
      <bitfield>...</bitfield>
      <percent>0.8176100628930818%</percent>
    </entry>
    <entry date="Sat␣Nov␣11␣17:56:06␣CET␣2017">
      <message type="BT_HAVE">
        <piece index="66"/>
      </message>
      <bitfield>...</bitfield>
      <percent>0.8176100628930818%</percent>
    </entry>
     ...
    <entry date="Sat␣Nov␣11␣17:56:33␣CET␣2017">
      <piece_downloaded index="1589"/>
      <bitfield>...</bitfield>
      <percent>0.8805031446540881%</percent>
    </entry>
     ...
    <entry date="Sat␣Nov␣11␣18:32:22␣CET␣2017">
      <message type="BT_HAVE">
        <piece index="219"/>
      </message>
      <bitfield>...</bitfield>
      <percent>10.628930817610064%</percent>
    </entry>
  </history>
</sharehistory>
```

6. Summary and Conclusion

This chapter summarizes this thesis results and describe problems and possible extension of the implemented prototype.

6.1. Summary

This thesis provides a prototype implementation for evidence collection in BitTorrent. It is implemented as a plugin in Vuze, a modern BitTorrent client. To collect useful evidence an external time source is included. Filters for IPs and countries achieve, that only interesting peers are added for observation. When observing a target the complete network traffic between the target and the plugin is stored with timing information. After collecting enough traffic, defined by a threshold, pieces and files are restored. In addition files containing the collected evidence as XML are created. Finally all files are packed and signed to obtain tamper proof evidence.

6.2. Conclusion

The intention of this thesis is, that not all current solutions for evidence collection in P2P networks are that reliable and complete as possible for the use in court. With the prototype implementation there is an example how to tackle the discussed problems in current solutions. It claims not to be the best or most efficient solution, but it considers the criticism points in evidence collection, and show how to do it better.

The tests, running within the two test scenarios, showed the full functionality of the prototype. So all steps of the process of evidence collection, work as expected with all requirements. The resulting evidence produced by the two tests, are admissible, authentic, complete, and reliable and therefore fulfill the chain of custody.

A limitation of the collected evidence is the usability in court, in terms of verification of shared files. In the nature of BitTorrent pieces are not shared as block per file, but distributed over the complete torrent. This means that it is very unlikely to download connected blocks of files. There are clients which are able to to this but it is not usual for non streaming clients. In consequence it depends on the court if the verification of a small part, a bigger continuously parts, or even the whole file is needed to see the file as identified and shared.

6.2.1. Problems and Solutions

During the implementation of the plugin prototype there arose some problems which had to be solved.

Vuze Documentation

Starting implementing a small plugin for Vuze is really easy. On the developers wiki from Vuze[12] there is a plugin development guide available which describes step by step what to do and what is possible. There is even a repository with example plugins and other plugins used in Vuze to look after example code. The problems arose when rarely used functions of the Vuze plugin API where used, because the Java documentation itself is almost a self explanation of the variables and methods without giving more information of the usage and functionality. This sometimes leads to a "try and error" development to get the right methods or listeners when searching a specific functionality of Vuze.

[12]wiki.vuze.com

Extension of BitTorrent protocol

By the implementation of the message reassembler, it is required to extract the shared data out of a BT_PIECE message. So the given *ByteBuffer* has to be split up in a leading meta data section and a shared data section. In the original standard this split point is fixed, but not in the Azureus messaging protocol[13], a BitTorrent extension of Vuze where the length of the meta section is of variable length. As all Vuze installations use this extension when communicating which each other, the prototype has to support it as well. Sadly, there is no out of the box solution to decode the message and get this specific split point to separate the data. So a own method had to be implemented to compute the variable padding section of each message, to get the exact split point.

Multithreading

Inside Vuze the business logic the user interface and the plugins run in different threads, which makes perfectly sense from the view of an software architect. During the implementation there were two problems with conflicting thread access and shared data. The first problem was showing changes in the UI originated inside Vuze, like adding torrents. So the source of a change is Vuze, the data is in the plugin, which is shown in the UI. It was hard to separate the different thread accesses and also the choice of the data structure was not the best. The second problem was when accessing the *ByteBuffer* of received messages. As Vuze and the prototype want to process the data this access had to be synchronized. It was hard to identify the problem, because the exception was not useful in this case and there were no hints in the documentation

6.3. Possible Extensions

The implemented prototype in this thesis is only a very basic version for evidence collection in P2P networks. There are many possible optimizations and extensions

[13]wiki.vuze.com/w/Azureus_messaging_protocol

regarding the functionality and usability, which are not directly covered. Three of them are described in detail.

Optimize the download strategy

As mentioned in the conclusion, with the BitTorrent protocol usually files are not downloaded continuously. This can cause a very wide spread distribution of the single pieces. But this behavior depends on the piece download strategy of the specific clients. In the simplest case pieces are downloaded by their order, which is, as an example, perfect for streaming a movie to start watching without having the whole file. Also thinkable would be a random order, but this strategy is also not optimal in term of availability and speed of a torrent. The most clients use implementations of the rarest piece first strategy, which keeps a BitTorrent swarm healthy. In reference to this thesis it would be thinkable to implement a download strategy to download specific files continuously until a threshold is reached. So it would be possible to download to a certain, usable degree, or even whole files for better verification, and especially presentation.

Removing downloaded data from torrent

In the current implementation the BitTorrent process is not influenced by the plugin, with the only exception of intercepting outgoing BT_PIECE messages. In consequence the complete download and disk storing of pieces is covered by Vuze. To remove already downloaded and verified pieces from the torrent, access to the *diskmanager* is needed, which is not provided by the Vuze plugin API. There are ways to access it, but for compatibility and simplicity only the plugin API was used in this thesis. There are other possible solutions to intercept or tamper incoming BT_PIECE messages, but that would have a huge side effect on the health of a peer. That is because a peer continuously sending incorrect data, or allegedly does not send requested pieces at all, will be dropped.

Compare files with the external provided original files

The verification if files are bit wise identical, or even parts of them, is trivial by hash values or direct comparison. But as digital files with the same content can be stored in various formats and compressions it is a non trivial task to compare a file by its content. So as a possible extension all files of an observed torrent could be downloaded and verified by a third party. This "original" files are then available in the plugin to have a reference to check and verify all random pieces, or even single messages.

Bibliography

[1] The biggest ever bittorrent leak: Mediadefender internal emails go public. https://torrentfreak.com/mediadefender-emails-leaked-070915/, September 2007.

[2] LG Köln Urteil vom 31. Oktober 2012 - Az. 28 O 306/11. https://openjur.de/u/611182.html, 2013.

[3] Bittorrentspecification. https://wiki.theory.org/BitTorrentSpecification, February 2017.

[4] Plugin Development Guide. https://wiki.vuze.com/w/Plugin_Development_Guide, February 2017.

[5] Holger Bleich. Redtube-Abmahnungen: Gericht ging Briefkastenfirma auf den Leim. https://heise.de/-2103801, January 2014.

[6] Eoghan Casey. *Digital Evidence and Computer Crime*. Academic Press, 2004.

[7] Ole Damm. LG Hamburg: Ausdruck der Ermittlungsfirma beweist nicht illegales Filesharing. http://www.damm-legal.de/lg-hamburg-ausdruck-der-ermittlungsfirma-beweist-nicht-illegales-filesharing, August 2009.

[8] European Court of Justice (Second Chamber). Judgement from 26.04.2017 - c-527/15, April 2017.

[9] LG Berlin. Urteil vom 03.05.2011, Az. 16 O 55/11, 2011.

[10] LG Hamburg. Urteil vom 14.03.2008 - Az. 308 O 76/07, 2008.

[11] LG Köln. Urteil vom 31.10.2012 - az. 28 o 306/11, October 2012.

[12] Georg Linhard. Sicherheitsaspekte von Peer-to-Peer Netzwerken. mathesis, Johannes Kepler University, October 2006.

[13] Marlom A. Konrath, and Marinho P. Barcellos, and Rodrigo B. Mansilha. Attacking a swarm with a band of liars: evaluating the impact of attacks on bittorrent. *2007 7th International Conference on Peer-to-Peer Computing*, 00:37–44, 2007.

[14] Jerry Norton and Heinrich Nagel. evidence. Encyclopedia Britannica, May 2017.

[15] Bettina Nunner-Krautgasser and Philipp Anzenberger. *Evidence in Civil Law - Austria*. Institute for Local Self-Government and Public Procurement Maribor, 2015.

[16] Ole Damm. LG Berlin: Die Ermittlung von IP-Adressen für Filesharing-Abmahnungen kann unzuverlässig sein / Eine Krähe hackt der anderen ein Auge aus. http://www.damm-legal.de/lg-berlin-die-ermittlung-von-ip-adressen-fuer-filesharing-abmahnungen-kann-unzuverlaessig-sein-eine-kraehe-hackt-der-anderen-ein-auge-aus, November 2012.

[17] Sandvine. Global internet phenomena report 1h 2013. *Global Internet Phenomena*, page 13, 2013.

[18] Sandvine. Global internet phenomena 2015 asia - pacific & europe. *Global Internet Phenomena*, page 3, 2015.

[19] Michael Sonntag. Introduction to computer forensics. *JKU*, 2012.

[20] The Java Tutorials. Lesson: Generating and verifying signatures. https://docs.oracle.com/javase/tutorial/security/apisign/index.html, 2015.

[21] Wikipedia. Vuze — wikipedia, the free encyclopedia, 2017. [Online; accessed 25-May-2017].

[22] Kai Hwang Xiaosong Lou. Collusive piracy prevention in p2p content delivery networks. *IEEE Transactions on Computers*, 58:970 – 983, February 2009.

Lebenslauf

Persönliche Daten:

Name:	Alexander Lemmé BSc
Anschrift:	Lorch 7
	4470 Enns
Telefon:	0650/8933389
Mail:	lemme@gmx.at
Geburtsdatum:	8.10.1989 in Linz
Familienstand:	ledig
Staatsbürgerschaft:	Österreich

Schulausbildung:

2013 - 2018	Master Studium Network & Security an der Johannes Kepler Universität Linz
2009-2013	Bachelor Studium Informatik an der JKU Linz
22. Juni 2009	Reife- und Diplomprüfung mit gutem Erfolg
2004 - 2009	HTL Traun für Kommunikations- und Informationstechnologie
2000 - 2004	Bundesrealgymnasium Enns
1996 - 2000	Volksschule Enns

Bisherige Tätigkeiten:

Seit Juni 2016	Softwareentwickler bei CGM Arztsysteme Österreich GmbH
August & September 2013	Ferialarbeit bei CGM Arztsysteme Österreich GmbH
Mai 2013	Bachelorarbeit : „Regelbasiertes Reasoning zur Erstellung eines Lebenslaufs auf Basis von Facebook & LinkedIn" beim Projekt TheHiddenU am Institut für Telekooperation
1.August - 30.November 2012	Projektpraktikum beim Projekt TheHiddenU am Institut für Telekooperation
22.August - 28.August 2011	Ferialarbeit am Institut RICAM
1.März - 30.Juni 2010	Arbeit während des Semesters am Institut RICAM
16.Juli - 31.Juli 2009	Ferialarbeit am Institut RICAM
1.Juli - 31.Juli 2008	Schulpraktikum am Institut RICAM
2.Juli - 27.Juli 2007	Schulpraktikum am Institut RICAM der Österreichische Akademie der Wissenschaften: Unterstützung des Systemadministrators
3.Juli - 31. Juli 2006	Siemens Transportation Systems Wien

Aktivitäten:

Seit 2010	Jugendleiter bei der Pfadfindergruppe Enns
2.-6. September 2015	Live-Ticker GAC World Tour, Austrian Open, Wels für unas media productions
31.März - 7.April 2014	Live-Ticker ITTF Spanish Open, Almeria für unas media productions

Sonstiges:

Führerschein Klasse B

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, February 2018

A. Appendix

A.1. info.xsd

Listing A.1: info.xsd

```
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ip" type="xs:string"/>
  <xs:element name="id" type="xs:string"/>
  <xs:element name="date" type="xs:string"/>
  <xs:element name="client" type="xs:string"/>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="infohash" type="xs:string"/>
  <xs:element name="peer">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ip"/>
        <xs:element ref="id"/>
        <xs:element ref="date"/>
        <xs:element ref="client"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="torrent">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="infohash"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="progress">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:short"
                        name="piece-count"
                        use="optional"/>
          <xs:attribute type="xs:short"
                        name="total-piece-count"
                        use="optional"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="start-time" type="xs:string"/>
  <xs:element name="end-time" type="xs:string"/>
  <xs:element name="file">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="progress"/>
        <xs:element ref="start-time" minOccurs="0"/>
        <xs:element ref="end-time" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute type="xs:string"
                    name="complete"
                    use="optional"/>
      <xs:attribute type="xs:string"
                    name="name"
                    use="optional"/>
      <xs:attribute type="xs:string"
                    name="size"
```

```
                          use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="header">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="peer"/>
        <xs:element ref="torrent"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="file-list">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="file"
                          maxOccurs="unbounded"
                          minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="time-server">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:float"
                            name="offset"/>
          <xs:attribute type="xs:string"
                            name="time-server"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="info">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref="header"/>
        <xs:element ref="file-list"/>
        <xs:element ref="time-server"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.2. history.xsd

Listing A.2: history.xsd

```
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ip" type="xs:string"/>
  <xs:element name="id" type="xs:string"/>
  <xs:element name="date" type="xs:string"/>
  <xs:element name="client" type="xs:string"/>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="infohash" type="xs:string"/>
  <xs:element name="peer">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ip"/>
        <xs:element ref="id"/>
        <xs:element ref="date"/>
        <xs:element ref="client"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="torrent">
    <xs:complexType>
      <xs:sequence>
```

```
            <xs:element ref="name"/>
            <xs:element ref="infohash"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="message">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute type="xs:string"
                              name="type"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>
<xs:element name="bitfield" type="xs:float"/>
<xs:element name="percent" type="xs:string"/>
<xs:element name="piece_downloaded">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute type="xs:short"
                              name="index"
                              use="optional"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>
<xs:element name="entry">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="message"
                        minOccurs="0"/>
            <xs:element ref="piece_downloaded"
                        minOccurs="0"/>
        <xs:element ref="bitfield"/>
        <xs:element ref="percent"/>
      </xs:sequence>
      <xs:attribute type="xs:string"
                    name="date"
                    use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="header">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="peer"/>
        <xs:element ref="torrent"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="history">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="entry"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="sharehistory">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="header"/>
        <xs:element ref="history"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.3. messages.xsd

Listing A.3: messages.xsd

```
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ip" type="xs:string"/>
  <xs:element name="id" type="xs:string"/>
  <xs:element name="date" type="xs:string"/>
  <xs:element name="client" type="xs:string"/>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="infohash" type="xs:string"/>
  <xs:element name="peer">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ip"/>
        <xs:element ref="id"/>
        <xs:element ref="date"/>
        <xs:element ref="client"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="torrent">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="infohash"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="size" type="xs:short"/>
  <xs:element name="message-type">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
```

```
          <xs:attribute type="xs:short"
                        name="count"
                        use="optional"/>
          <xs:attribute type="xs:string"
                        name="id"
                        use="optional"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="total-message-amount"
              type="xs:short"/>
  <xs:element name="in-message-list-summary">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="size"/>
        <xs:element ref="message-type"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="out-message-list-summary">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="size"/>
        <xs:element ref="message-type"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="time" type="xs:long"/>
  <xs:element name="description" type="xs:string"/>
  <xs:element name="payload" type="xs:string"/>
```

```
<xs:element name="message">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="date"/>
      <xs:element ref="time"/>
      <xs:element ref="description"/>
      <xs:element ref="payload" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:string"
                  name="id"
                  use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="header">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="peer"/>
      <xs:element ref="torrent"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="messages-summary">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="total-message-amount"/>
      <xs:element ref="in-message-list-summary"/>
      <xs:element ref="out-message-list-summary"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="in-message-list">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="message"
                  maxOccurs="unbounded"
```

```
                              minOccurs="0"/>
          </xs:sequence>
       </xs:complexType>
    </xs:element>
    <xs:element name="out-message-list">
       <xs:complexType>
          <xs:sequence>
             <xs:element ref="message"
                              maxOccurs="unbounded"
                              minOccurs="0"/>
          </xs:sequence>
       </xs:complexType>
    </xs:element>
    <xs:element name="messages">
       <xs:complexType>
          <xs:sequence>
             <xs:element ref="header"/>
             <xs:element ref="messages-summary"/>
             <xs:element ref="in-message-list"/>
             <xs:element ref="out-message-list"/>
          </xs:sequence>
       </xs:complexType>
    </xs:element>
</xs:schema>
```