City Research Online

City St George's, University of London

Citation: Adamsky, Florian (2016). Analysis of bandwidth attacks in a bittorrent swarm. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/16158/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

FLORIAN E. W. ADAMSKY

ANALYSIS OF BANDWIDTH ATTACKS IN A BITTORRENT SWARM

ANALYSIS OF BANDWIDTH ATTACKS IN A BITTORRENT SWARM

FLORIAN E. W. ADAMSKY

CITY UNIVERSITY
LONDON

A thesis submitted for the degree of
Doctor of Philosophy (Information Engineering)

School of Mathematics, Computer Science & Engineering
City University London

August 2016 – version 1.1

Florian E. W. Adamsky: *Analysis of Bandwidth Attacks in a BitTorrent Swarm,* A thesis submitted for the degree of, Doctor of Philosophy (Information Engineering), © August 2016

SUPERVISORS:
Prof. Dr. Muttukrishnan Rajarajan
Dr. Syed Ali Khayam

LOCAL SUPERVISOR:
Prof. Dr. Rudolf Jäger

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to the loving memories of
Petra Claire Adamsky

1955 – 2015

and
Eduard Walter Adamsky.

1950 – 2016

ABSTRACT

The beginning of the 21st century saw a widely publicized lawsuit against *Napster*. This was the first Peer-to-Peer software that allowed its users to search for and share digital music with other users. At the height of its popularity, Napster boasted 80 million registered users. This marked the beginning of a Peer-to-Peer paradigm and the end of older methods of distributing cultural possessions. But *Napster* was not entirely rooted in a Peer-to-Peer paradigm. Only the download of a file was based on Peer-to-Peer interactions; the search process was still based on a central server. It was thus easy to shutdown Napster. Shortly after the shutdown, *Bram Cohen* developed a new Peer-to-Peer protocol called *BitTorrent*.

The main principle behind BitTorrent is an incentive mechanism, called a *choking algorithm*, which rewards peers that share. Currently, BitTorrent is one of the most widely used protocols on the Internet. Therefore, it is important to investigate the security of this protocol. While significant progress has been made in understanding the BitTorrent choking mechanism, its security vulnerabilities have not yet been thoroughly investigated. This dissertation provides a security analysis of the Peer-to-Peer protocol BitTorrent on the application and transport layer.

The dissertation begins with an experimental analysis of bandwidth attacks against different choking algorithms in the BitTorrent seed state. I reveal a simple exploit that allows malicious peers to receive a considerably higher download rate than contributing leechers, thereby causing a significant loss of efficiency for benign peers. I show the damage caused by the proposed attack in two different environments—a lab testbed comprised of 32 peers and a global testbed called *PlanetLab* with 300 peers. Our results show that three malicious peers can degrade the download rate by up to 414.99 % for all peers. Combined with a Sybil attack with as many attackers

as leechers, it is possible to degrade the download rate by more than 1000 %. I propose a novel choking algorithm which is immune against bandwidth attacks and a countermeasure against the revealed attack.

This thesis includes a security analysis of the transport layer. To make BitTorrent more Internet Service Provider friendly, BitTorrent Inc. invented the Micro Transport Protocol. It is based on User Datagram Protocol with a novel congestion control called Low Extra Delay Background Transport. This protocol assumes that the receiver always provides correct feedback, otherwise this deteriorates throughput or yields to corrupted data. I show through experimental evaluation, that a misbehaving Micro Transport Protocol receiver which is not interested in data integrity, can increase the bandwidth of the sender by up to five times. This can cause a congestion collapse and steal a large share of a victim's bandwidth. I present three attacks, which increase bandwidth usage significantly. I have tested these attacks in real world environments and demonstrate their severity both in terms of the number of packets and total traffic generated. I also present a countermeasure for protecting against these attacks and evaluate the performance of this defensive strategy.

In the last section, I demonstrate that the BitTorrent protocol family is vulnerable to Distributed Reflective Denial-of-Service attacks. Specifically, I show that an attacker can exploit BitTorrent protocols (Micro Transport Protocol, Distributed Hash Table, Message Stream Encryption and BitTorrent Sync to reflect and amplify traffic from BitTorrent peers to any target on the Internet. I validate the efficiency, robustness, and the difficulty of defence of the exposed BitTorrent vulnerabilities in a Peer-to-Peer lab testbed. I further substantiate lab results by crawling more than 2.1 million IP addresses over Mainline Distributed Hash Table and analyzing more than 10,000 BitTorrent handshakes. The experiments suggest that an attacker is able to exploit BitTorrent peers to amplify traffic by a factor of 50, and in the case of BitTorrent Sync 120. Additionally, I observe that the most popular BitTorrent clients are the most vulnerable ones.

PUBLICATIONS

Some ideas and figures have appeared previously in the following peer-reviewed publications:

CONFERENCE PAPERS

Adamsky, F.; Khayam, A.; Jäger, R.; Rajarajan, M.: P2P File-Sharing in Hell: Exploiting BitTorrent Vulnerabilities to Launch Distributed Reflective DoS Attacks. In. *9th USENIX Workshop on Offensive Technologies (WOOT), 2015, Washington, D.C., USA*

Adamsky, F.; Khayam, A.; Jäger, R.; Rajarajan, M.: Who is going to be the next BitTorrent Peer Idol? In. *12th IEEE International Conference on Embedded and Ubiquitous Computing, 2014, Milan, Italy.*

Adamsky, F.; Khayam, A.; Jäger, R.; Rajarajan, M.: Security Analysis of the Micro Transport Protocol with a Misbehaving Receiver. In. *4th IEEE International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, 2012, Sanya, Hainan, China.*

Adamsky, F.; Khan, H.; Rajarajan, M.; Khayam, A.; Jäger, Rudolf: Poster: Destabilizing BitTorrent's Clusters to Attack High Bandwidth Leechers. In. *18th ACM Conference on Computer and Communications Security, 2011, Chicago, USA.*

JOURNAL PAPERS

Adamsky, F.; Khayam, A.; Jäger, R.; Rajarajan, M.: Stealing Bandwidth from BitTorrent Seeders. In. *In Elsevier Journal Computers & Security, 2014*

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [Knu74]

ACKNOWLEDGMENTS

This research project would not be possible with the support and the help of many people. My special thanks go to Prof. Dr. Muttukrishnan Rajarajan for supervising this project as well as Dr. Syed Ali Khayam and Prof. Dr. Rudolf Jäger for providing me with valuable comments, suggestions and discussions. I also would like to thank a few students and colleagues for stimulating discussions and collaborations: Christopher Köhnen, Christian Köbel and Hassan Khan.

Finally, I would like to thank my family, friends and my girlfriend Claudia for supporting me.

CONTENTS

I INTRODUCTION 1

1 INTRODUCTION TO SECURITY OF BITTORRENT 3

1.1 Motivation . . . 5

1.2 Challenges in P2P security . . . 6

1.3 Research Hypothesis and Objectives . . . 7

1.4 Contribution of this Thesis . . . 8

1.4.1 Vulnerability Analysis of the BitTorrent protocol 8

1.4.2 Sybil Attack in a BitTorrent Swarm . . . 9

1.4.3 Novel Seeding Algorithm . . . 9

1.5 Thesis Structure . . . 9

2 BACKGROUND ON PEER-TO-PEER AND BITTORRENT 11

2.1 Terminology . . . 11

2.2 Peer-to-Peer Paradigm . . . 12

2.2.1 Unstructured Overlays . . . 14

2.2.2 Structured Overlays . . . 19

2.3 BitTorrent Overview . . . 21

2.3.1 Choking Algorithm . . . 23

2.3.2 Rarest Piece First Algorithm . . . 25

2.3.3 Micro Transport Protocol (uTP) . . . 26

3 LITERATURE SURVEY AND RELATED WORK 31

3.1 Service Availability . . . 31

3.1.1 Sybil and Eclipse Attacks . . . 31

3.1.2 Distributed Denial of Service Attacks . . . 34

3.1.3 Bandwidth and Connection Attacks . . . 35

3.2 Document Authentication . . . 36

3.2.1 Content Pollution Attacks . . . 37

3.2.2 Metadata Pollution Attacks . . . 37

3.2.3 Index Poisoning . . . 37

3.3 Anonymity of the Peers . . . 38

3.3.1 Friend-to-Friend Network . . . 39

3.3.2 Dining Cryptographers . . . 40
3.3.3 Onion/Garling Routing . . . 41

II ANALYSIS OF THE ATTACKS 45

4 METHODOLOGY 47
4.1 Local P2P Testbed System . . . 47
4.1.1 Hardware . . . 48
4.1.2 Software . . . 48
4.2 PlanetLab – Global Research Network . . . 53
4.2.1 Architecture . . . 53
4.3 Discussion and Related Testbed Systems . . . 54

5 STEALING BANDWIDTH FROM BITTORRENT PEERS 55
5.1 Introduction . . . 55
5.2 Background . . . 55
5.2.1 Allowed Fast Extension . . . 56
5.2.2 Seeding Algorithms . . . 57
5.3 Bandwidth Attacks in Detail . . . 58
5.3.1 Seeding Algorithms . . . 60
5.3.2 Programming Errors in Clients . . . 61
5.3.3 Allowed Fast Extension Attack . . . 62
5.4 Experimental Evaluation . . . 64
5.4.1 Experimentation on Local Cluster Testbed . . . 64
5.4.2 Experimentation with PlanetLab Testbed . . . 69
5.4.3 Discussion . . . 70
5.5 Countermeasures . . . 71
5.5.1 Allowed Fast Attack . . . 71
5.6 Novel Seeding Algorithm: Peer Idol . . . 72
5.6.1 Algorithm Details . . . 73
5.6.2 Implementation Details . . . 74
5.6.3 Experimental Evaluation . . . 75
5.7 Comparison with Related Work . . . 81
5.8 Summary . . . 82

6 EXPLOIT BANDWIDTH TO CREATE CONGESTION 85
6.1 Introduction . . . 85
6.2 Attack Scenarios . . . 85

6.2.1 Congestion Collapse . . . 85
6.2.2 Steal Bandwidth . . . 87
6.3 Details and Evaluation of Attacks . . . 87
6.3.1 Lying about the One-way Delay . . . 88
6.3.2 Lazy Optimistic Acknowledgment . . . 90
6.3.3 Optimistic Acknowledgment . . . 91
6.4 Discussion of the Proposed Attacks . . . 92
6.4.1 Impact of Attacks . . . 92
6.4.2 Comparison of the Attacks . . . 93
6.5 Countermeasures . . . 94
6.5.1 Randomly Skipped Packets . . . 94
6.5.2 Delay Attack . . . 97
6.6 Comparison with Related Work . . . 98
6.6.1 Exploitation Congestion Avoidance Control . . . 98
6.6.2 LEDBAT Performance . . . 99
6.7 Summary . . . 99
7 REDIRECT BANDWIDTH FROM PEERS TO ARBITRARY VICTIMS 101
7.1 Introduction . . . 101
7.2 Background . . . 101
7.2.1 Distributed Reflective Denial-of-Service Attacks 102
7.3 Methodology . . . 103
7.3.1 Testbed System . . . 103
7.3.2 Micro Transport Protocol (uTP) Wireshark Dissector Plugin . . . 104
7.3.3 Analyse Script . . . 104
7.4 Details of the Attack . . . 105
7.4.1 Two-way Handshake in uTP . . . 105
7.4.2 Exploiting BitTorrent Handshake via uTP . . . 106
7.4.3 Exploiting Message Stream Encryption Handshake . . . 110
7.4.4 Exploiting Distributed Hash Table Messages . . . 112
7.4.5 Exploiting BitTorrent Sync . . . 114
7.4.6 Discussion . . . 115
7.5 Experimental Evaluation . . . 116

7.5.1 Efficiency 116
7.5.2 Robustness 117
7.5.3 Evadability 121
7.6 Countermeasures and Mitigation Strategies 122
7.6.1 Three-way Handshake over uTP 123
7.6.2 Disable Packet Stuffing for the First Data Packet 123
7.6.3 Enforcing a Valid ACK as a Third Packet 123
7.6.4 Distributed Hash Table 124
7.7 Comparison with Related Work 124
7.8 Summary 125

iii CONCLUSION 127

8 CONCLUSION 129
8.1 Research Hypothesis and Objectives 129
8.1.1 Research Hypothesis 129
8.1.2 Thesis Objectives 130
8.2 Summary of the Achievements 131
8.3 Future Directions 132

iv APPENDIX 133

A APPENDIX 135

BIBLIOGRAPHY 141

LIST OF FIGURES

Figure 2.1 Graphical differentiation between client-server and Peer-to-Peer (P2P) paradigm. Circles with a C are clients, S is a server, and P are peers. There is no central entity in (b) compared to (a). 13

Figure 2.2 Unstructured overlay that routes all queries according to the flooding algorithm. Every query q contains a Time To Live (TTL) value to limit its lifespan. In this example, Peer P_1 sends a query q with a TTL value of 2 to all of its neighbors. 15

Figure 2.3 Unstructured overlay that routes all queries according to the random walk algorithm. Every query q contains a Time To Live (TTL) value to limit its lifespan. In this example, Peer P_1 sends a query q with a TTL value of 3 to a random peer from its peer set. 16

Figure 2.4 Basic functionality of a download with Napster. 18

Figure 2.5 Routing table from node 110 for a Distributed Hash Table network with an address space of 2^3. 20

Figure 2.6 A file F which is split into pieces with its Secure Hash Algorithm 1 (SHA-1) hashes. . . . 22

Figure 2.7 Peer P_1–P_4 represent participating peers and p_1–p_4 represent a piece of the file. Pieces with a blue background color are pieces which actual peers have and pieces with the a white background are missing pieces. 23

Figure 2.8 Ideal traffic flow with foreground traffic (red) and Micro Transport Protocol background traffic (blue). 28

Figure 2.9 Version 1 header of a Micro Transport Protocol (uTP) packet. 28
Figure 2.10 Two-way handshake to initiate a connection between two uTP nodes. The text on the outer edges reflects the state of the protocol. 29
Figure 2.11 Two-way handshake to initiate a connection between two uTP nodes. The text on the outer edge reflects the state of the protocol. 29
Figure 3.1 Example of an eclipse attack against peer P_4. Red circles are attackers and blue circles are benign peers. 32
Figure 3.2 Friend-to-friend network where each peer is connected to a small number of friends. 40
Figure 3.3 With Dining Cryptographers, an author can publish an document or a message m without revealing its identity. 41
Figure 3.4 Onion Router (O_1–O_3) receives a message and decrypts its content. The content consists of a readable header and an encrypted payload. A router sends the encrypted payload to the next router according to the header. 42
Figure 4.1 A network diagram of the testbed system. The testbed system consists of peer $P_1 - P_{32}$ and a controller (ctrl). 47
Figure 4.2 Photograph of the testbed with 32 peers, 1 controller peer, and a separate switch. 49
Figure 4.3 A diagram of the architecture of Thayeria. . . . 51
Figure 5.1 File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has a 1 Mbps upload limit. (a) A normal bandwidth attack. (b) A bandwidth attack combined with the Allowed Fast Attack. The error bars show 95 % confidence intervals. 65

Figure 5.2 File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has 5 Mbps upload limit. (a) A normal bandwidth attack. (b) A bandwidth attack combined with the allowed fast attack. The error bars show 95 % confidence intervals. 66

Figure 5.3 File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has 10 Mbps upload limit. (a) A normal bandwidth attack. (b) A bandwidth attack combined with the allowed fast attack. The error bars shows 95 % confidence intervals. 67

Figure 5.4 File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has a 5 Mbps upload limit. A Sybil attack in which I increased the number of attackers and reduced the number of leechers with every iteration. . . 68

Figure 5.5 File transfer on PlanetLab of a 100 MiB file with a piece size of 64 KiB via BitTorrent. The data was produced with 1 seeder with a 1 Mbps upload limit, 300 leechers with 1 Mbps download limits, and 4 malicious peers. All values are the average values of ten iterations. The error bars show 95 % confidence intervals. 70

Figure 5.6 File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder with a 1 Mbps upload limit. (a) Allowed fast attack. (b) Allowed fast attack with a patched seeder. . . 72

Figure 5.7 Effects of seeding algorithms on the average download speed in different environments. The upload speed of the seeder was gradually increased. The error bars show 99 % confidence intervals. 76

Figure 5.8 The number of peers connected to the seeder in a swarm of 32 peers. All values are average values of ten iterations. The error bars show 99 % confidence intervals. Notice that it can be observed that a seeder that makes use of Peer Idol (PI) has more connections to other peers than the other seeding algorithms. This improves the stability and robustness of BitTorrent. 77
Figure 5.9 The unchoke ratio of attackers and leechers in different seeding algorithms without an optimistic unchoke slot. The data was produced with 1 seeder with a 5 Mbps upload limit, 29 leechers with 900 kbit/s download limits and 3 malicious peers. All values are averages of ten iterations. 78
Figure 5.10 Average download speed of the different seeding algorithm with 3 attackers with no upload or download limit and 29 leechers with an upload limit of 20 Mbps. The error bars show 99 % confidence intervals. 80
Figure 6.1 A possible scenario where an attacker exploits the congestion control to create congestion on a slow path. The blue lines represent the traffic from an attacker and the red lines represent traffic generated a victim. 86
Figure 6.2 One-way delay measurement with a normal receiver and an attacker. 88
Figure 6.3 File transfer of a 100 MiB file via uTP under the following network conditions: 100 Mbps bandwidth, 25 ms delay, 10 ms variance, and normal distribution. 90
Figure 6.4 Comparison of the input buffer of a normal receiver and an attacker. 91

Figure 6.5 Comparison of attacks based on the bandwidth produced. All values are the averages of 10 iterations. The error bars show 95 % confidence intervals. 94
Figure 6.6 All values are averages of 10 iterations. The error bars show 95 % confidence intervals. 95
Figure 6.7 File Transfer of a 100 MiB File via uTP under the following network conditions: 100 Mbps bandwidth, 25 ms delay, 10 ms variance and a normal Distribution. It takes approximately 3–5 s to detect both attacks and terminate the connection. 96
Figure 6.8 File Transfer of a 100 MiB File via uTP under the following network conditions: 100 Mbps bandwidth, 25 ms delay, 10 ms variance and a normal distribution. 97
Figure 7.1 Schematic diagram of the threat model of a Distributed Reflective Denial-of-Service (DRDoS) attack. 102
Figure 7.2 In uTP the initiator sends a ST_SYN packet to the receiver. The receiver acknowledges this with a ST_STATE packet. The connection is then established (two-way handshake). 107
Figure 7.3 Diffie-Hellman handshake of two BitTorrent peers that make use of Message Stream Encryption to avoid traffic shaping. 111
Figure 7.4 Screenshot of the network analyses tool *Wireshark* that displays responses from a BitTorrent Sync node to a single PING request. 115
Figure 7.5 Amplification attack with one attacker, 31 amplifiers and one victim. The blue line shows the payload size that the attacker sends to the amplifier. The red line shows the payload size that the amplifier sends to the victim. 117

Figure 7.6 Architecture and supervision tree of the BitTorrent Crawler. Every rectangle is a process. DB denotes the database. 118
Figure 7.7 Histogram of the payload size from the Distributed Hash Table responses which are caused by GET_PEERS requests. The numbers above the bars are the average Bandwidth Amplification Factor values. 119
Figure 7.8 Numbers above the peaks are Bandwidth Amplification Factor (BAF) values; the numbers in brackets are the relative frequency. It can be observed that three packet sizes stand out: 665 bytes, 1000 bytes, and 1438 bytes. These three packet sizes would be the most often response in an amplification attack. 120
Figure A.1 File transfer of a 300 MiB File via uTP and parallel to that, a constant User Datagram Protocol (UDP) stream of 50 Mbps under the following network conditions: Bandwidth: 100 Mbps half duplex and Delay: 0 ms. 140

LIST OF TABLES

Table 4.1 Hardware specification of every machine in the testbed system. 48
Table 5.1 BitTorrent clients in combination with the seeding algorithm used ordered by market share. . 60
Table 5.2 BitTorrent clients ordered by market share according to [Van11b]. Column *Vulnerable* shows if the client supports the allowed fast semantics and if it is vulnerable to the proposed attack. 63

Table 7.1 Client software and version number of the inspected Libtorrent Extension Protocol messages. 121
Table 7.2 Comparison of amplification vulnerabilities where an X denotes the firewall technology with which it can be defended. 122
Table A.1 Summary of all BAF and PAF values of the inspected BitTorrent messages. 135

LIST OF LISTINGS

Listing 4.1 Shell one-liner to start a Thayeria instance. . . 51
Listing 4.2 Example of the source code of a controller peer with Thayeria. 52
Listing 6.1 Source code of the maximal window calculation according to BEP 29 [Nor10]. 89
Listing 7.1 Example of the generation of an uTP packet with Perl. 104
Listing A.1 Source code of PI implemented as a libtorrent extension. 135

LIST OF ALGORITHMS

Algorithm 5.1 Algorithm that generates the allowed fast set according to BEP 6 [HC08]. 56
Algorithm 5.2 Pseudocode of the Fast Extension Attack. . . . 62

ACRONYMS

ACK	Acknowledgment
AIMD	Additive-Increase/Multiplicative-Decrease
AL	Anti Leech
AMP	Azureus Message Protocol
ARPANET	Advanced Research Projects Agency Network
BAF	Bandwidth Amplification Factor
BC	Borda Count
BEP	BitTorrent Enhancement Proposals
BTSync	BitTorrent Sync
CIDR	Classless Inter-Domain Routing
CM	Condorcet Method
CharGen	Character Generator Protocol
DC	Dining Cryptographers
DDoS	Distributed Denial-of-Service
DHT	Distributed Hash Table
DNS	Domain Name System
DPI	Deep Packet Inspection
DRDoS	Distributed Reflective Denial-of-Service
DSL	Digitial Subscriber Line
DoS	denial-of-service
EMT	Extended Message Type

EVM	Erlang Virtual Machine
F2F	Friend-to-Friend
FTP	File Transfer Protocol
FU	Fastest Upload
GNU	GNU's Not Unix
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I2P	Invisible Internet Project
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPv4	Internet Protocol (IP) version 4
IPv6	IP version 6
ISP	Internet Service Provider
LAN	Local Area Network
LEDBAT	Low Extra Delay Background Transport
LPD	Local Peer Discovery
LTEP	Libtorrent Extension Protocol
LTS	Long Term Support
LW	Longest Waiter
MITM	man-in-the-middle
MLDHT	Mainline Distributed Hash Table
MP3	MPEG-1/2 Audio Layer III
MSB	Most Significant Bit

MSE	Message Stream Encryption
NFS	Network File System
NIS	Network Information System
NTP	Network Time Protocol
Opt-ACK	optimistic Acknowledgment (ACK)
OS	Operating System
P2P	Peer-to-Peer
PAF	Packet Amplification Factor
PEX	Peer Exchange
PI	Peer Idol
PXE	Preboot Execution Environment
QOTD	Quote of the Day
RFC	Request for Comments
RF	Round Robin (RR) (fixed)
RIAA	Recording Industry Association of America
RMON	Remote Network Monitoring
RR	Round Robin
RPC	Remote Procedure Call
RTT	Round-trip Time
SACK	Selective Acknowledgment
SDoS	single-source denial-of-service (DoS)
SHA-1	Secure Hash Algorithm 1
SNMP	Simple Network Management Protocol
SPI	Stateful Packet Inspection

SPOF	Single Point of Failure
SSDP	Simple Service Discovery Protocol
SSH	Secure Shell
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TOR	The Onion Router
TTL	Time To Live
UDP	User Datagram Protocol
URL	Uniform Resource Locator
uTP	Micro Transport Protocol
VDHT	Vuze Distributed Hash Table (DHT)
VM	Virtual Machine
VoIP	Voice over IP
WoW	World of Warcraft
XOR	Exclusive-or

Part I

INTRODUCTION

"The true delight is in the finding out rather than in the knowing."

—Issac Asimov

1

INTRODUCTION TO SECURITY OF BITTORRENT

At the beginning of the 21st century, there was a widely publicized lawsuit against *Shawn Fanning*, which shaped public misunderstanding of the Peer-to-Peer (P2P) paradigm. Fanning, a computer hacker, programmed the first P2P software called *Napster*. This software allowed its users to search and share digital music, mostly MPEG-1/2 Audio Layer III (MP3) files, with other participants. *Napster*, at the height of its popularity, had 80 million users registered [Gre02]. The Recording Industry Association of America (RIAA) made with a lawsuit against Napster an example of *Fanning*. This was the beginning of the P2P paradigm and the end of old modes of distributing cultural possessions.

Public understanding of P2P is often connected to illegal file sharing. Even when file sharers make use of P2P technology, it is not restricted to illegal file sharing. Behind the term P2P hides a network paradigm that can be used for a variety of actions, such as video and audio streaming, Voice over IP (VoIP), instant and chat messaging, and file sharing, among others. File sharing can also be legal: nearly all GNU's Not Unix (GNU) Linux distributions provide their Operating System (OS) via BitTorrent[1][2][3]. Artist, musicians and film makers use this technology to distribute their work to gain popularity. Moreover, companies such as Facebook [Van11a] and Twitter use P2P technology to update thousands of their servers. Twitter has even noted that their server deployment with BitTorrent is 75 times faster compared to their previous central solution [Van10]. The game manufacturer Blizzard Entertainment uses BitTorrent via a proprietary client called *Blizzard Downloader*[4] to distribute games like World of Warcraft (WoW),

1 http://www.ubuntu.com/download/alternative-downloads
2 http://www.slackware.com/torrents/
3 https://www.debian.org/CD/torrent-cd/
4 http://www.wowpedia.org/Blizzard_Downloader

Diablo 3, and StarCraft 2. Such examples demonstrate that P2P technology is often deployed in situations which require a large amounts of data and/or receivers.

This technology can also be used to distribute politically sensitive material. On April 5th 2010, the platform *Wikileaks* released the video *Collateral Murder*, depicting the killing of over a dozen people in New Baghdad. *Wikileaks* distributed this video via Hypertext Transfer Protocol (HTTP), and with BitTorrent[5]. Even though legal measures were used to shut down *Wikileaks*, the prevention of content distribution in P2P distribution is difficult. Peer-to-Peer tools are able to contribute in a number of ways to the promotion of the freedom of speech.

According to the report *Enemies of the Internet* [11, p. 5] from the non-governmental organization *Reporter without Borders*, the number of countries that censor information on the Internet has risen rapidly in recent years. Over 60 countries filter the content of the Internet to some degree. Moreover, this is a growing trend. Countries which filter the Internet include not only repressive regimes, but also places such as the United Kingdom and Australia. Peer-to-Peer technology like The Onion Router (TOR) can help citizens and journalists bypass censorship infrastructure to access filtered information.

By far, the most successful P2P protocol is BitTorrent. According to a *Sandvine Report* [Van12] from 2012, BitTorrent is responsible for 36.8 % of all upstream traffic in North America and 31.8 % of upstream traffic in Europe during peak hours. This makes it the third most popular protocol in Europe and the fourth in North America. Nevertheless, I view these statistics skeptically, as they come from Deep Packet Inspection (DPI) companies who are interested in selling hardware. In the scientific community, it is indisputable that BitTorrent is currently one of the most popular protocols. Consequently, BitTorrent is the target of many attacks from anti-P2P companies, which work closely with the music, television and film industries [Med11]. Active measurement based studies on real-world torrents have demonstrated the existence of such attacks [Dhu+08b; DHW11]. A question that arises in a light of such attacks: What are ways to secure a P2P network with-

5 http://collateralmurder.com/en/download.html

out a central entity? Before I discuss this question in depth, I would like to discuss my motivations for completing this research project.

1.1 MOTIVATION

My motivation to work on this research project originated from my fascination on P2P systems. Considering the history of the Internet, I think the P2P paradigm which lacks a central entity is a more *natural* way of communicating over the Internet. Especially the P2P protocol BitTorrent kept my fascination, because it is the first protocol with an incentive mechanism that rewards good behavior and punishes bad behavior. In the next paragraphs, I would like to address the following questions: Why is it worthwhile to investigate the security of a P2P protocol? Additionally, who would benefit from more secure P2P protocols?

The security of P2P protocols remains an open problem. Security is not a state, rather it is a process comparable to a never ending game of cat and mouse. On the one hand, there are adversaries, who are searching for vulnerabilities and developing new attack strategies. On the other hand, there are programmers, administrators and security architects, who are trying to defend against such attacks. Researchers sit on both sides developing new countermeasures and mitigation strategies and investigating attack strategies to prevent future damage. This is especially important for P2P applications, because of their widespread use.

Peer-to-Peer applications are more vulnerable to remote exploits than applications that make use of client-server architecture. Those applications have vulnerabilities much like other software. These vulnerabilities are, however, particularly dangerous in P2P applications because of their 'ideology of openness and sharing' [VLO10, p. 8]. Every participant in a P2P network, known as a *peer*, acts as a server and client simultaneously. Consequently, every peer has to open a network socket and accept connections to participate in a given network. Therefore, a fully participating peer has an exception in its firewall to allow P2P traffic in its Local Area Network (LAN). This makes a peer

more vulnerable to attacks, in particular compared to a client-server model where a client opens a network socket but does not accept incoming connections.

Furthermore, P2P applications are interesting targets for attackers because of their distributed nature. For example, the P2P network Mainline Distributed Hash Table (MLDHT) is one of the largest overlay networks with around 15–27 millions users per day [WK13]. BitTorrent is still responsible for most of the Internet's upstream traffic. If an attacker finds a vulnerability in the network protocol or software, the attacker would have an enormous number of targets to choose from. In the worst case scenario, an attacker finds a vulnerability to redirect traffic from a P2P application to an arbitrary victim. That would result in a Distributed Denial-of-Service (DDoS) attack. In such cases, the entire Internet community would benefit from more secure P2P protocols.

Specifically: all P2P application users would benefit from more secure P2P protocols. In the previous example, users would unknowingly participate in an attack. Moreover, companies using this protocol would benefit from more secure P2P protocols. Lastly, the mistakes that have been made with current P2P protocols should be avoided when designing future P2P protocols. It is a challenging task to secure a P2P application. Approaches drawn from the previous client-server paradigm do not work in a P2P environment. The next section describes the problems encountered in making a P2P application more secure.

1.2 CHALLENGES IN P2P SECURITY

A typical client-server environment fits the following rubric: one or multiple servers provide specific services (e. g. a web page or a video), and clients consume these services. To make this service secure against attacks, the server is crucial. Because, the server is under the full control of the service provider, it is easier to secure services. A service provider can apply the following security strategies to make a server more secure:

- Keep the OS including the services software, up-to-date to protect against the latest threats.
- Install a Stateful Packet Inspection (SPI) or DPI firewall in front of the server to filter malicious traffic.
- Install an Intrusion Detection System (IDS) on the server to monitor suspicious activity and warn the system administrators as soon as possible about potential attacks.
- Patch a software vulnerability on the software which runs on a server.

These countermeasures and mitigation strategies are only successful if there is a central entity or multiple entities that are under the control of the service provider. It remains a challenging task to make P2P networks secure without a central entity. Every participating peer is both a server and client. Such a framework means the formerly employed measures to secure a network service no longer apply to a P2P environment.

One approach to securing P2P systems is to define rules which punish improper behavior and reward proper behavior. Steinmetz and Wehrle [SW05, p.12] describe P2P as a 'paradigm shift from coordination to cooperation, from centralization to decentralization, and from control to incentives'. BitTorrent was the first protocol that includes an incentive mechanism, called a *choking mechanism*, to reward sharing peers who acted properly and those who did not. This thesis shows that BitTorrent's incentive mechanism is not continuously designed. This flaw leaves room for exploitation. This thesis also shows that is important to use a secure transport protocol. Before examining these matters, I state my research questions and hypothesis.

1.3 RESEARCH HYPOTHESIS AND OBJECTIVES

In this section, I would like to state my research hypothesis and the objectives of this thesis. Within 5 years, this thesis aims the following objectives:

- Investigate the impact of bandwidth attacks against application and transport layer of the P2P protocol BitTorrent.
- Understand the aspects of bandwidth attacks related to security in a BitTorrent swarm.
- Propose systematic improvements to the found vulnerabilities.

This is archived by analyzing the protocols and running experiments in real-world environments to contribute to a more secure P2P protocol. The following main hypothesis is made:

HYPOTHESIS 1 *Securing the bandwidth consumption of P2P protocols improves the security of the P2P swarm.*

This hypothesis can be split up into two sub-hypotheses:

HYPOTHESIS 1.1 *Peer-to-Peer protocols that do not have bandwidth security policies pose a security threat to both the swarm and the Internet.*

HYPOTHESIS 1.2 *The P2P protocol BitTorrent does not have sufficient bandwidth security.*

The thesis makes several contributions to the field of P2P security.

1.4 CONTRIBUTION OF THIS THESIS

This thesis investigates bandwidth attacks in a BitTorrent swarm and provides insights into impact of such attacks. Key contributions are detailed in the following sections:

1.4.1 *Vulnerability Analysis of the BitTorrent protocol*

Analyzing the vulnerabilities of the BitTorrent protocol is a challenging task. BitTorrent has a variety of parameters and is highly dependent on the bandwidth capabilities of participating peers. Additionally, BitTorrent integrates a reciprocal mode of sharing—which makes

it more difficult to test. Simulations and mathematical models do not properly reflect the impact of bandwidth attacks in real-world scenarios.

I solved this problem by building a real P2P testbed system and developing a software framework. This framework made it possible to test the hypotheses presented in this thesis. Based on this framework, I analyzed BitTorrent protocol according to different threat models and found substantial vulnerabilities.

1.4.2 *Sybil Attack in a BitTorrent Swarm*

Botnets, a large distributed system of malware-infected hosts, are currently inexpensive and therefore their use is increasing. To better understand the impact of a Sybil attack against a BitTorrent swarm, I created an experiment based on *PlanetLab* to investigate this scenario. The resulting experiments provide a number of insights into the problem. I tested all the seeding algorithms used by BitTorrent clients and came to the conclusion that no algorithm sufficiently addresses the security problems that I found. Consequently, I developed a new seeding algorithm.

1.4.3 *Novel Seeding Algorithm*

In the current study's experiments, I show that an attacker who carries out a Sybil attack with a botnet can drastically decrease the performance of the BitTorrent swarm. To prevent such attacks, I developed a novel seeding algorithm which mitigates bandwidth attacks against seeders. This algorithm implements the incentive mechanism in BitTorrent continuously through all peers.

1.5 THESIS STRUCTURE

The remainder of this thesis is organized as follows. Chapter 2 introduces the P2P paradigm and BitTorrent in particular. I provide an

overview of the associated literature and related work in this field in Chapter 3. For this research, it was necessary to build an experimental setup. I describe the construction of this setup in Chapter 4. I then describe, in Chapter 5, how an attacker can exploit BitTorrent extensions and seeding algorithms to slow download speed for all other peers. In Chapter 6, I show how the congestion control Low Extra Delay Background Transport (LEDBAT) from the P2P transport protocol Micro Transport Protocol (uTP) can be exploited to create congestion on a specific path. Chapter 7 examines ways an attacker can exploit the two-way handshake of uTP to create a DDoS attack. Chapter 8 concludes this thesis with a summary of its achievements and a discussion of open research problems.

2

BACKGROUND ON PEER-TO-PEER AND BITTORRENT

This chapter defines the terminology which will be used throughout this thesis and describes the P2P paradigm and BitTorrent protocol. Before I describe the BitTorrent protocol in depth, I provide a brief overview of a P2P paradigm and how it differs from a client-server paradigm.

2.1 TERMINOLOGY

This section defines the terminology used through-out the thesis. No standardized, terminology currently exists.

ACTIVE PEER SET: The active peer set for a peer is the subset of its peer set that it can send data to.

CHOKED: Peer P is choked by peer Q when Q does not send data to P.

FREE RIDER: A peer which only downloads data and denies uploading to other peers.

INFOHASH: A 20-byte Secure Hash Algorithm 1 (SHA-1) hash that uniquely identifies a torrent.

INTERESTED: A peer has data that another peer wishes to acquire.

LEECHER: A peer is a leecher when it is downloading the content of a torrent and uploading previously acquired content for other peers.

MACHINE: A physical or virtual computer with an Internet Protocol (IP) stack.

NODE: A machine that runs a Distributed Hash Table (DHT) implementation.

PEER ID: A 20-byte SHA-1 hash that uniquely identifies a peer.

PEER SET: Each peer maintains a list of peers it knows about within a swarm. This list is known as peer set.

PEER: A machine that runs a BitTorrent client with a torrent.

PIECE: The equally sized parts called a download is divided into.

SEEDER: A peer is a seeder when it has downloaded the content completely and shares it with other leechers.

SUB-PIECE: The section of a *Piece* is further equally divided into.

SWARM: All peers sharing a torrent.

TORRENT: A file that contains metadata about the swarm and distributed files.

2.2 PEER-TO-PEER PARADIGM

Peer-to-Peer systems have received increased attention in the past years, and are primarily known for illegal file sharing. However, the term denotes a network paradigm.

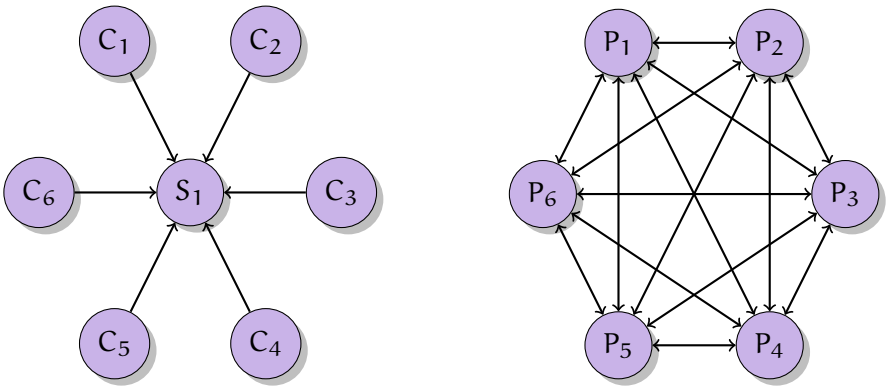
The term *peer* is defined by [Dic13] as follows:

> *peer* (pɪə) 3. a. a person who is an equal in social standing, rank, age, etc (from Old French *per*, from Latin *pār* equal)

According to this definition, *Peer-to-Peer* suggests a conversation or a connection between equals. No one has more rights than, or stands above, others. Essentially, P2P is one of the oldest forms of communication. Our telephone system was designed as a P2P system. Moreover, decentralization was built into the core of the Advanced Research Projects Agency Network (ARPANET), the original Internet. It was designed as a P2P system where all universities were handled

as 'equal players' and not 'in a master/slave or client/server relationship' [MH01, p. 4]. Additionally, two long-established Internet protocols namely Usenet and Domain Name System (DNS), can be seen as predecessors of the P2P paradigm. Consequently, the Internet was built on P2P communication patterns. A graphical distinction between client-server and P2P paradigms is shown in Figure 2.1.

Differentiation between client-server and P2P



(a) Client-Server Paradigm (b) Peer-to-Peer Paradigm

Figure 2.1: Graphical differentiation between client-server and Peer-to-Peer (P2P) paradigm. Circles with a C are clients, S is a server, and P are peers. There is no central entity in (b) compared to (a). Source: own representation.

Figure 2.1 (a) depicts a central server denoted with S_1. This server provides a service that clients $\{C_1, \ldots, C_6\}$ want to acquire. This service could be an HTTP website, a File Transfer Protocol (FTP)-server, or a video stream. When S_1 is offline, clients are no longer able to reach the service. This is called a Single Point of Failure (SPOF). In contrast, Figure 2.1 (b) has no such SPOF. There is also no distinction between a server and a client, because peer is a server and client at the same time.

A technical definition is, however, more difficult to define, as there are protocols and applications that make use of both paradigms. A first attempt to define P2P is in [Sch01]. The study defines P2P as a system where all participants share a part of their hardware resources (e. g. processing power, network link, etc.) and they communicate directly without intermediary entities. Another definition, from

[Rou+04], states that a P2P system has to be self organizing and should have decentralized control.

These definitions are given in the context of a single service and are therefore insufficiently accurate. Request for Comments (RFC) 5694 defines P2P as a system where participants are involved in transactions that are related to a service. These transactions, however, do not directly benefit participants [Cam09]. For instance, in BitTorrent, a peer uploads pieces of a file to other peers. In *SETI@home*[1], volunteers provide idle time from their computers to search for extraterrestrial intelligence. Therefore, P2P is most generally about sharing resources.

These intensive goals require 'a well-designed network infrastructure at the application level' [YuK12, p. 29]. For this reason, designers of a P2P application need to address questions such as how peers find each other to share resources? The answer to this question impacts the structure of the P2P application, and can in part be understood in two extremes as follows. On one side is a complete *unstructured overlay*, where there is no maintenance overhead but there is no guarantee that users find what they searching for. On the other side is a *structured overlay*, which guarantees that users find what they are looking for but at the expense of additional maintenance work. In between the two extremes lay the *hybrid overlays*. The next chapters provide a brief overview of these typologies.

2.2.1 *Unstructured Overlays*

If the structure of a P2P network is generated at random, it is classified as an *unstructured overlay* [BYL09, p. 46]. The first P2P applications were simple and used unstructured overlays to communicate between peers. For instance, the P2P application *Gnutella* used a simple *flooding mechanism* to build an overlay network. The following sections details this algorithm.

1 http://setiathome.ssl.berkeley.edu/

2.2.1.1 *Flooding and Expanding Ring Algorithms*

Let us assume that every peer maintains a list of peers called *neighbors* or a *peer set*. Once a peer is connected and has peers in its peers set, it can start sending queries to the network. A *query* may be a search request for a file or a ping message to find other peers. The flooding algorithm works as follows. A peer sends a request to all of its neighbors. The neighbors who receive the request forward it to all of their own neighbors except to the peer from which the query came. This continues until a peer drops this query. An example can be seen in Figure 2.2.

Example of an Unstructured Overlay based on Flooding Algorithm

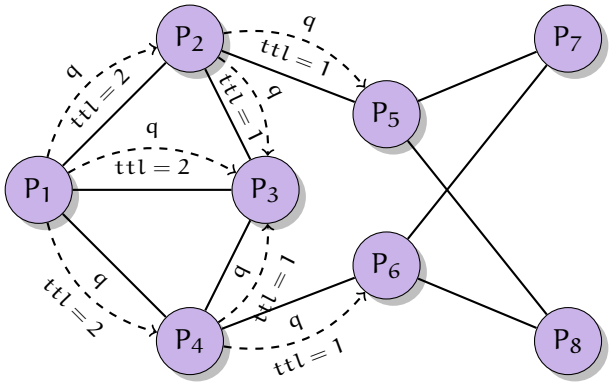


Figure 2.2: Unstructured overlay that routes all queries according to the flooding algorithm. Every query q contains a Time To Live (TTL) value to limit its lifespan. In this example, Peer P_1 sends a query q with a TTL value of 2 to all of its neighbors. Source: own representation based on [BYL09, p. 47].

The peer P_1 in Figure 2.2 sends a query q to all of its neighbors. To avoid endless forwarding of queries, every query contains a Time To Live (TTL) value. If a peer receives a query it decreases the TTL value by one and forwards it to all of its neighbors. For instance, P_2 receives query q from peer P_1 and reduces the TTL value from two to one and forwards it to peer P_3 and P_5. When a TTL value reaches the value zero, peers stop forwarding the query. The P2P application *Gnutella*, which makes use of the flooding algorithm, sets the TTL value to seven [KM02].

There are, however, some disadvantages to this approach. First, the query will not reach every peer [MS07, p. 62]. This can be seen in Fig-

ure 2.2, in which peer P_7 and P_8 do not receive the query, because the TTL is too low. Second, as the name suggests, the network is flooded with queries. This is inefficient because it reduces the speed of downloads. The Internet connection of a peer in Gnutella, for example, is quickly congested because of the large number of queries it needs to forward [Clio0; BA05]. Peer P_3 in Figure 2.2, for example, receives the query q three times, from P_1, P_2, and P_3.

To overcome this problem, a successive variation called an *expanding ring* exists [BYL09, p. 48]. A peer begins with a query that is flagged with a low TTL value. If this query is successful, the process stops. If this query is unsuccessful, the peer increases the TTL value and sends it to all of its neighbors. This variation is especially helpful if a query is successful quickly. Another approach to avoid flooding a network is the *random walk* algorithm.

2.2.1.2 *Random Walk Algorithm*

One alternative to the flooding algorithm is the *random walk* algorithm. In random walk, the querying peer selects one peer of its peer set at random and sends a query to this peer only. The selected peer receives the query, again selects a peer at random, and forwards the query to this peer. Figure 2.3 provides an example of the algorithm.

Example of an Unstructured Overlay based on Random Walk Algorithm

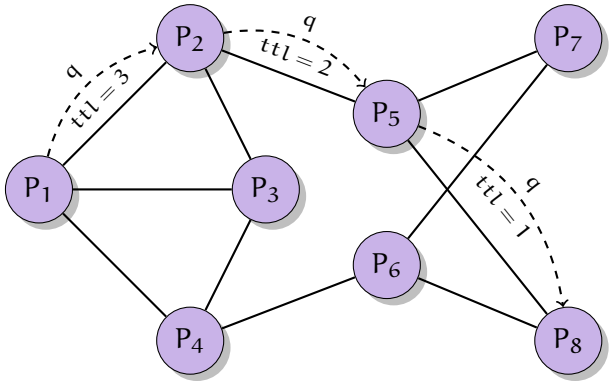


Figure 2.3: Unstructured overlay that routes all queries according to the random walk algorithm. Every query q contains a Time To Live (TTL) value to limit its lifespan. In this example, Peer P_1 sends a query q with a TTL value of 3 to a random peer from its peer set. Source: own representation based on [BYL09, p. 49].

The peer set from P_1 in Figure 2.3 contains $\{P_2, P_3, P_4\}$. Peer P_1 selects one peer at random, in this case P_2, and sends it the query q. As in the flooding algorithm in Section 2.2.1.1, every query contains a TTL value that is decreased with every transmission to limit its lifetime. P_2 again selects a peer from its peer set at random and sends it the query q. The querying peer can send multiple queries at once to improve response time. On the one hand, this approach does not flood the network. On the other hand, it may take some time to find a specific object in the P2P network. Additionally, it largely depends on the correct value of TTL. Early P2P application used an unstructured topology to build an overlay network.

2.2.1.3 *Early P2P applications*

Two of the first P2P applications are *Napster* and *Gnutella*.

NAPSTER Shawn Fanning published a beta version of *Napster* in June 1999, in a few short months, it was the most downloaded program that year [MS07, p. 55]. Napster is considered to be the first P2P application, although strictly speaking, it is not based on a P2P network. It was nevertheless an inspiration for other applications to build a real P2P network with the same functionality. The purpose of Napster was to provide a file sharing platform through which users could download files from other users unknown to them. Figure 2.4 shows an example of a file transfer.

When a client joins the network, it first tells the server S_1, as shown in Figure 2.4, what files the client provides. The Napster server S_1 adds these files to a database. Subsequently, the client can send search queries. For example, client C_2 sends a search query to S_1. As a result, server S_1 replies with a list of peers that provide this file. Then, client C_2 is able to contact other peers to download the file directly. Because server S_1 is still the central node in this application, it is built on the client-server paradigm rather than on P2P paradigm. Gnutella was inspired by Napster and is in contrast to Napster, a true P2P application.

Example of a basic File Transfer with Napster

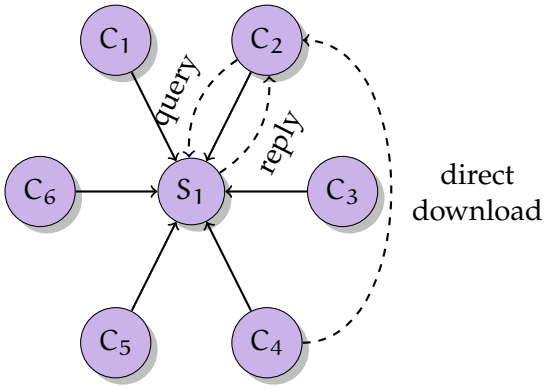


Figure 2.4: Circles with a C are clients and S is a server. The client C_2 sends a search query q to the server S_1 and receives in return a list of clients that provides the searched file. Then, client C_2 can download the file directly from client C_4. Source: own representation based on [MS07, p. 56].

GNUTELLA Justin Frankel and Tom Pepper developed the first version of Gnutella in March 2000 [Kan01]. In contrast to Napster it is a real P2P application without a central entity. Gnutella is built on the flooding algorithm described in Section 2.2.1.1. Gnutella consists of five message types: PING, PONG, QUERY, QUERYHIT, and PUSH [KM02]. A peer uses the PING message to find other peers in the network. If a peer receives a PING message, it replies with a PONG message. The QUERY message contains a search string to find files in Gnutella. If a peer receives a QUERY message and holds the files that have been requested, it replies with a QUERYHIT message. If a peer which holds the file is behind a firewall, the requesting peer sends a PUSH message along the original chain to ask the peer to push the file.

In addition to the disadvantages described in Section 2.2.1.1, Gnutella has an additional disadvantage. Adar and Huberman found that nearly 70 % of participating peers were not sharing any files with other peers [AH00]. They call this phenomenon *free-riding*. This problem undermines the basic notion of resource sharing in P2P networks. Furthermore, it demonstrates that participating peers do not share resources voluntarily and that an *incentive mechanism* is necessary. The P2P application BitTorrent provides such mechanism. To overcome the

limits of unstructured overlays researchers have begun to work on a new category of P2P applications called *structured overlays*.

2.2.2 *Structured Overlays*

The limitations of unstructured overlays, such as their inefficiency and the inability to find rare objects, necessitates more efficient and deterministic topologies, including *structured overlays*. According to [BYL09, p. 75] a structured overlay combines a 'specific geometrical structure with appropriate routing and maintenance mechanisms'. In the following subsections, I provide a brief overview of the most important structured overlays.

2.2.2.1 *Kademlia*

Maymounkov and Mazières have proposed *Kademlia*, a 'peer-to-peer <*key*, *value*> storage and lookup system' based on an Exclusive-or (XOR) metric topology [MM02]. The essential idea is that a node should have more accurate information about closer nodes (neighbors) than nodes that are further away. Additionally, when keys and values from the hash table belong to same key space, nodes that are close to a key are responsible for storing the values for that. According to these rules, a node which is looking for values to a specific key, needs to ask close nodes about the values. If a node does not have these values, it at least provides closer nodes which may have more information.

Initially, each participating node generates a N-bit random ID, a unique identifier for each peer. Kademlia sets $N = 160$. To know which nodes are close, it is necessary to have a distance metric. Kademlia uses an XOR metric as a distance function, seen in Equation (2.1).

$$d(x, y) = x \oplus y, \tag{2.1}$$

where: x = N-bit node ID from a participating node;

y = another N-bit node ID.

The XOR metric is a non-euclidean distance metric. This means that a node from Germany and a node from the United Kingdom can be considered closest because they choose a similar node ID even when Round-trip Time (RTT) is high. Additionally, the XOR metric has interesting properties. The closest node to x is x itself, which means $d(x,x) = 0$. Exclusive-or is symmetric, meaning that $\forall\, y, x : d(x,y) = d(y,x)$. This guarantees that close nodes save similar details about the address space. This metric also provides the triangle inequality $d(x,z) \leqslant d(y,z) + d(x,y)$.

Each node maintains a routing table that contains N *buckets*, each bucket contains k nodes. The routing table organizes these nodes in a binary tree. Figure 2.5 shows a simplified routing table where $N = 3$.

Example of a Routing Table in Kademlia

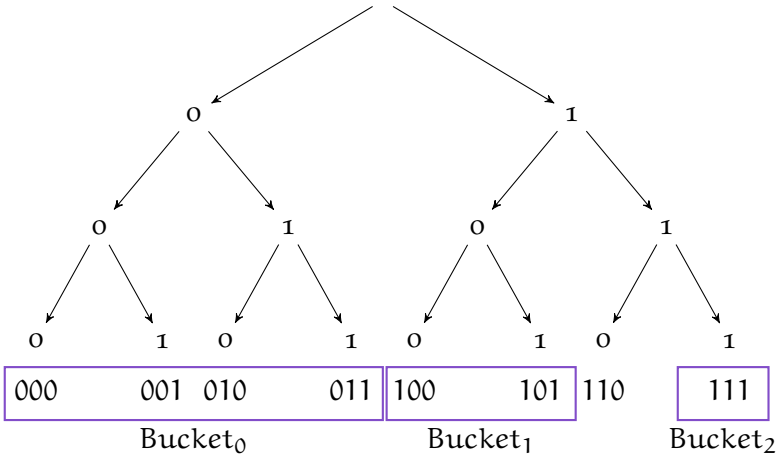


Figure 2.5: Routing table from node 110 for a Distributed Hash Table network with an address space of 2^3. Source: own representation based on [Use14].

The node in Figure 2.5 has the node ID 110. Every new node that it discovers is added to this routing table. The closer nodes are to themselves, the more buckets a node contains. A node decides in which buckets another node will be saved according to the most common bit-prefix, the number of sequentially shared bits beginning from Most Significant Bit (MSB). Because node 001 and node 110 share 0 bits together, node 001 is saved in Bucket_0. Node 111 shares 2 bits with node 110, therefore it is added to Bucket_2. Two popular Bit-

Torrent implementations, *Mainline* and *Vuze*, make use of a modified Kademlia implementation to avoid a central tracker.

2.3 BITTORRENT OVERVIEW

BitTorrent was developed by Bram Cohen in 2001 and was designed for the community of etree[2] [Peto5]. In this section, I describe a download of a single file with the original BitTorrent specification, defined in BitTorrent Enhancement Proposals (BEP) 3 [Coho8b].

The main tenant of BitTorrent is to split a large file into many fixed size *pieces* and these pieces into *sub-pieces*. BitTorrent clients that implement versions of the protocol specifications prior to 3.2 use 1 MiB as fixed-size while newer versions use 256 KiB. Sub-pieces typically have a size of 16 KiB. This makes it easier to transfer a large file among others because different pieces can be shared with different peers. BitTorrent also makes use of *pipelining*. This means that it typically has several requests pending simultaneously. This ensures a steady download rate. The initial client calculates a SHA-1 hash for each piece seen in Figure 2.6. This information is saved together with a Uniform Resource Locator (URL) of *trackers* in a meta-info file called *.TORRENT. This file contains the information necessary to download a file. This file is typically uploaded to a *torrent discovery website* where, other people can download the file via HTTP/Hypertext Transfer Protocol Secure (HTTPS).

The example in Figure 2.7 contains 4 peers. The group of peers who are sharing a file is called a *swarm* and are denoted as: $N = \{P_1, \ldots, P_n\}$, where all participating peers are denoted as P. The complete file F is divided into 4 pieces which are denoted as $F = \{p_1, \ldots, p_n\}$, where each piece is denoted as p. To ease the explanation, I have not divided the pieces into sub-pieces. I denote the set of downloaded pieces from P_1 as $F(P_1) = \{p_1, p_3\}$ and the set of missing pieces as $F'(P_1) = \{p_2, p_4\}$. The peer P_2 has all the pieces, meaning $F'(P_2) = \{\}$ and as such is called a *seeder*. Let us assume that P_2 is the

2 etree is a community of music lovers which shares live recordings from concerts (bootlegs): http://www.etree.org/

Example of the File Splitting Feature of BitTorrent

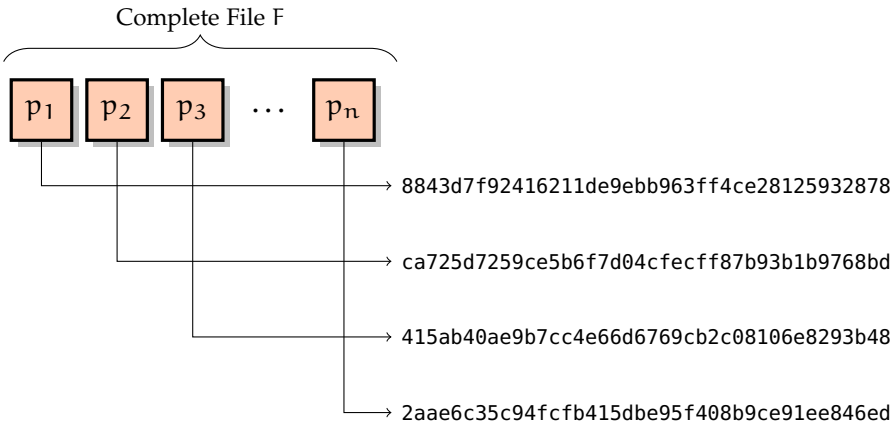


Figure 2.6: A file F which is split into pieces with its Secure Hash Algorithm 1 (SHA-1) hashes. Source: own representation.

initial seeder and uploaded the *.TORRENT file to a popular *BitTorrent portal*. All other peers are called *leechers*. The *.TORRENT file also contains a URL of a *tracker*, a central software which maintains which peers are involved in the swarm. A number of extensions to avoid a central software, e. g. DHT [Loeo8], Peer Exchange (PEX) [Useo8] and Local Peer Discovery (LPD) [Noro9], exist. Assume that peers P_1, P_3, and P_4 have downloaded the *.TORRENT file and want to download its content. The first step of these peers is to contact the *tracker*.

The *tracker* returns a list of random peers (IP address and port numbers) to the requested peer. This list is known as the *initial peer set*. The requested peer iterates through this list and attempts to connect to these peers. Once peer P_2 has uploaded the FILE.TORRENT file to a *BitTorrent portal*, it must be the *initial seed*. This means it must join the swarm and participate as a *seeder* for some time. A torrent is called *alive* if at least one copy of every piece in the swarm exists. It follows, that the swarm in shown in Figure 2.7 is alive. However, if the seeder P_2 goes offline the swarm would no longer be alive, because piece p_2 would be missing.

A BitTorrent client is automatically a *seeder* when a client recognizes that a file is complete. Let us assume that peer P_3 is a new peer and wants to join the swarm. P_3 first contacts the *tracker* and receives the *initial peer set* which includes P_2 and P_4. Peer P_3 connects P_2 and P_4 and begins a session with the BitTorrent handshake. The handshake

Simplified Example of a BitTorrent Swarm

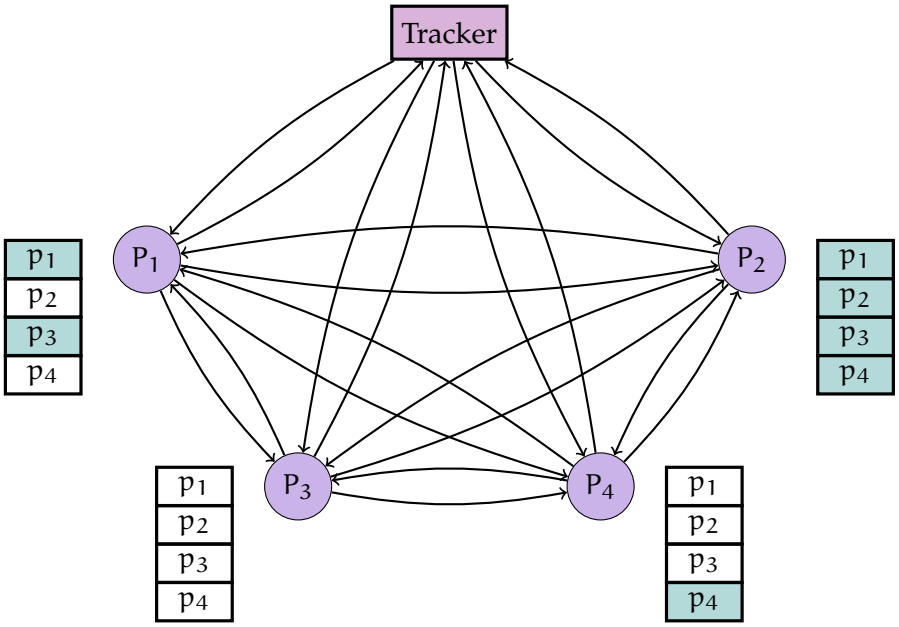


Figure 2.7: Peer P_1–P_4 represent participating peers and p_1–p_4 represent a piece of the file. Pieces with a blue background color are pieces which actual peers have and pieces with the a white background are missing pieces. Source: own representation based on [Cho+12].

also contains information as to which extension the client supports. The next step is peer selection and piece exchange which are completed in BitTorrent by the *choking algorithm* and the *rarest piece first* algorithm. These algorithms are detailed in Section 2.3.1 and 2.3.2.

2.3.1 *Choking Algorithm*

The heart of BitTorrent is the choking algorithm. This decides which peers may download and which may not. Choking is done for the following reasons. First, it prevents free-riders. Second, it ensures a consistent download rate. Third, the Transmission Control Protocol (TCP) congestion control behaves poorly when sent across many connections simultaneously [Coho8b].

The algorithm works in a reciprocal fashion and favors peers who upload. There are two states at either end: uninterested/interested and unchoked/choked. A peer is interested when another side has data which the peer would like to acquire. For example, in Figure 2.7,

the set of interested pieces for P_1 is $F'(P_1)$. On the other hand, a peer is uninterested when the opposite peer has no *interesting* data. Choking means that a peer does not send data until unchoking occurs. Data will only be sent when one side is interested and the other side is unchoked.

The choking algorithm behaves differently in leecher and seeder states. I begin by discussing the leecher state. By default, every peer has 4 unchoked slots and decides, according to the following policy, which peer is unchoked.

1. Every 10 seconds, all peers are ordered by their download rate. The three highest peers are unchoked.

2. Every 30 seconds, one peer is randomly chosen to be unchoked. This is called *optimistic unchoking*.

Optimistic unchoking is completed for the following reasons. A new peer, for example, P_3 in Figure 2.7, joins a swarm but has nothing to share. If reciprocity is taken literally then P_3 would be unable to receive any pieces as it has nothing to share. However, optimistic unchoking prevents this problem. Peer P_3 must wait until P_1, P_2, or P_3 optimistically unchokes it and allows it to download pieces, even when P_3 has nothing to share. This bootstrapping process takes several minutes to deploy. In Section 5.2.1, I describe an extension of the BitTorrent protocol which increases the speed of this process. Furthermore, optimistic unchoking provides peers with the opportunity to interact with new peers. This has advantages and produces a change in that a new peer has a higher download capacity. If the peer in the optimistic slot has a better download rate than a peer which allocates a regular unchoked slot, then this peer is replaced with the optimistic peer.

According to the original specification [Coho8b], the choking mechanism behaves slightly differently for a seeder. The seeder compares the upload rate, instead of the download rate, of the peers. However, this behavior has changed without announcement from BitTorrent Inc. Since mainline client version 4.0.0, the seeder compares time a peer

was last unchoked [LUM06]. The authors of [LUM06] provide experimental evidence that this change improves the fairness of BitTorrent compared with the original specifications. The next section highlights the piece selection algorithm of BitTorrent.

2.3.2 *Rarest Piece First Algorithm*

A good piece selection algorithm is crucial for the performance of a P2P application. This is because the underlying problem in distributing pieces randomly is the *Coupon Collector' problem*. Let us imagine the following situation. A file is divided into m pieces, distributed randomly, and each peer only receives one piece. To have a constant probability that each piece exists at least once, there must be $\Omega(m \ln m)$ peers in the swarm [MS07, p. 234]. To improve this situation, BitTorrent uses an algorithm that includes four policies that are gathered together under the name *rarest piece first*.

The underlying assumption of these policies is that a peer is aware of pieces from its neighbors. If a peer has downloaded a piece successfully and checked its SHA-1 hash, it will send a HAVE message to all peers which are in its peer set. All other peers do the same. Consequently, a peer knows which pieces the peers in the activate peer list have and can calculate the availability of each piece. It is worthwhile to note that no peer has a global view of the complete swarm. Every peer has only an approximation from among their neighbors. In the following section, I describe the four policies according to the original paper [Coh03].

RANDOM FIRST: If a peer has less than four pieces $|F(P_i)| < 4$, then the peer follows the *random first policy*. The peer chooses a random piece and requests it. The idea behind this policy is to get pieces as quickly as possible at the beginning. After downloading four random pieces, the peer switches to the *rarest piece first policy*.

RAREST PIECE FIRST: Because a peer knows which pieces the neighborhood has, it can calculate the availability of each piece. As

the name suggests, in the rarest piece first policy, the peer downloads the pieces which have the lowest availability first. In the example from Figure 2.7, peer P_1 would first request piece p_2 from $F'(P_1)$, because only P_2 has a copy of it. The other missing piece p_4, has a higher availability because two copies of it exist.

This technique has two advantages. First, it reduces the probability that a rare piece becomes unavailable if a peer goes offline. Second, it ensures that a peer receives *interesting* pieces which other peers want to acquire. Each new HAVE message that arrives, updates the availability of a piece.

STRICT PRIORITY: As discussed in Section 2.3, a piece is divided into sub-pieces. If a peer requests a sub-piece, this policy dictates that the following requests belong to the same piece. This helps to finish a piece before other pieces are requested and to complete pieces as soon as possible.

END GAME MODE: A peer follows this policy once it has requested all pieces. To avoid waiting for a piece from a slow peer, the peer sends requests for the remaining piece to all peers in its peer set which hold a copy. This could otherwise delay a peer from finishing the last piece. When it has received the requested piece, it sends a CANCEL message to the other peers.

Before December 2008, all pieces and requests were transmitted over TCP. BitTorrent Inc. announced at the time, however, that *uTorrent*, BitTorrent's free client, will replace its default transport protocol TCP with a novel User Datagram Protocol (UDP)-based protocol called uTP [Haz08].

2.3.3 *Micro Transport Protocol (uTP)*

To better understand why BitTorrent has switched to uTP, it is important to understand the problems with TCP. Transmission Control Protocol makes use of the congestion avoidance algorithm Additive-Increase/Multiplicative-Decrease (AIMD) which combines linear growth

(additive increase) and exponential reduction (multiplicative decrease). Multiple TCP flows that use AIMD eventually converge to use an equal amount of bandwidth [CJ89]. Because BitTorrent is a P2P application, it has multiple TCP connections in its peer sets. Consequently, BitTorrent receives more bandwidth than applications that use a single TCP connection. As a result, BitTorrent affects foreground traffic like web browsing and email.

Previously, to solve this problem, nearly every BitTorrent client had the option to set a fixed bandwidth limit, meaning BitTorrent use a fixed amount of bandwidth. Let us suppose that μ is the bandwidth capability of an Internet connection, t is the bandwidth consumption of concurrent TCP flows, and b is the static bandwidth limit of the BitTorrent client. If

(1) $b > (\mu - t)$, then BitTorrent uses too much bandwidth and interrupts the other TCP connections;

(2) $b < (\mu - t)$, then BitTorrent uses too little bandwidth and there will be $\mu - t - b$ unused bandwidth;

(3) $b = (\mu - t)$, the ideal amount of is used. But t is not constant and will soon become (1) or (2).

This is an inflexible solution. Additionally, it requires knowledge about bandwidth capabilities which may not be available to inexperienced users. To solve these problems, BitTorrent has developed a new transport protocol based on UDP with a new congestion control, LEDBAT. This algorithm detects unused head room with the one-way delay measurement and automatically adjusts the bandwidth limit b. An ideal traffic flow is depicted in Figure 2.8.

If the foreground traffic—in Figure 2.8 (blue)—increases, the background traffic (red) should automatically throttle back. Conversely, if the foreground traffic decreases, the background traffic should take the unused head room (remaining bandwidth). This novel transport protocol is specified in BEP 29 [Nor10]. The uTP header of every packet is shown in Figure 2.9.

Micro Transport Protocol has much in common with TCP. The protocol controls the connection flow with a sliding window and verifies

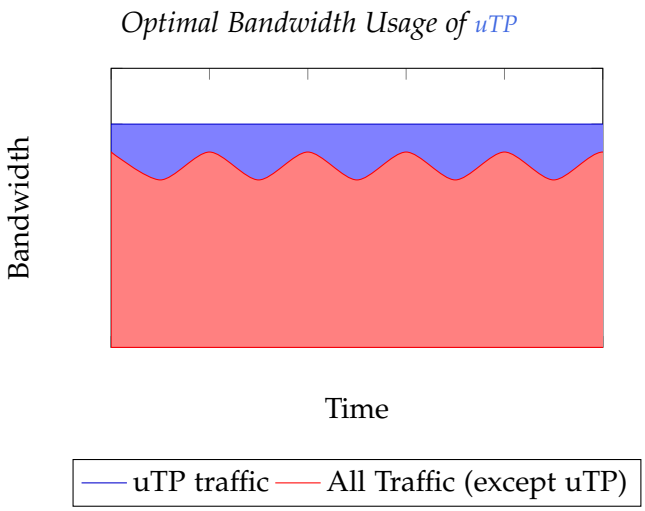


Figure 2.8: Ideal traffic flow with foreground traffic (blue) and Micro Transport Protocol background traffic (red). Source: own representation.

Header of a uTP Packet

0	4	8	16	32
type	version	extensions	connection ID	
timestamp t_{snd} (in microseconds)				
timestamp difference Δt (in microseconds)				
window size				
sequence number			acknowledgment number	

Figure 2.9: Version 1 header of a Micro Transport Protocol (uTP) packet. Source: own representation based on [Nor10].

data integrity with the help of sequence numbers. Sequence numbers, however, refer to packets instead of bytes. Micro Transport Protocol also supports Selective Acknowledgment (SACK) via an extension which is enabled by default in the reference implementation *libutp*[3]. In contrast to TCP, SACK is implemented using a bitmask where each bit represents a packet in the send window. When a bit is set, the receiver received this packet, and vice versa. Micro Transport Protocol initiates a connection with a two-way handshake instead the three-way handshake used by TCP. The message flow of this handshake is shown in Figure 2.10.

The initiator in Figure 2.10 sends a ST_SYN packet (similar to a TCP packet with a set SYN flag) to the receiver. The receiver acknowl-

3 https://github.com/bittorrent/libutp

Two-way Handshake of uTP

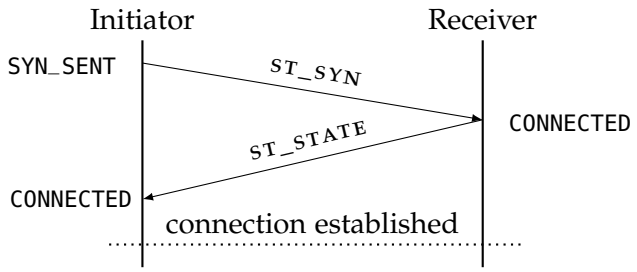


Figure 2.10: Two-way handshake to initiate a connection between two uTP nodes. The text on the outer edges reflects the state of the protocol. Source: own representation based on BEP 29 [Nor10].

edges the ST_SYN packet with a ST_STATE packet (similar to a TCP packet with a set ACK flag). A connection between the two machines is then established. The main difference between TCP and uTP is the novel congestion control algorithm LEDBAT.

2.3.3.1 *Low Extra Delay Background Transport*

The congestion control LEDBAT was defined in RFC 6817 in December 2012 [Sha+12]. The novel congestion algorithm uses a one-way delay measurement as the principal congestion control. This measurement works in part because the sender includes a 32-bit timestamp value in the header field of a data packet. The receiver calculates the one-way delay measurement and includes a 32-bit timestamp difference in the acknowledgment. An example of the network flow of LEDBAT is shown in Figure 2.11.

Example of the One-way Delay Measurement

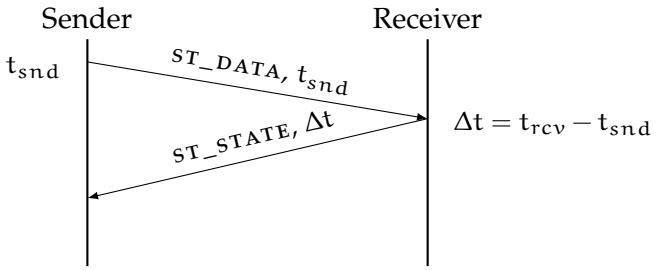


Figure 2.11: One-way delay measurement as a congestion control with two uTP nodes. Source: own representation based on RFC 6817 [Sha+12].

The sender, as shown in Figure 2.11, includes the time, t_{snd} in microseconds, in the data packet upon sending the packet. The receiver can then calculate the one-way delay Δt by subtracting the t_{snd} from the time the data packet is received, t_{rcv}. The receiver returns an acknowledgment that contains Δt in the header field TIMESTAMP DIFFERENCES. The sender saves a history of 100 Δt values in a vector $\vec{h}$. The uTP stack does not interpret the time difference Δt as an absolute value, rather as an relative value compared to previous data points. In LEDBAT, Equation (2.2) is used to determine whether to increase or decrease the send window.

$$\text{off_target} = \begin{cases} + & \text{if } \min(\vec{h}) < 100 \text{ ms} \\ = & \text{if } \min(\vec{h}) = 100 \text{ ms} \\ - & \text{if } \min(\vec{h}) > 100 \text{ ms} \end{cases} \tag{2.2}$$

where: $\vec{h}$ = history of 100 Δt values.

If the lowest value in vector $\vec{h}$ is less than 100 ms, LEDBAT increases the send window, and if it is more than 100 ms, LEDBAT decreases the send window. If the lowest value in vector $\vec{h}$ is 100 ms then LEDBAT does not change the send window.

3

LITERATURE SURVEY AND RELATED WORK

In this chapter, I provide an overview of related work in the field of P2P security with a focus on BitTorrent. The classification of related work is according to the security requirements discussed in Section 1. According to [MS07, p. 199], there are three security requirements which must be considered in every P2P network: service availability, document authentication, and peer anonymity.

3.1 SERVICE AVAILABILITY

The main task of a P2P network is to provide a certain service. In case of BitTorrent, this service is the distribution of files. Attacks that disturb this service are grouped under the term *service availability*. BitTorrent's ecosystem consists of the following components: leecher, seeder, peer discovery, and torrent discovery. Vulnerabilities in each of these components can disturb the availability or efficiency of BitTorrent.

3.1.1 *Sybil and Eclipse Attacks*

The name *Sybil* comes from book, written in 1973 and of the same name, by *Flora Rheta Schreiber*. The book concerns a woman with multiple personality disorder and is based on a true story. In large scale P2P networks, a variation of personality disorder can be an effective attack against service availability. This is a well-known type of attack and was originally introduced by [Dou02]. An attacker injects multiple fake peers into a network which are all under the control of the attacker. Douceur argues that it is nearly impossible to present a distinct identity to an unknown peer without using a central trusted authority. This creates the foundation for a number of attacks.

Douceur also presents a number of defensive tactics in [Dou02]. He notes that storage, communication, and computation resources are limited. His approach takes advantage of this and posits the following ideas. In terms of communication resources, a peer can broadcast a request to another peer and only accepts replies within a given interval of time. In terms of storage resources, a peer can challenge another peer to store large amounts of unique data. And in terms of computational resources, a peer can challenge another peer to solve a unique computational puzzle. An obvious disadvantage of these defenses is the expenditure of resources in the form of time, energy, and computational power. Moreover, this only makes it difficult to create multiple *Sybils* on one machine. These approaches do not prevent an attacker from renting botnet time from which to use the *Sybil* attack. An attack related to a Sybil attack is an *eclipse attack*.

In an *eclipse attack*, an attacker attempts to inject nodes into the victim's routing table which are under the control of the attacker. This attack is also known as *routing table poisoning*. This causes the victim to only communicate with these malicious nodes instead of legitimate nodes [Loc+10; Cas+02, p. 200]. It is named an *eclipse attack* because the victim is *eclipsed* by the attacker. A simple example is demonstrated in Figure 3.1.

Example of an Eclipse Attack

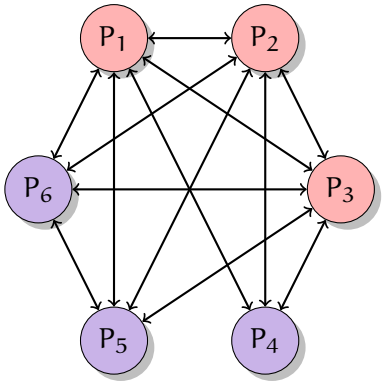


Figure 3.1: Example of an eclipse attack against peer P_4. Red circles are attackers and blue circles are benign peers. Source: own representation.

Figure 3.1 shows a simple P2P network with overall six peers. Peers P_1–P_3 are malicious peers which eclipse the victim Peer P_4. As a con-

sequence, the victim is cut off from the P2P network and the attacker determines the messages the victim receives. This attack is a general problem in overlay networks. This observation is made in [Sin+06]. Overlay networks are the basis of a P2P network. A problem arises in that the membership of overlay networks is generally open or only loosely restricted. An attacker who controls a large number of neighbors from legitimate nodes can disrupt the overlay communication of these nodes. Singh et al. propose in [Sin+06] a countermeasure based on anonymous auditing with moderate churn. BitTorrent makes use of a structured overlay network MLDHT to find new peers.

Wang and Kangasharju demonstrate in [WK12] a real-world Sybil attack combined with MLDHT index poisoning attack. The study differentiates between *vertical* and *horizontal* attacks. A horizontal attack attempts to pollute the routing table from as many peers as possible whereas a vertical attack attempts to place as many Sybils as possible in the routing table of a specific peer. A vertical attack is similar to an eclipse attack. Interestingly, the study found that both types of attacks are already widely used currently.

In a recent paper, [Hei+15] show that the underlying P2P network in the electronic cash system *Bitcoin* [Nak09] is vulnerable to eclipse attacks. This allows attackers to filter victims's view of the *block chain*[1]. This results in the victim wasting computing power when mining blocks. Furthermore, if an attacker has eclipsed a number of nodes, the attacker can launch N-confirmation double-spending attack on a merchant. Typically, a merchant only ships goods if the bitcoin transaction is confirmed. If an attacker sends a transaction to an eclipsed merchant, it confirms the transaction with fake nodes in the merchant's routing table. By doing so, the attacker can receive goods without payment. Sybil and Eclipse attacks are typical attacks against overlay networks. The next attacks detailed targets client-server infrastructure but can be initiated from a P2P network.

1 A database that contains all Bitcoin transactions.

3.1.2 *Distributed Denial of Service Attacks*

The goal of denial-of-service (DoS) attacks is to disrupt a specific service, e. g. an HTTP server, by sending a large amount of traffic to the service. If a DoS attack is initiated by a single machine, it is called a single-source DoS (SDoS) attack. However, it is difficult for a single machine to generate the sufficient number of packets and amount of bandwidth to disrupt a service. However, an attacker can gain control of many machines and from each of these generate requests to the targeted service. Such an attack is a DDoS attack [Mir+05] and is more dangerous than SDoS attacks.

There are two methods to find peers which share the same file, *tracker* and *trackerless*. A peer has to send a GET request to a *tracker* to receive a list of peers in return. The GET request contains the IP address and port number of an active peer. [Sia07] show that both values can be forged to inject the victim's IP address and port into the *tracker's* peer list. The *tracker* can then broadcast this information to other peers, who are trying to connect to the victim. If the swarm is large enough this can be exploited to initiate a DDoS attack. This can be easily fixed by only using the source IP address of a peer.

Another method to exploit BitTorrent's tracker for a DDoS attack is shown in [HKZ07]. This study discovered that BitTorrent peers 'trust the tracker without implementing any authentication or verification procedures' [HKZ07, p. 1]. An attacker can deploy a modified tracker and include victim's IP addresses in the peer list. Legitimate users trust the response from the tracker and try to make a connection to the victim. This does not generate much traffic, however, the victim must hold these connections until a timeout occurs. As a result, the attacker exhaust the connection resources from the victim and legitimate peers cannot connect to the victim.

[EGM] detail another form of a DDoS attack. In this form, the attacker reports the victim as one of the *trackers*. This attack exploits the lack of a BitTorrent handshake between a peer and the *tracker* as there is between peers. This can be exploited by creating a new torrent file, putting the victim's IP address in the torrent, and publishing

this torrent on a popular torrent discovery site. The authors note that this would be ineffective because statistics from the torrent discovery website would show that there is no valid *tracker* available. More effective is the exploitation of the multi-tracker feature, described in BEP 12 [Hofo8b]. This attack uses a modified *tracker* to announce the victim as an additional *tracker*, publishing fake statistics to a torrent discovery website. The authors note that the victim can be any machine on the Internet.

In [STRo7], researchers found vulnerabilities in the membership management of P2P systems which could be exploited to launch a DDoS attack. They investigated the *Kad Network*, a P2P protocol based on a modified version of *Kademlia* (see Section 2.2.2.1). The first vulnerability arises in the search mechanism of Kademlia. An attacker who receives a lookup query will return a list of peers that contains the victim. The peer who receives this list contacts the victim who does not need to be a member of the P2P network. Another vulnerability the researchers found is in the gossip-based video broadcast system *ESM* [Chu+04]. Each member periodically picks a random neighbor to send a subset of neighbors. A malicious peer can send a gossip message to legitimate members that contains the victim as a valid contact. Later, these members send a gossip message to the victim which can result in a DDoS attack.

3.1.3 *Bandwidth and Connection Attacks*

Bandwidth attacks are first mentioned by [Dhu+o8b]. This attack is detailed as follows. A BitTorrent peer maintains n peers in its peer set. From these n peers, a peer chooses u peers for an unchoke slot and o peers for an optimistic unchoke slots. The intention of a bandwidth attack is to occupy most of the u unchoke slots with the effect that another peer which would like to download from the victim does not get an unchoke slot. As a consequence, the attacker steals bandwidth from the victim and reduces the victim's efficiency. An attacker can occupy unchoke slots by exploiting weakness from the choking algorithm. To exploit the chocking algorithm mentioned in

BEP 03 [Coho8b], an attacker needs a higher download speed as other peers. In Section 5.3.1, I describe different choking algorithms and how to exploit these algorithms.

[Dhu+08b] were the first to investigate bandwidth attacks and connection attacks regarding BitTorrent. They define bandwidth attacks as peers who try to allocate a upload slot from the seeder as soon as possible to stop the seeder. They study's measurements show that bandwidth attacks are mostly ineffective and it is only possible to increase download time by up to 10 %.

Another paper by the same authors, [Dhu+08a], investigates a bandwidth and a connection attack against the initial seeder to determine the possibility of stopping the seeder. The idea behind this bandwidth attack was to occupy most of the seeder's unchoke slots by downloading at a faster rate than other peers. The connection attack has aimed to consume the majority of the seed's connection slots in its early stages. These measurements showed that bandwidth attacks are ineffective and that it is only possible to increase the download time by up to 10 % and never by more than a factor of five. However, this research also shows that BitTorrent seeds are vulnerable to connection attacks and that it is possible to prevent a seeder with an Azureus client[2] from distributing files.

3.2 DOCUMENT AUTHENTICATION

Document authentication refers to the authentication of files that are distributed via BitTorrent. According to [Cue+10], 30 % of all torrents include fake content. Fake content includes unintended files such as malware or scam websites. According to [Kry+11], more than 99 % of fake content is malware or scam websites. The question arises, how can a peer ensure that a file is not fake?

[Kry+11] present a countermeasure called *Fake Detector*. Fake content is often detected by users who download the content and then report its lack of validity in the comments of the torrent file. It is then the responsibility of the portal administrator to remove the torrent file

2 Azureus is a BitTorrent client which is now known as *Vuze*.

from the website. Such a solution requires human intervention and a file is only removed from a single portal. However, *Fake Detector* detects fake content after its creation by the IP address of the publisher. This is possible because, as the authors discovered, 90 % of fake content is published by a few users. However, this simple solution does not work if publishers of fake content switch to dynamic IP addresses rather than static IP addresses.

3.2.1 *Content Pollution Attacks*

If a malicious user publishes a large number of decoys (same or similar metadata) to a torrent discovery site, this is called a *content pollution attack*. The consequence of such an attack is that if user queries for specific content, the predominant content pulled is fake. In [San+11], researchers propose a novel countermeasure against content pollution attacks called *Funnel*. *Funnel* counts positive (non-polluted) and negative votes (polluted) and computes content's reputation using *subjective logic*.

3.2.2 *Metadata Pollution Attacks*

Metadata pollution is similar to *content pollution* except that changes are made to the metadata of a torrent, such as the filename, type, and title [BYL09, p.339]. The content may be fake content or an irrelevant file. Such an attack causes users to download incorrect content.

3.2.3 *Index Poisoning*

Index poisoning is an attack that exploits the search capabilities of P2P networks. Nearly all P2P systems have an index which is used to either find content or peers. In an *index poisoning attack*, an attacker introduces a large number of fake entries into the index. BitTorrent itself has no search functionality. However, BitTorrent makes use of DHT which has a search functionality. [LNR06] investigated this at-

tack in structured and unstructured P2P networks. The study found that both networks are highly vulnerable to this type of attack. As a countermeasure, they introduced a distributed blacklisting scheme but noted that more work is necessary. Another security requirement of P2P networks is anonymity.

3.3 ANONYMITY OF THE PEERS

Anonymity helps peers not only avoid prosecution but also prevents censorship or oppressive measures. The BitTorrent protocol itself does not provide mechanisms to hide the identity of peers and researchers found that it is easy to invade the privacy of BitTorrent users.

A study by [Le +10], published a set of simple techniques to undermine the privacy of BitTorrent users. Most *trackers* support a SCRAPE-ALL request to provide detailed statistics on torrents. This request returns infohashes of all torrents for whom the *tracker* is responsible, the number of peers that have downloaded the full content, and the number of currently subscribed peers, among other information. By exploiting this request, an attacker can quickly gain a list of all identifiers from the *tracker*. This is the foundation of spying techniques which exploit the ANNOUNCE request of the *tracker* to acquire a list of IP addresses from peers. The authors demonstrated that with these two simple techniques they were able to determine at least 90 % of the peers distributing each content. They concluded that it is easy to spy on most BitTorrent users and that the protection of IP to content mapping of P2P file sharing users remains an open question.

The same applies to DHT implementations based on Kademlia. There is no privacy feature in both MLDHT and Vuze DHT (VDHT). The latter supports an additional message called REPLICATE-ON-JOIN. With this message it is possible to ask a peer for the infohashes it is currently downloading. The study [WH10] showed that this feature can be exploited for both fun and profit. It is possible to profit from this because the authors were able to monitor nearly eight million IP addresses downloading 1.5 million torrents. This could also be

used for fun because the authors showed that with this feature it is possible to bootstrap a BitTorrent search engine. In the moment, *Vuze* has removed this feature to provide better security [Cha+09].

However, in general techniques exist for P2P applications that provide privacy and anonymity. The following sections examine anonymity in P2P networks.

3.3.1 *Friend-to-Friend Network*

As trust is a pervasive issue on the Internet, private P2P systems called Friend-to-Friend (F2F) networks exist. A peer in a P2P system that makes use of F2F has connections only to communication partners that are trustworthy, such as friends. This means a peer connects to a small number of known peers [CC05]. The IP address of a peer is known only by its direct neighbors. For example, in Figure 3.2, the real identity of the receiver R is known only to peers F_6–F_8. To enable world-wide communication, it is necessary for every peer to have a pseudo address. Suppose the sender S with the pseudo address C3D wants the send a message m to the receiver R with the pseudo address F23. Sender S uses *flooding* (see Section 2.2.1.1) to send the message m to its friends F_1–F_3. These friends then send the message to their friends until the TTL is zero or the message has reached receiver R. The communication channel between friends must be encrypted, otherwise an attacker who monitors the communication channel can read the content of messages which may reveal the identity of the sender or receiver.

An attacker cannot map pseudo address to a real IP address except if the attacker is a friend of the victim. This suggests that an attacker can only reveal the identity of a peer through social deception. Additionally, peers can change pseudo addresses with each login to the network. The majority of P2P networks that provide anonymity make use of F2F networks, e. g. Freenet[Cla+01], GNUnet[3], OneSwarm[4] [Isd+10].

3 GNU's Framework for Secure Peer-to-Peer Networking: https://gnunet.org/
4 https://www.oneswarm.org/

Example of a Friend-to-friend Network

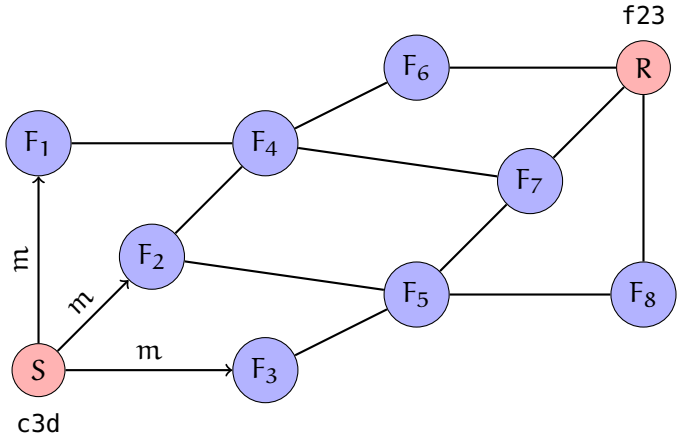


Figure 3.2: Friend-to-friend network where each peer is connected to a small number of friends. Source: own representation.

3.3.2 *Dining Cryptographers*

Dining cryptographers is an easy method to hide an author's identity and was first proposed by [Cha81]. Peer-to-Peer networks based on this method are called Dining Cryptographers (DC) networks. Suppose that one of n peers, where $n > 2$, would like to publish a message m. Each peer i should be able to read this message but should be unaware of its publisher. Figure 3.3 shows an example where n is three. Every *cryptographer* or peer C_i creates a random number x_i and sends this number to its right neighbor C_{i-1}. Each peer can then XOR this random number from its own, $h_i = x_i \oplus x_{i-1}$. Only the peer who would likes to publish the message makes an additional XOR calculation, $h_i = h_i \oplus m$. Then, all peers publish their h_i. The sum of all h_i is equal to the message m. If no peer has sent a message, the result would be 0.

This technique ensures that a passive attacker cannot reveal the author's identity. An active attacker can only learn the h_i of a specific peer. However, an active attacker can anonymously disrupt the service by sending random bits instead of the real h_i. In such case nobody is able to read the message m. An additional problem of DC is the complexity of its computation and communication. This makes it difficult to use in large P2P swarms.

Example of Dining Cryptographers

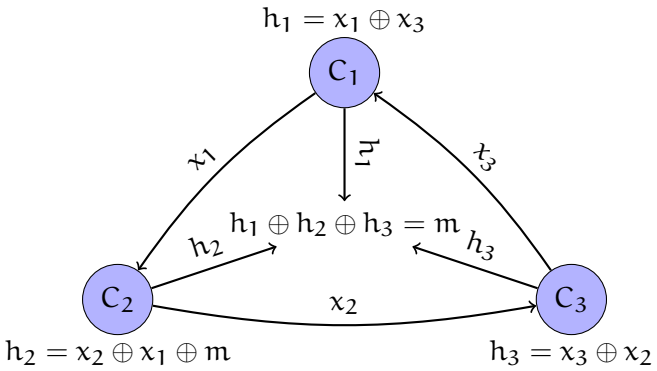


Figure 3.3: With Dining Cryptographers, an author can publish an document or a message m without revealing its identity. Source: own representation based on [MS07, p. 213].

3.3.3 *Onion/Garling Routing*

Goldschlag, Reed, and Syverson in [GRS99] propose *onion routing* to provide low-latency anonymous communication for TCP applications. It references *mix-networks* or *mix-cascades* in [Cha81]. The most prominent project based on onion routing is the TOR project[5]. The project does not see itself as a P2P network though it is a P2P network according to the definition by [MS07, p. 215]. Many years of research were necessary for this project and it includes around 3 million users currently.

If sender S wants to send a message m to receiver R with TOR, sender S first randomly selects n onion router from the network. Sender S needs all the public keys of the selected router. Suppose $n = 3$, then sender S uses the public key from O_n and encrypts the message m, and adds the address from receiver R. This message m is then encrypted with the public key from onion router O_{n-1} and the address of onion router O_n is added. Sender S iterates this process until it reaches the onion router O_1, then sender S sends this layered encrypted message to O_1, as depicted in Figure 3.4.

Each router O_n removes a layer by decrypting the received message. This message contains a readable header with routing instructions and an encrypted body. A router sends the encrypted message

5 https://www.torproject.org/

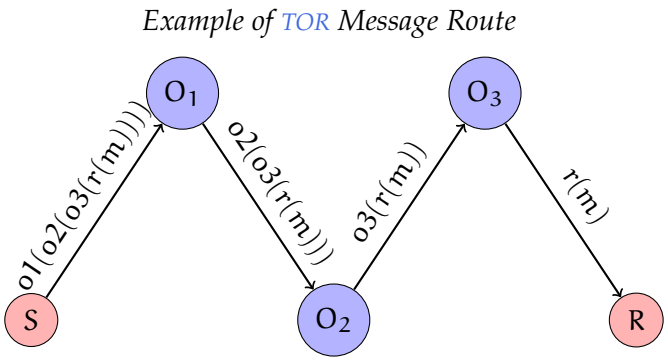


Figure 3.4: Onion Router (O_1–O_3) receives a message and decrypts its content. The content consists of a readable header and an encrypted payload. A router sends the encrypted payload to the next router according to the header. Source: own representation based on [MS07, p. 215].

to the next router, according to the routing instructions. This process iterates until the message reaches its destination. If receiver R wants to answer sender S the process is reverses.

Suppose an attacker monitors the communication between O_1 and O_2. The attacker can only see an encrypted message not the original sender, the destination, nor its payload. There are two types of onion routers, also known as *relays*: middle relays and exit relays. Middle relays receive encrypted traffic and forward it to another relay, like O_2 in Figure 3.4. If an attacker introduces itself as a middle relay, the attacker can only see routers in-between. An exit relay is the final router before the message reaches its destination, like O_3 in Figure 3.4. Attacker presenting themselves as exit relays are able to read the message but cannot figure out who was the original sender.

Garlic routing is an extension of onion routing and was first coined by Michael J. Freedman in Roger Dingledine's *Free Haven* master's thesis. The Invisible Internet Project (I2P)[6] is an anonymous P2P network that makes use of garlic routing. It supports a feature which joins several messages with independent routing instructions on each level into a new onion. This reduces the load between routers and makes traffic analysis difficult. It also differs from onion routing in that the path is unidirectional. This is advantageous because the path for the return route must be chosen anew, improving anonymity.

6 https://geti2p.net/en/

However, Niedermayer notes that bidirectional connections are often used to establish a symmetric session key [Nie10, p. 126].

One P2P file sharing program that makes use of this onion-like routing is *OneSwarm* [Isd+10]. It is used by hundreds of thousands of users and combines onion routing-like privacy with BitTorrent-like performance. In *OneSwarm*, users have control over the amount of trust they place in other peers by making use of F2F networks. However, Prusty, Levine, and Liberatore in [PLL11] show that *OneSwarm* is vulnerable to a novel timing attack. This attack requires two attackers and can distinguish whether a peer is the source of queried content.

In terms of privacy, it may seem that TOR can be used for protection with BitTorrent. However, Manils et al. in [Man+10] demonstrate three attacks that reveal the real identity of BitTorrent users, even when they are using TOR to protect their privacy. The first attack presented is a simple inspection of BitTorrent control messages, namely ANNOUNCE message to the *tracker* and the *extension protocol handshake* [NSH08] which both may contain the real IP address of BitTorrent users. Unlike this attack, the second attack guarantees the revelation of IP address. This is a typical man-in-the-middle (MITM) attack, where an attacker adds itself to the list of peers, that the *tracker* returns to a peer. The last attacks affects DHT. Because TOR does not support UDP and DHT makes use of UDP, the client is unable to send a DHT message over the TOR interface, using the public interface instead. An attacker could subsequently use DHT to find a peer with the same port number.

Part II

ANALYSIS OF THE ATTACKS

"The more sophisticated the technology,
the more vulnerable it is to primitive attack.
People often overlook the obvious."

— *Doctor Who [JW12]*

4

METHODOLOGY

This chapter describes the current study's methodology which I use to confirm and refute hypotheses concerning the security of BitTorrent. The next sections provide information about the testbed systems which were used.

4.1 LOCAL P2P TESTBED SYSTEM

An initial idea was to create a complete virtual testbed system on a high-performance server. After creating such a testbed system, the hard disc was not efficient enough to allow for more than 10 peers to read and write simultaneously without a large delay. Thus, it was necessary to build a distributed testbed with real hardware. The new testbed system consist of 32 machines, one controller, one monitor machine, and a switch. A network diagram of the testbed system is shown in Figure 4.1.

Network Diagram of the Testbed System

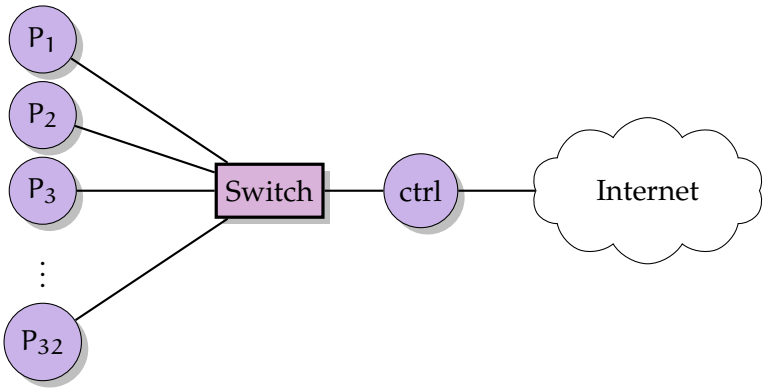


Figure 4.1: A network diagram of the testbed system. The testbed system consists of peer $P_1 - P_{32}$ and a controller (ctrl). Source: own representation.

All machines are connected to a network switch. The controller machine is the only machine with access to the Internet. In the next two

sections, I describe and analyze the hardware and software components of this testbed system.

4.1.1 *Hardware*

The testbed system consist of 33 computers and one switch. All computers are Lenovo IdeaCentre Q180 Mini-PC with the hardware specification listed in Table 4.1. The principal requirement for the computers is hardware that is comparable to a normal desktop computer. Second, the computers must be cheap and stackable. Third, there must be at least a Gigabit Ethernet interface.

HARDWARE	SPECIFICATION
Model:	Lenovo IdeaCentre Q180-VC71-HGE
CPU:	Intel® Atom™ D2700 (2 × 2.13 GHz, 1 MB cache)
RAM:	2048 MB, PC3-10600 1333MHz DDR3
Hard disk:	320 GB
Ethernet:	Gigabit Ethernet on planar, Realtek RTL8111E-V
Wireless:	11 b/g/n Wireless 4, Realtek RTL8188CE

Table 4.1: Hardware specification of every machine in the testbed system. Source: [Len11].

Moreover, the switch must have at least 33 ports and support Gigabit Ethernet. Another requirement is that the switch is able to generate statistics about network traffic such as Remote Network Monitoring (RMON) and Simple Network Management Protocol (SNMP). Figure 4.2 shows a photograph of the testbed system. The Ethernet cable connectors have different color to identify categories with ease. Blue connectors are peers 1–8, red connectors 9–16, yellow connectors 17–24, and green connectors 25–32. The next section explains the OS and software run on these peers.

4.1.2 *Software*

This section examines both system-software and the distributed experimentation framework.

Photograph of the Testbed System

Figure 4.2: Photograph of the testbed with 32 peers, 1 controller peer, and a separate switch. Source: own representation.

4.1.2.1 *System-Software*

It is important that every machine has the same OS with the same software and the same configuration. Otherwise it gets hard to automatize the experimental process. This means to waste a lot of time to setup the experimental environment, instead of thinking about the experiment itself and the results. A automatized experimental process is also the basis of reproducible results. Therefore, I installed Ubuntu 12.04.1 Long Term Support (LTS) and configured one master machine and cloned this installation to all other machines. For the cloning process I used *Clonezilla*[1] which makes use of Trivial File Transfer Protocol (TFTP), Network File System (NFS), and Preboot Execution Environment (PXE).

I installed a Secure Shell (SSH) server on the master machine to control every machine remotely. The /HOME directory was mounted via NFS from the controller peer. This facilitated the distribution of files to all machines. Additionally, I configured the Network Infor-

1 http://clonezilla.org/

mation System (NIS) to have a central user and group administration. This ensured that all machines had the same user accounts. To add new users, it was only necessary to create accounts on the controller machine. I also installed software and libraries in advance (e. g. interpreter, compiler, debugger, etc.) to save time in the future. I also wrote an experimentation framework to automate the process of a distributed experiment.

4.1.2.2 *Distributed Experimention Framework*

Peer-to-Peer networks by nature necessitate several machines for an experiment. A BitTorrent security experiment can be used as an example. Such an experiment requires at least one tracker, one seeder, and several leechers and attackers. In this context, attackers means to attack the swarm. To coordinate such an experiment, one controller machine and several executive machines are needed. The controller machine delegates tasks to different executive machines, monitors the progress of each machine, and collects data after the experiment. The executive machine receives functions and evaluates them, returning the result to the controller. Additionally, some experiments require additional aspects, including background traffic or logging. These aspects should be easy to activate if necessary. These factors fostered the development of a distributed experiment framework, called *Thayeria*[2]. *Thayeria* is written as a module for the programming language Perl 5[3].

Thayeria is inspired by the distribution capabilities of the programming language *Erlang*[4]. *Erlang* makes it easy to call a function on multiple remote machines even when the module is not present on the machines. Therefore, I developed the following design criteria for a distributed experimentation framework:

- Distributed;
- Peers grouping;

2 The name Thayeria is derived from a swarm fish species.
3 http://www.perl.org/
4 http://www.erlang.org/

- Easy to begin on multiple machines;
- Serialized modules or functions if not present;
- Blocking and non-blocking function calls;
- Extensible via plug-ins.

All criteria are included in the current version of *Thayeria*. Figure 4.3 depicts the architecture of this framework. The experiment script runs on the controller machine C_1 and uses the *Thayeria* client to connect to executive machines $S_1 \ldots S_n$ which make use of the *Thayeria* server. The criteria *distributed* means that a *Thayeria* machine always starts a server and a client at the same time. This makes it easy to replace one machine with another.

Diagram of the Architecture of Thayeria

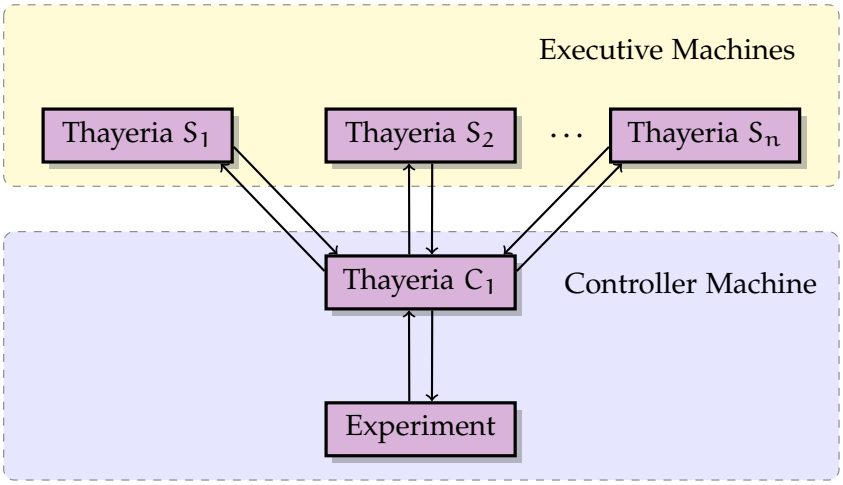


Figure 4.3: A diagram of the architecture of Thayeria. Source: own representation.

Thayeria is easy to initiate. If *Thayeria* is installed, the following shell one-liner is necessary to begin it:

Listing 4.1: Shell one-liner to start a Thayeria instance.

```
1 user@peer$ perl -MThayeria -e 'Thayeria->new()->run'
```

This one-liner can be executed on multiple machines with the program *parallel-ssh*[5]. If a controller machine calls a function on an executive machine, it automatically serializes the module and sends it

5 https://code.google.com/p/parallel-ssh

to the executive machine. This means, the executive machine does not need to have the module installed. Currently, this functionality is limited to modules without dependencies. The next requirement was that it would be possible to call a function in synchronized and asynchronized ways. The first mode is necessary for immediate results and will block the program until the results has arrive. Asynchronized function calls can be used when the program does not need the results immediately and will not block the program. An example for the source code of a controller peer can be found in Listing 4.2. Lines 1–2 import the Thayeria module in the current script. Lines 4–5 create a new Thayeria object. Lines 12–19 show a synchronous function call and lines 24-33 show an asynchronous function call with a callback specified. The event loop is entered in line 36.

Listing 4.2: Example of the source code of a controller peer with Thayeria.

```
1  use Thayeria;
2  use Thayeria::RPC;
3
4  my $machine = Thayeria->new()
5      or croak "Couldn't create a Thayeria machine: $ERRNO";
6
7  $machine->connect_to('deepthought')
8      or croak "Couldn't connect to deepthought: $ERRNO";
9
10 print "Successfully connected to deepthought\n";
11
12 ## synchronous call
13 my $ret = RPC::call({
14     machine  => $machine->machines->{'deepthought'},
15     module   => 'TestPackage',
16     function => 'meaning_of_life',
17     args     => ['str', 23, {name => 'test'}],
18 });
19
20 print "The meaning of life is: = $ret\n";    ## 42
21
22 $machine->connect_to('hactar')
23     or croak "Couldn't connect to hactar: $ERRNO";
24
25 ## asynchronous call
26 my $ret = RPC::call({
27     machine  => $machine->machines->{'hactar'},
28     module   => 'Ultimate::Weapon',
29     function => 'build_bomb',
30     cb       => sub {
31         my ($event_handler, $rpc_href) = @_;
32
```

```
33          print "$rpc_href->{value} bomb makes BOOOOOOOM\n";
34  }});
35
36  ## enter event loop
37  $machine->run;
```

This framework has no security considerations because the testbed system is only available in a separate network with controlled access.

4.2 PLANETLAB – GLOBAL RESEARCH NETWORK

For a short period of time, I acquired access to the *PlanetLab*[6] testbed system. PlanetLab [Pet+06a] is a research overlay network with more than 1000 machines in more than 700 locations which are geographically distributed. All machines are connected to the Internet and therefore experience the problems of unreliable networks including latency, packet loss, and packet corruption, among others. However, it provides an environment to test hypotheses in a large-scale environment. To use the PlanetLab testbed system, a potent server must be provided for the PlanetLab infrastructure.

4.2.1 *Architecture*

The architecture of PlanetLab is built on a special OS which needs to be installed on the server provided to the testbed system. This OS creates Virtual Machines (VMs) on demand to isolate different environments from each other. A researcher can create a *slice*, a set of allocated VMs across PlanetLab. These VMs are accessible via SSH. I refer interested reader to [Pet+06b] for an detailed evaluation of PlanetLab in terms of CPU, memory, bandwidth, jitter, and disk space consumption.

6 https://www.planet-lab.org

4.3 DISCUSSION AND RELATED TESTBED SYSTEMS

In this Section, I would like to discuss why I used testbed systems and compare it with related ones. There are three evaluation methodology which can be used to confirm and refute hypotheses concerning the security of BitTorrent: analytical, simulation and experimental.

BitTorrent is a complex protocol that uses a reciprocal mode of sharing which depends on a number of parameters including the number of leechers, number of seeders, bandwidth capabilities of peers, download and upload speed, used BitTorrent implementation, and the number of pieces, among other factors. The numerous parameters make it unsuitable for simulations and mathematical models. With this in mind, and considering the number of publications that using testbed systems [Wu+10; LUM06; Leg+07; Che+08; EGM; DHW11; Pia+07; Lio+06], I employed real testbed systems as well.

Barcellos, Mansilha, and Brasileiro in [BMB08] present *TorrentLab*, an evaluation platform to investigate BitTorrent by simulations and experiments on live networks. The biggest difference compared to the testbed system from Section 4.1 is that they combined simulations and experimentation on one platform. The advantage of this method is that they can easily compare the results of the simulations with the result of the experiments. In their work, they used 30 machines, a similar number of machines I have used in Section 4.1.

Rao, Legout, and Dabbous investigate in [RLD10] whether it is possible to perform realistic BitTorrent experiments on a cluster. For their experiments they used a single machine with 100 BitTorrent peers and compared it with results from PlanetLab. They came to the conclusion that network latency and packet loss have marginal impact on the download completion time. Therefore, they suggest that a cluster can be used to perform realistic BitTorrent experiments. In comparison, they used a single machine instead of multiple machines. This has the advantage that they can increase the number of BitTorrent peers dynamically.

5

STEALING BANDWIDTH FROM BITTORRENT PEERS

5.1 INTRODUCTION

An attacker who would like to attack BitTorrent has four components as attack vectors: leechers, seeders, peers and torrent discovery. Peer discovery techniques have evolved with the introduction of DHT, PEX and LPD. Additionally, major torrent discovery websites such as *PirateBay* have switched to magnet links [Pir12]. A magnet link is SHA-1 hash which identifies metadata of a torrent. A peer uses this link to download metadata from other participating peers. Therefore, magnet links are more resilient against attacks than *.TORRENT files. This leaves leechers and seeders as the most vulnerable components of this ecosystem. One attack that targets leechers and seeders is a *bandwidth attack*. While significant progress has been made in understanding the BitTorrent choking mechanism, its security vulnerabilities have not been thoroughly investigated.

This chapter provides an analysis of different extensions and choking algorithms in seed state and reveals vulnerabilities that an attacker can exploit. Additionally, I propose a countermeasure against these vulnerabilities and propose a novel seeding algorithm which is more resilient against bandwidth attacks than previous algorithms. I first provide an overview of different seeding algorithms and the BitTorrent Fast Extension.

5.2 BACKGROUND

In this section, I give a brief overview of the BitTorrent Fast Extension and a variety of seeding algorithms.

5.2.1 *Allowed Fast Extension*

Harrison and Cohen published BEP 6 [HCo8] in 2008, in which they describe the *allowed fast extension*. This extension introduces four components: state machine reworking, have-all, suggest-piece, and allowed-fast [Coho8a]. I focus only on the allowed fast extension.

The BitTorrent choking algorithm has one disadvantage. If a new peer joins a swarm, it lacks pieces to offer other peers. This means the peer must wait until another peer optimistically unchokes it (see Section 2.3.1). This process can take, at worst, several minutes. The allowed fast extension increases the speed of this process, enabling the download of a set of pieces even when a peer is choked. This allows a peer to quickly acquire pieces and to engage in BitTorrent's reciprocal system.

If a peer asks for a piece but is choked, the requested peer sends an allowed fast message to this peer which contains a list of pieces it can download. This list of pieces is called the *allowed fast set* and is unique for every peer. This set is generated according to the pseudo code given in Algorithm 5.1.

Algorithm 5.1: Algorithm that generates the allowed fast set according to BEP 6 [HCo8].

```
1  x ⟵ 0xFFFFFF00 & ip
2  x.append(infohash)
3  while |x| < k do
4  |   x ⟵ SHA-1(x)
5  |   for i ← 0 to 4 do
6  |   |   break if(|a| == k)
7  |   |   piece ⟵ partition first 4 Bytes from x
8  |   |   add piece to a if it's not already there
9  |   end
10 end
```

Lines 1 and 2 in Algorithm 5.1 removes the last octet of the IP version 4 (IPv4) address of the requesting peer, concatenate it with the infohash of the torrent, and save it to variable x. The algorithm (line 4) then calculates the SHA-1 hash of the content of variable x and saves it again in x. Lines 5–9 slice 4 bytes from the beginning of the SHA-1 hash and convert this to an integer. This integer is interpreted as a

piece number and a new candidate for the allowed fast set. Line 8 checks if the piece number is not already included in the set. If this is not the case, it is added to the set. This process is repeated until the set contains k pieces (BEP 06 sets $k = 10$).

5.2.2 *Seeding Algorithms*

This section provides details about seeding algorithms that are used by different BitTorrent clients. The seeding algorithm described in BEP 03 [Coho8b] is Fastest Upload (FU).

5.2.2.1 *Fastest Upload*

The FU algorithm is similar to the choking mechanism in leech state (Section 2.3.1), but in this case a seeder considers the download rate instead of the upload rate. Consequently, this algorithm favors the fastest downloaders and does not provide an incentive mechanism. This algorithm was designed based on premise that fast downloaders are also fast uploaders, though it has been shown that this is a false assumption. Asynchronous Internet connections which provide more download capacity and a smaller upload capacity are widely used in home and office networks.

5.2.2.2 *Round Robin*

Round Robin (RR) is a well-known OS scheduling algorithm that allots each process equal processing time. In the case of BitTorrent, RR provides each peer the same number of pieces. This means an upload slot rotates every n pieces.

5.2.2.3 *Anti Leech*

Chow, Golubchik, and Misra in [CGMo8] found that a BitTorrent download from a leecher is slower at the beginning and end of the download. They argue that initially a leecher only has a few pieces and that at the end of the process, it is harder to find a peer with missing pieces. The authors solve this problem by introducing a novel

seeding algorithm that prefers peers with only a few pieces or nearly all pieces. I am not currently aware of its implementation apart from *libtorrent* (version 0.16 or higher) where the algorithm is called Anti Leech (AL). The seeder calculates, for every peer p in the peer list, a score $a(p)$ according to Equation (5.1):

$$a(p) = \begin{cases} f - f(p) & \text{if } f(p) < \frac{f}{2} \\ f(p) \times \frac{1000}{f} & \text{otherwise,} \end{cases} \tag{5.1}$$

where: p = participating peer;
f = number of pieces from the complete file;
$f(p)$ = number of downloaded pieces from a peer p.

The seeder then sorts all peers according to $a(p)$ value and the first three peers are unchoked.

5.2.2.4 *Longest Waiter*

Legout, Urvoy-Keller, and Michiardi in [LUM06] found that, starting with version 4.0.0, the mainline client introduced a new seeding algorithm, which I call Longest Waiter (LW). This algorithm sorts all peers in ascending order according to their waiting time. The three peers that have waited the longest are unchoked.

5.3 BANDWIDTH ATTACKS IN DETAIL

Section 3.1.3 describes bandwidth attacks and summarizes related work. In a bandwidth attack that focuses on seeding algorithms, a malicious peer steals bandwidth by occupying the unchoke slot of a victim. Consequently, the download speed of benign leechers is reduced. If the seeder has set a *seeding ratio*[1], the seeder will stop seeding earlier. These attacks are hard to detect because the attacker needs only a misbehaving client, a client that does not follow the BitTorrent specification.

1 A *seeding ratio* is a limit that can be set in BitTorrent clients. If this limit is reached, the client stops seeding.

A seeder maintains n peers in its peer set. From these n peers, the seeder chooses u peers for an unchoke slot and o peers for an optimistic unchoke slot. BEP 03 [Coho8b] sets $u = 3$ and $o = 1$. If an attacker is able to occupy these u slots, other peers may starve. This is because other peers can then only receive pieces through the o optimistic unchoke slots. This slot is hard to attack, because it is composed of randomly choosen peers. Transmission Control Protocol distributes the available bandwidth evenly across all connections. Thus, the bandwidth consumption b from a malicious peer is:

$$b_A = u \times \frac{b}{u+o}, \tag{5.2}$$

where: u = number of peers which occupy an unchoke slot;
o = number of peers which occupy an optimistc unchoke slot;
b = overall bandwidth capabilities of a peer.

The bandwidth consumption of the other peers who receive pieces through optimistic unchoking is then:

$$b_L = o \times \frac{b}{u+o}, \tag{5.3}$$

where: u = number of peers which occupy an unchoke slot;
o = number of peers which occupy an optimistc unchoke slot;
b = overall bandwidth capabilities of a peer.

Because $u > o$, a bandwidth attack can generate significant damage to a BitTorrent swarm. An attacker can generate even more damage when this attack is combined with a Sybil attack. It is easy to detect seeders in a swarm, because they send a complete BITFIELD message within the first data packet. As there are a variety of seeding algorithms, attackers require different strategies to exploit different algorithms. I examined the default seeding algorithms of prominent BitTorrent clients and list these in Table 5.1.

#	Client	Version	Market Share	Seeding Algorithm
1	uTorrent	3.2.2	47.97 %	Longest Wait
2	Vuze	4.8.1.2	22.49 %	Fastest Upload
3	Mainline	7.7.2	13.01 %	Longest Wait
4	Transmission	2.61	7.00 %	Fastest Upload
5	Unknown		5.22 %	n/a
6	Libtorrent	0.16.10	1.02 %	Round Robin

Table 5.1: BitTorrent clients in combination with the seeding algorithm used ordered by market share. Source: own representation based on [Van11b].

5.3.1 *Seeding Algorithms*

In this section, I describe how an attacker can exploit different seeding algorithms to occupy an unfair number of unchoke slots.

5.3.1.1 *Fastest Upload*

This FU algorithm, described in BEP 03, is used by 29.49 % BitTorrent clients. It can be easily attacked if an attacker has a high download capacity [Pia+07]. Because, the seeder sorts all peers according to download rate, an attacker can introduce high bandwidth peers which would occupy the u unchoke slots.

5.3.1.2 *Round Robin*

A single attacker cannot attack RR, because every peer receive the same number of pieces. This makes it impossible for an attacker to gain a permanent slot. If there are multiple attackers, benign peers can be forced to wait longer to acquire pieces.

5.3.1.3 *Anti Leech*

The authors of [CGM08] note that AL could be exploited by pretending to have none or nearly all pieces. However, they argue that this would be difficult because it requires a source code modification. This statement presupposes security as a result of difficulty. I disagree with this notion because source code modification in a BitTorrent client is trivial for an attacker. To avoid such an attack the authors

proposes a simple countermeasure in which the seeder keeps track of uploaded pieces. If a peer has reached the limit, the seeder places this peer on a blacklist. However, this countermeasure is not implemented in *libtorrent*.

5.3.1.4 *Longest Waiter*

The seeding algorithm LW is mostly used by clients from BitTorrent Inc. An attacker cannot gain advantages over benign peers through this algorithm. Every additional attacker who is connected to a seeder increases waiting time for benign peers.

5.3.2 *Programming Errors in Clients*

Programming errors such as heap or buffer overflows, can cause significant damage to a remote host. An attacker may exploit these programming errors to execute malicious code on a host. Programming errors increase the attack vector for P2P applications. If there is a programming error in the seeding algorithm, an attacker can exploit this error, resulting in free riding.

During my research, I found such a programming error in libtorrent. The experimental work in Section 5.4 shows that the seeding algorithm RR is the most vulnerable. Because RR cannot be influenced by an attacker, it was clear that this artifact stemmed from a programming error. Together with the author of libtorrent, I found that a faulty counter variable was responsible for the behavior. This counter counted the number of bytes during the last unchoke round and was reset for every unchoke round (typically every 15 seconds), instead of every unchoke. After I informed the author of this problem, he quickly wrote a patch which solved the issue for future versions. In the following sections, I refer to the fixed RR implementation as RR (fixed) (RF).

5.3.3 *Allowed Fast Extension Attack*

The allowed fast extension has one security property integrated into the algorithm. It prevents multiple IP addresses from the same network from receiving a greater number of pieces than defined in the allowed piece set. This is enforced by Line 1 in Algorithm 5.1. This line removes the last octet from the IP address. This means a piece set is valid for a whole /24 network (254 IP addresses).

Another security consideration is that an attacker repeatedly requires pieces from the allowed piece set. This is possible because, as BEP 06 states, 'A peer MAY reject requests for already Allowed Fast pieces if the local peer lacks sufficient resources, if the requested piece has already been sent to the requesting peer, or if the requesting peer is not a starting peer'. According to RFC 2119 [Bra97] 'MAY' means optional, implying that it is possible to ask for the same pieces repeatedly and only depends on the implementation.

To test this attack, I wrote a simple attack script that exploits the allowed fast attack to steal bandwidth from a peer. The pseudo code of this attack script is shown in Algorithm 5.2.

Algorithm 5.2: Pseudocode of the Fast Extension Attack.

```
1  foreach incoming message M do
2      switch M do
3          case HAVE_ALL or HAVE do
4              Reply with HAVE_NONE
5              Send INTERESTED
6          case ALLOWED_FAST or CHOKE do
7              add piece to S_1
8              begin thread with endless loop
9                  forall pieces of S_1 do
10                     Send REQUEST for piece
11                     Wait n seconds
12                 end
13             end
14     end
15 end
```

The attack script begins with a BitTorrent handshake to the target, to which the target responds with its own BitTorrent handshake. At this point, the target indicates if the client supports the allowed fast

extension. After the handshake, the client sends an ALLOWED_FAST message which includes a piece number. The exploit presented in Algorithm 5.2 starts a new thread (lines 9–14) for every ALLOWED_FAST message that arrives. The thread contains an endless loop that requires all pieces the victim indicates by the ALLOWED_FAST messages. This allows an attacker to consistently acquire bandwidth from a BitTorrent peer.

Table 5.2 lists prominent BitTorrent clients that I have tested for the allowed fast attack. The uTorrent and Mainline client support allowed fast extension, but only half of the semantics. This is because BitTorrent Inc. has asked academics, e. g. [Haro8a], to study the consequences of this extension. When more experimental results are available, this extension will be activated by default. To the best of my knowledge, this is the first security investigation into this extension.

#	Client	Version	Market Share	Vulnerable
1	uTorrent	3.2.2	47.97 %	No
2	Vuze	4.8.1.2	22.49 %	Yes
3	Mainline	7.7.2	13.01 %	No[2]
4	Transmission	2.61	7.00 %	No
5	Unknown		5.22 %	n/a
6	Libtorrent	0.16.10	1.02 %	Yes

Table 5.2: BitTorrent clients ordered by market share according to [Van11b]. Column *Vulnerable* shows if the client supports the allowed fast semantics and if it is vulnerable to the proposed attack. Source: own representation based on own survey.

Vuze is partially vulnerable to the allowed fast attack. This is because after a client has downloaded a piece from the allowed fast set 64 times, the client rejects all further requests. Still, I listed Vuze as vulnerable in Table 5.2 because an attacker can easily restart an attack with the request counter returning to zero. Transmission has the extension included in the source code but only as a comment. It is likely that it will be commented out in future releases. The next section examines this attack in an actual network to understand how the protocol reacts.

2 At the moment, but code is in mainline so will be vulnerable soon.

5.4 EXPERIMENTAL EVALUATION

I performed experiments using private torrents in a local testbed (Section 4.1) and with the global testbed Planetlab (Section 4.2). All experiments simulated a *flash crowd*[3] scenario. Every leecher disconnected after receiving a complete copy of the file and only the initial seeder stayed connected for the complete duration of the experiment. This is because BitTorrent does not reward active seeders. To create a more realistic scenario, every node generated random HTTP background traffic, as researchers such as [VV08] hypothesize that background traffic impacts performance of distributed applications. Similar to [Dhu+08c; DHW11], I use a *delay ratio* (d) metric to quantify the effectiveness of an attack:

$$d = \frac{t_d'(x) - t_d(x)}{t_d(x)}, \tag{5.4}$$

where: x = arbitrary piece;

t_d = average download time of x without attack;

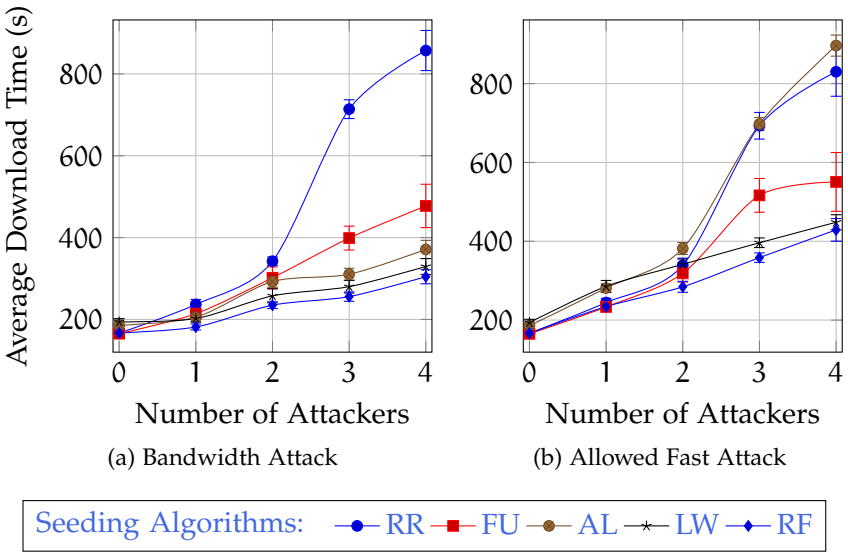
t_d' = average download time of an ongoing attack.

In the following experimental evaluation, the Figures show the average download time which I have measured and the paragraphs use the delay ratio metric to put the results into perspective.

5.4.1 *Experimentation on Local Cluster Testbed*

In this experiment, I attacked the initial seeder with 1, 2, 3 and 4 attackers and compared the results to an experiment without an attacker. I repeated this experiment for every seeding algorithm discussed in Section 5.2.2. The upload limit from the seeder was set to 1, 5 and 10 Mbps. Leechers did not have upload or download limits. All results are the average values of ten iterations.

Results of the Experiment with a Seeder with 1 Mbps



(a) Bandwidth Attack (b) Allowed Fast Attack

Figure 5.1: File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has a 1 Mbps upload limit. (a) A normal bandwidth attack. (b) A bandwidth attack combined with the Allowed Fast Attack. The error bars show 95 % confidence intervals. Source: own representation based on own survey.

5.4.1.1 *Seeder with 1 Mbps Upload Limit*

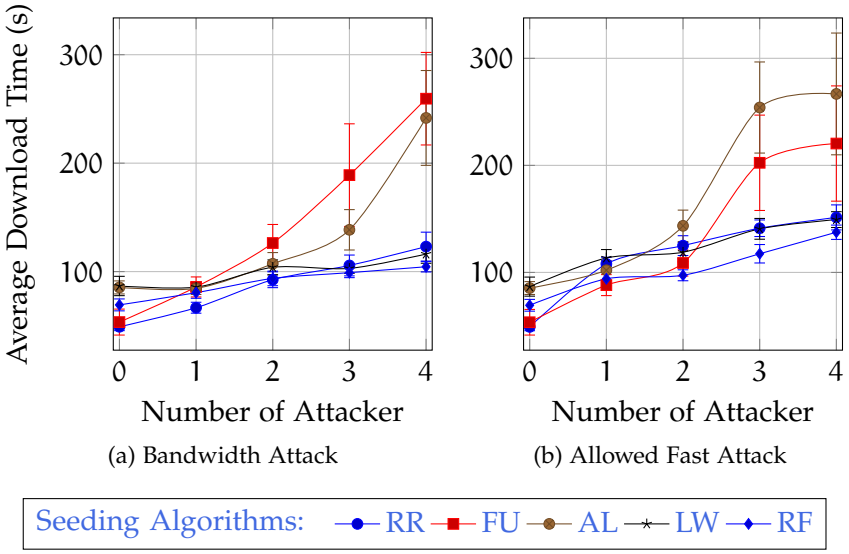
Figure 5.1 shows the average download time with an increasing number of attackers. My initial observations of Figure 5.1 (a) note that RR is the most vulnerable algorithm. With one attacker, the delay ratio d of RR increased by 42.17 %, with two attackers by 105.60 %, with three attackers by 328.76 %, and with four attackers by 414.80 %. This is the highest increase and can be explained by the fact that the RR implementation in libtorrent was incorrect and favored attackers, as explained in Section 5.3.2. The allowed fast attack did not significantly increases in RR, compared to the normal bandwidth attack.

5.4.1.2 *Seeder with 5 Mbps Upload Limit*

Figure 5.2 depicts an attack against a seeder with a 5 Mbps upload limit. Contrary to the attack against the seeder with a 1 Mbps upload limit, the most vulnerable algorithm is not RR, rather it is FU. This indicates that the programming error is only visible when the seeder

3 An experimental setup where all peers begin simultaneously.

Results of the Experiment with a Seeder with 5 Mbps



(a) Bandwidth Attack (b) Allowed Fast Attack

Figure 5.2: File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has 5 Mbps upload limit. (a) A normal bandwidth attack. (b) A bandwidth attack combined with the allowed fast attack. The error bars show 95 % confidence intervals. Source: own representation based on own survey.

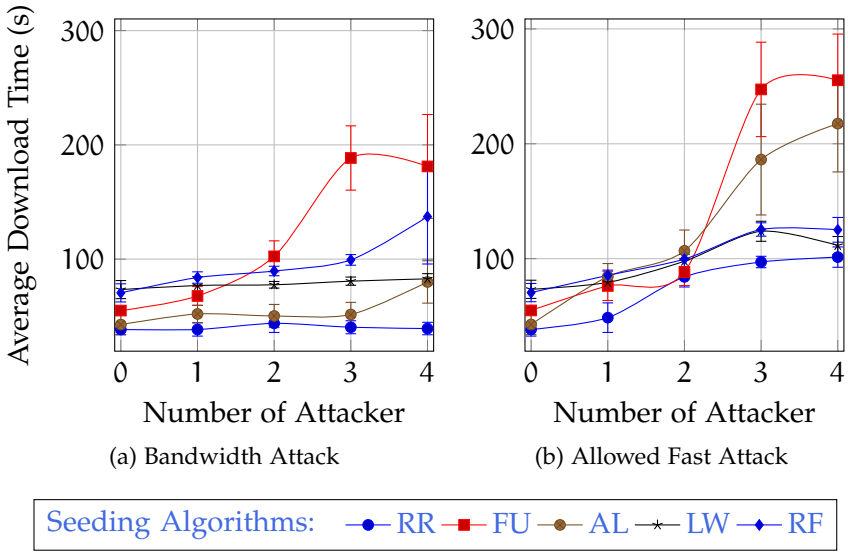
has low bandwidth capabilities. The d of FU increased by 60.64 % with one bandwidth attacker and by 385.41 % with four attackers. As written in Section 5.4.1, leechers and attackers have the same bandwidth capabilities. Nevertheless, the attacker is able to request more pieces than competitive leechers using the simple attack script in Algorithm 5.2.

The next most adversely impacted algorithm is AL. The d of AL during a bandwidth attack with one attacker is not significantly different from the experiment without attacker. However, d with two attackers reaches 25.52 %, three attackers 62.21 %, and four attackers 182.99 %. Similar to the experiment with a 1 Mbps limit against AL, the allowed fast attack increases the impact. The allowed fast attack achieves, against AL, the following d values respectively to the number of attackers: 19.45 %, 68.05 %, 197.44 % and 212.31 %. As in the previous experiment, the seeding algorithms RF and LW are affected the least. The d of LW increases 33.93 % during a normal bandwidth attack with four attacker and 72.19 % during an allowed fast attack.

The d of RF also increases by 50.74 % during a normal bandwidth attack and by 98.44 % with an allowed fast attack with four attackers.

5.4.1.3 *Seeder with 10 Mbps Upload Limit*

Results of the Experiment with a Seeder with 10 Mbps



(a) Bandwidth Attack (b) Allowed Fast Attack

Figure 5.3: File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has 10 Mbps upload limit. (a) A normal bandwidth attack. (b) A bandwidth attack combined with the allowed fast attack. The error bars shows 95 % confidence intervals. Source: own representation based on own survey.

Finally, I repeated the experiment with a seeder with a 10 Mbps upload limit (Figure 5.3). In general, the more bandwidth seeder has, the more resilient it is against bandwidth attacks. In this experiment, the most vulnerable algorithm was again FU. The d of FU with one, two, three, and four bandwidth attackers increased by 23.29 %, 86.36 %, 242.97 %, and 229.54 %, respectively. In and allowed fast extension attack with three attackers, the d was degraded by 349.94 %. The second most vulnerable algorithm was the AL seeding algorithm, similar to the experiment using a 5 Mbps limit. An attack with one attacker increased the d by 97.66 % and by 409.15 % with four attackers. With four attackers, the d of the broken RR implementation increases by 163.94 %, of LW by 52.00 %, and of RF by 77.51 %.

5.4.1.4 *Launching Bandwidth Attacks in Sybil Mode*

In a Sybil attack, an attacker injects multiple fake peers, all of which are under the control of the attacker, into a network (see Section 3.1.1) [Dou02]. This section evaluates the efficacy of the proposed allowed fast attack in Sybil mode.[4] In this experiment, I increased the number of attackers and reduced the number of leechers with every iteration until I had the same number of attackers and leechers. Figure 5.4 depicts the results with a seeder with 5 Mbps upload capacity.

Results of a Sybil Attack with a Seeder with 5 Mbps

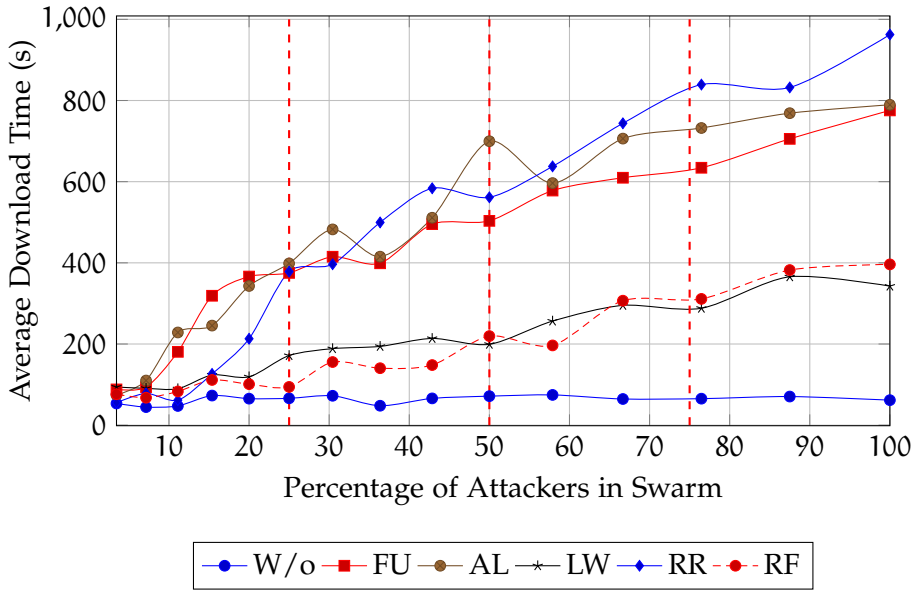


Figure 5.4: File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder that has a 5 Mbps upload limit. A Sybil attack in which I increased the number of attackers and reduced the number of leechers with every iteration. Source: own representation based on own survey.

The Sybil attack shows that as more attackers are injected into the swarm, the impact of the attack becomes progressively severe. When 25 % from the swarm are attackers, it is possible to increase the average download time for all peers by up to more than 500 % if the seeder uses FU, AL or RR. The seeding algorithms LW or RF also concede a d of more than 250 %. If there are half as many attackers as leechers, the d of the leechers increases by up to 700 % if a seeder

4 This is a realistic scenario as botnets are available for hire for as little as $0.50 per bot [Nam09].

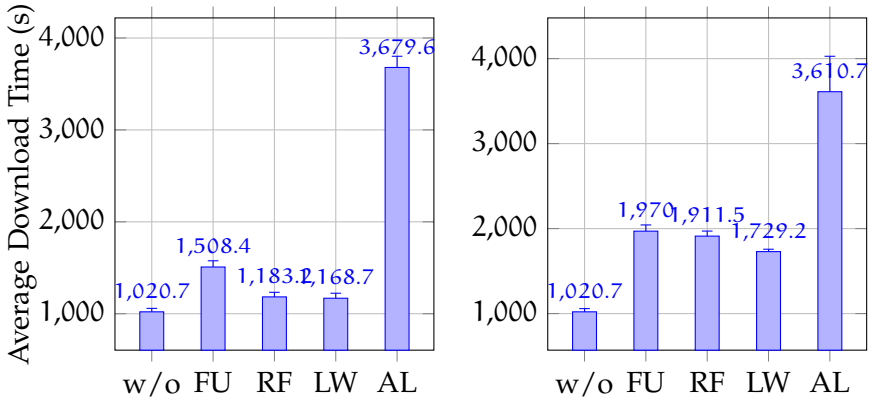
makes use of FU, AL, or RR. However, if a seeder uses LW or RF then this value is more than 300 %. If an attacker introduces the same number of attackers as leechers, then the most vulnerable algorithms FU, AL and RR increase the average download time by up to 1000 %. The effect on RR is the worst with a d increase of 1561.18 %. In LW and RF, the d increases by more than 500 %.

5.4.2 *Experimentation with PlanetLab Testbed*

The following experiments were performed on the PlanetLab platform, I described in detail in Section 4.2, to test the large scale effects of bandwidth attacks. The current study's large scale experimental structure consisted of one seeder, one tracker, and 300 leechers. This scale of this setup is inspired by similar experiments in previous studies, e. g. [Sir+07], [Pia+07] and [Che+08]. All experiments simulated a *flash crowd* scenario and all peers had an upload and download rate of 1 Mbps. This is a judicious bandwidth number, because much of the developing world still operates within a range of 1 Mbps [Cot13]. I introduced four attackers (1.33 % of the leechers) without bandwidth limits. Figure 5.5 (a) shows the results of a bandwidth attack and Figure 5.5 (b) outlines the results of an allowed fast attack. All experimental results were averaged over ten runs.

The results show that AL is the most vulnerable seeding algorithm with a d of 247.68 %, as a result of its missing security feature. Note that this value is close to the cluster result presented earlier with a d of 275.80 %. Similar to previous experiments, the second most vulnerable algorithm is FU where a bandwidth attacker can increase the average download time of 300 leechers by up to 47.78 % and with an allowed fast attack by up to 92.32 %. Both RF and LW are relatively robust against bandwidth attacks with an increase of 15.92 % and 14.50 % respectively. However, the allowed fast attack is able to increase the average download times of RR and LW by up to 87.23 % and 69.41 %, respectively. I also repeated the experiment with less then 300 leechers and obtained similar results.

Results of the Experiment with PlanetLab



(a) Bandwidth Attack (b) Allowed Fast Attack

Figure 5.5: File transfer on PlanetLab of a 100 MiB file with a piece size of 64 KiB via BitTorrent. The data was produced with 1 seeder with a 1 Mbps upload limit, 300 leechers with 1 Mbps download limits, and 4 malicious peers. All values are the average values of ten iterations. The error bars show 95 % confidence intervals. Source: own representation based on own survey.

5.4.3 *Discussion*

The current study presents an in-depth security analysis of different seeding algorithms with BitTorrent. The results show that the seeding algorithms RR, FU and AL are vulnerable, while RF and LW are more resilient. A malicious peer that exploits these algorithms can increase its seeding score and in return receive more download time. It was not possible to cripple the network completely because of the optimistic unchoke slot. However, with unlimited resources, an attacker can slow down BitTorrent downloads for all peers to an unusable level.

I also conducted large-scale experiments on PlanetLab to examine real-world effects of bandwidth attacks. While the attack's impact on PlanetLab was more contained than in the cluster experiment, attack trends observed in the cluster testbed were validated by the PlanetLab experiments. Though, these experiments contained only one seeder and in a real-world swarm there are typically more seeders. However, a malicious peer can attack multiple seeders simultaneously. It is easy to find all seeders in a swarm. To do so, one must connect to the

tracker frequently and wait for a peer that sends a complete BITFIELD or a HAVE_ALL message. An exception in this instance is a seeder that uses the super-seed feature [Hofo8a].

5.5 COUNTERMEASURES

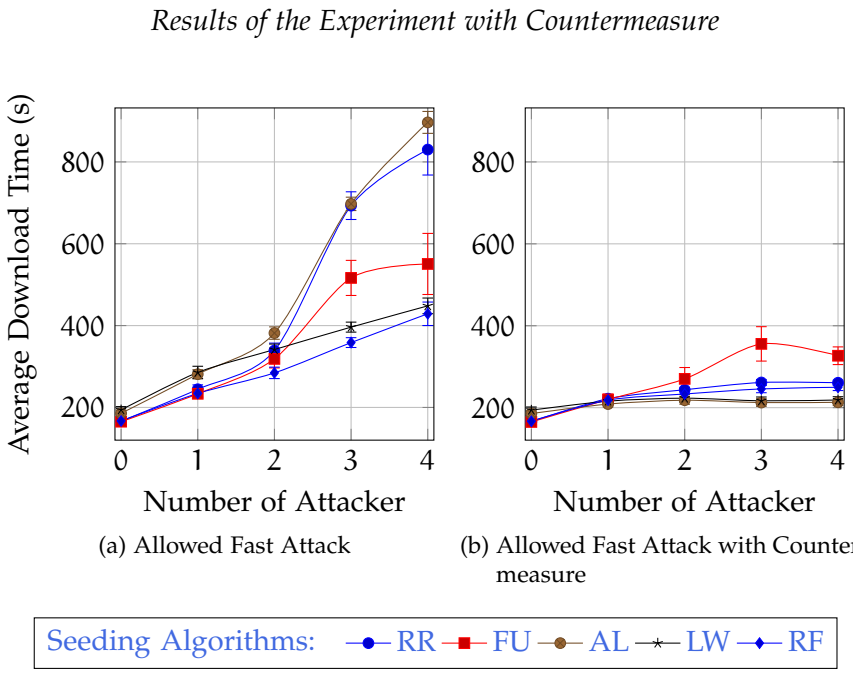
In this section, I propose a countermeasure against the allowed fast attack and a novel seeding algorithm which is highly resilient against bandwidth attacks.

5.5.1 *Allowed Fast Attack*

There are two possible countermeasures against the allowed fast attack. The first is to change the word *'MAY'* to *'MUST'* in the following sentence from BEP 6: '*A peer MAY reject requests for already allowed fast pieces (...)*'. However, this countermeasure prevents retransmission of a damaged piece. Another countermeasure is to upper bound the number of pieces that can be downloaded, for example, the client Vuze has a limit of 64. While this strategy reduces the effectiveness of this attack, it is still possible to restart an attack after the limit is reached.

In light of such factors, an effective countermeasure must restrict the IP address of the peer who has reached the limit to avoid restarting the attack. I implemented this countermeasure in libtorrent and repeated the experiment with a seeder with a 1 Mbps upload limit.

Figure 5.6 shows the experimental results of implementing a countermeasure that limits the number of allowed fast pieces. Figure 5.6 (a) shows the results from the allowed fast attack and Figure 5.6 (b) shows the results with a patched libtorrent version. With the countermeasure in place, the attack has nearly no effect on the seeder. However, the seeding algorithm FU is an exception. In this case, the attacker is not choked from the seeder because it is the fastest down-



(a) Allowed Fast Attack

(b) Allowed Fast Attack with Countermeasure

Figure 5.6: File transfer of a 100 MiB file with a piece size of 64 KiB via BitTorrent with a seeder with a 1 Mbps upload limit. (a) Allowed fast attack. (b) Allowed fast attack with a patched seeder. Source: own representation based on own survey.

loader. My countermeasure strategy against the allowed fast attack has been contributed to the community in libtorrent 0.16.11 [5].

5.6 NOVEL SEEDING ALGORITHM: PEER IDOL

As seen in the experimental evaluation, the seeding algorithm LW and RF are the most resilient against bandwidth attacks. However, if an attacker simply waits for some time, both algorithms will give the attacker an unchoke slot. Therefore, a working countermeasure would require a seeder to unchoke peers who have shared the most. For this, a seeder needs secure proof that a peer has shared pieces with others which is difficult to fake.

5 http://sourceforge.net/p/libtorrent/mailman/message/31298868/

5.6.1 *Algorithm Details*

In this subsection, I introduce a novel seeding algorithm that is difficult to exploit and ensures that only peers who have shared pieces are unchoked. I call this algorithm Peer Idol (PI). This algorithm requires leechers who would like to receive pieces from seeders to send a new message called VOTE to all seeders. This VOTE contains a list of ν peers. I set $\nu = 3$ because this is the number of unchoke slots set by BEP 03 [Coho8b]. The leecher who sends this message provides other leechers for the seeder according to their download rate. The notation $A \succ B$ indicates that the requesting peer has downloaded more from peer A than from peer B. Therefore a VOTE with $\nu = 3$ looks like $(A \succ B \succ C)$.

The seeder who collects the VOTE messages, awards each peer included in this message with points. Peer A receives 3 points, peer B receives 2 points, and lastly peer C receives 1 point. Every unchoke round, the seeder sorts all nominated candidates by score and unchokes the three candidate with the highest scores. Similar to the other seeding algorithms, the unchoked peers have two consecutive rounds to download pieces from the seeder. This avoids quick chocking and unchoking, known as *fibrillation* [Coho8b]. In mathematical terms, if $N = \{1, 2, \ldots, \nu\}$ is the peer set of the seeder, then $I \subseteq N$ contains the peers which are interested. For every peer $p \in I$, the PI score is calculated as follows:

$$PI(p) = \sum_{i=1}^{|I|} V_i(p). \tag{5.5}$$

where: I = list of interested peers;

$V_i(p)$ = returns 1 if peer i voted for peer p and 0 if not.

If two peers, p_1 and p_2, have the same score, $PI(p_1) = PI(p_2)$, the peer who has waited the longest receives the higher priority. After each unchoke round, PI resets the score of each peer. Suppose, the VOTE contains peers to which the seeder does not have a connection. In this case, the seeder can add these peers to a *candidate list*. This list con-

tains potential peers from LPD, PEX and DHT. If a BitTorrent client exhausts of peers, meaning it has fewer peers than MAX_CONNECTION, it randomly chooses a peer from this list and attempts to create a connection.

The scoring mechanism used in PI is a well-known election method used to count votes called Borda Count (BC). It is named after Jean-Charles de Borda [SJ06, p. 97], although has been developed independently multiple times. While developing PI, I also tried the scoring mechanism Condorcet Method (CM). This method gave rise to two difficulties:

- The complexity of CM is $\mathcal{O}(N^2)$, because all peers need to be compared with each other. In BC, however, the complexity is $\mathcal{O}(1)$, because the seeder simply adds votes together.
- The score of the last peer would be $CM(p) = 0$, because it would lose all comparisons.

To make PI more secure, I defined additional security properties. To incentivise the participation of leechers in PI, leechers are required to send VOTE messages to a seeder to be unchoked. This ensures that leechers send votes to a seeder. A seeder can make an exception if there are not enough peers for uploading. A misbehaving peer is disconnected and blacklisted if the VOTE meets one of the following conditions:

- more than v peers;
- the IP address of the requesting peer;
- repeated peers.

5.6.2 *Implementation Details*

I implemented PI as an BitTorrent extension that uses the Libtorrent Extension Protocol (LTEP) described in BEP 10 [NSH08]. The source code of this extension is provided in Listing A.1 in the Appendix. By

implementing it as an extension, PI does not interfere with the standard protocol and a peer can easily send votes only to seeders who support this extension. I set the number of votes in a single VOTE message to three for all experiments because this is the number of unchoke slots that BEP 03 [Coho8b] assigns. Nevertheless, PI introduces an extra message overhead which I evaluate in Section 5.6.3.1.

The LTEP headers begins with a 4 byte length field, followed by a 1 byte type field and a 1 byte Extended Message Type (EMT) field. The length field contains the length of the entire message. The type field is set to 20 because this indicates that the message is an LTEP message. The EMT field distinguishes the different extensions. I set the EMT field to 23, because this number is not currently used by another extension. This investigation does not consider IP version 6 (IPv6). However, the current implementation could be extended with ease. Instead of a VOTE message, a leecher could send a VOTE6 message which contains 18-octet IPv6 addresses.

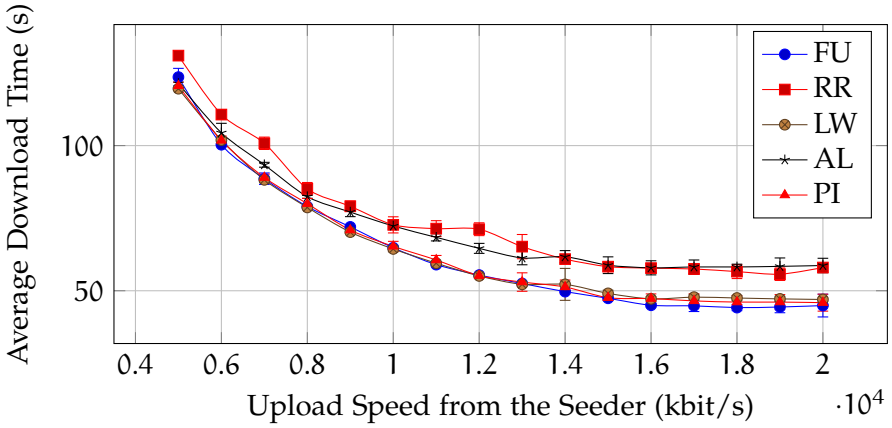
5.6.3 *Experimental Evaluation*

In this section, I detail experiments with PI using the testbed system described in Section 4.1. These experiments analyze the performance of PI in different environments. I will also detail experiments which examine the stability of PI, including scenarios in which peers go offline. This last experiment provides insights into the security features of PI.

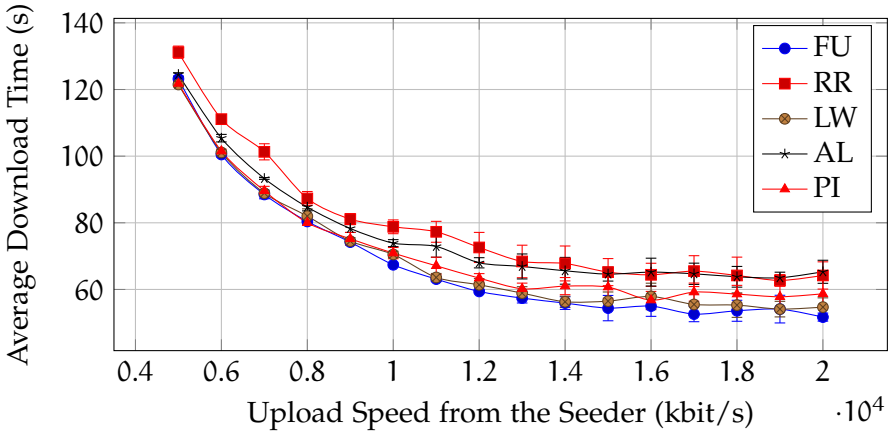
5.6.3.1 *Performance*

Performance is the unique selling point of BitTorrent. A new seeding algorithm should not make BitTorrent slower. Therefore, I designed two performance experiments to test the speed of PI in various environments. The hypotheses is that PI will not be slower than other seeding algorithms even with the message overhead described in Section 5.6.1. This is based on the observation that PI favors sharing peers. In the first experiment, I compared the performance of the seeding algorithms in an optimal environment where leechers do not have

Results of the Performance Experiment



(a) Leechers have no upload or download limit



(b) Leechers have different download limits and no upload limit

Figure 5.7: Effects of seeding algorithms on the average download speed in different environments. The upload speed of the seeder was gradually increased. The error bars show 99 % confidence intervals. Source: own representation based on own survey.

download or upload limits. The seeder had an upload speed limit that was gradually increased. Figure 5.7 (a) shows the average download speed of all peers as it relates to the upload speed of the seeder.

Figure 5.7 (a) shows the results of the first experiment. I limited the domain of all results to 5–20 Mbps, as there is no significant difference between the algorithms in the domain between 1–4 Mbps. At 5 Mbps RR and AL differentiate themselves from FU, LW and PI. Simply stated, RR represents the slower group and PI the faster group. The upload speed of the seeding algorithm RR ranges from 55.7–567 s and has a mean value of 120.5 s. Compared with the faster group, the range

is 45.9–566 s and the mean value is 111.5 s. The median difference between the slower and faster groups is 10 s.

5.6.3.2 *Stability*

I interpret the notion of stability for BitTorrent as a condition in which the service operates as expected even if some peers go offline. Suppose several peers go offline in a swarm with a low peer set cardinality. Consequently, peers either starve without enough peers to finish the download or peers have to request more peers from the tracker, increasing download time. Both scenarios disrupt the stability of the service. Thus, I conducted an experiment in which I counted the number of peers connected to the seeder. Table 5.8 shows the number peers connected to the seeder.

Results of the Stability Experiment

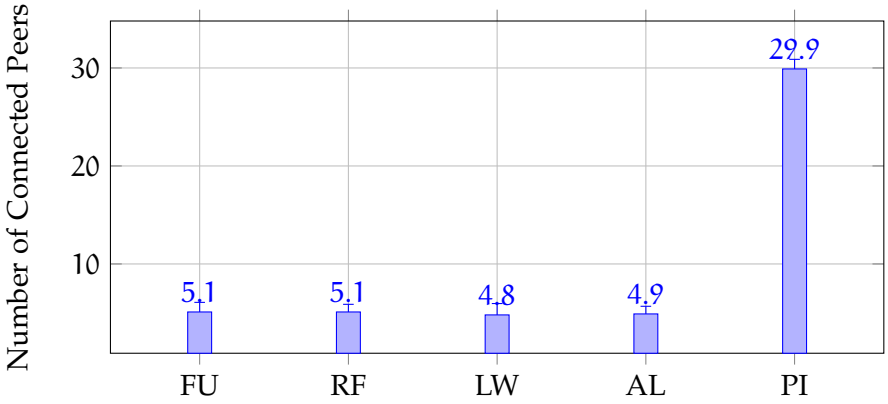


Figure 5.8: The number of peers connected to the seeder in a swarm of 32 peers. All values are average values of ten iterations. The error bars show 99 % confidence intervals. Notice that it can be observed that a seeder that makes use of Peer Idol (PI) has more connections to other peers than the other seeding algorithms. This improves the stability and robustness of BitTorrent. Source: own representation based on own survey.

All seeding algorithms with the exception of PI use only the tracker for new peers. The results of this experiment show these these algorithms request the tracker once or twice for new peers. The seeder that makes use of PI had a connection to nearly all peers. This is because with every VOTE that contains unknown peers, the seeder saves these peers to a list of potential candidates for connection. If a seeder has fewer peers, it randomly chooses a peer from this list and

attempts to form a connection. This mechansim lowers dependency on central tracker and increases the stability and robustness of BitTorrent. Wu et al. in [Wu+10] studied peer exchange in BitTorrent and concluded that peer exchange significantly reduces download time.

5.6.3.3 *Security*

In the next experiment, I investigate how vulnerable the PI algorithm is to bandwidth attacks and compare the results with the other algorithms. For that purpose, I included 3 malicious peers in the experiment that attack each algorithm in its own way. The attackers connect to the seeder 5 seconds before the leechers. This ensures that the attackers are getting the unchoke slots first. I measured how many attackers and leechers were unchoked. The comparison can be seen in Figure 5.9.

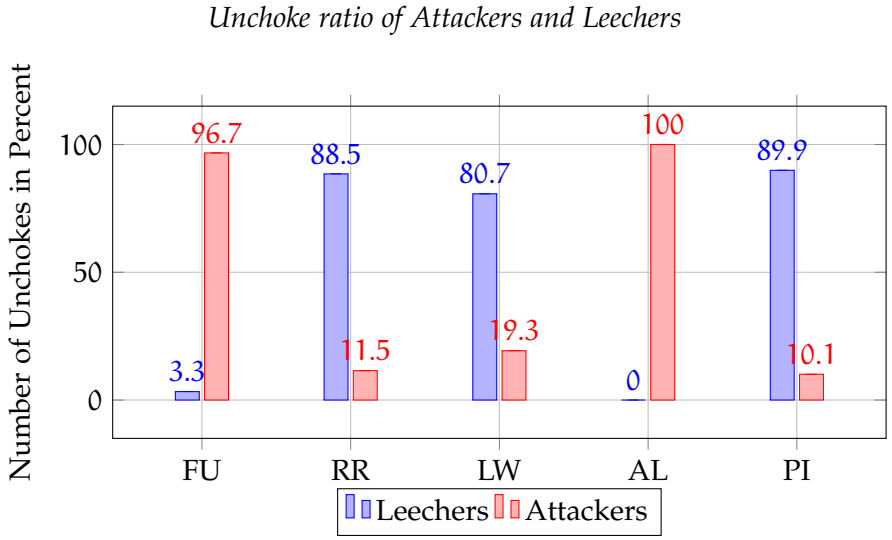


Figure 5.9: The unchoke ratio of attackers and leechers in different seeding algorithms without an optimistic unchoke slot. The data was produced with 1 seeder with a 5 Mbps upload limit, 29 leechers with 900 kbit/s download limits and 3 malicious peers. All values are averages of ten iterations. The 99 % confidence intervals of all values is < 0.05. Source: own representation based on own survey.

To exploit FU, malicious peers simply have to download faster their competitors. Thus, I equipped attackers with more bandwidth their competitors. The attack script requests random blocks and attempts to download as much as possible. The results in Figure 5.9 show that

FU is vulnerable against these types of attacks. Only 3.3 % of leechers were unchoked, whereas 96.7 % of attackers were unchoked. This confirms that FU unchokes the fastest downloaders and shows that, because attackers were faster and connected to the seeder before benign peers, FU almost exclusively unchoked attackers.

To attack RR, I used the same attack script against FU, because there is no vulnerability to be exploited except the introduction of more attackers. In comparison to FU, RR unchokes 88.5 % of leechers and only 11.5 % of attackers. The probability that RR chooses an attacker is given in Equation (5.6).

$$(P(A) = \frac{n_a}{n_p}), \tag{5.6}$$

where: n_a = number of attackers;

n_p = number of peers the seeder has in its peer set.

Therefore, the probability that a leecher is chosen by RR is $P(L) = \overline{P(A)} = \frac{n_p - n_a}{n_p}$. RR is quite robust against bandwidth attacks and provides each peer with the same number of pieces.

In AL, peers that have nearly all or barely any pieces are favored. To exploit this algorithm, the attacker does not send HAVE messages to the seeder. As a result, $F(p)$ from Equation (5.1) is always 0 and therefore $AL(p) = F$, the highest score for a peer. Benign leechers have to be content with an optimistic unchoke slot. The AL score of benign peers upon receiving a single piece is lower than malicious peers. This security weakness is described in the discussion section of the paper [CGM08]. The authors also present a solution to prevent such an attack, however it is not implemented in the current libtorrent version.

To exploit PI, the attack script sends a VOTE to the seeder every 10 seconds, which contains other attackers. Attackers can only VOTE for $(n - 1)$ attackers, because the requesting peer is not allowed to include itself to the VOTE. This means, the VOTE from attackers contains only two valid peers. The results indicate that the PI algorithm is the most robust algorithm against bandwidth attacks. It can be seen

in Figure 5.9 that PI unchokes 10.1 % of attackers and 89.9 % of the leechers.

What impact does the unchoke ratio have on the average download time of leechers? In another experiment, I included 3 malicious peers with no upload or download limits and 29 leechers with upload limits of 20 Mbps. As in the previous experiment, I attacked each seeding algorithm differently and gradually increased the upload limit of the seeder. The results are shown in Figure 5.10.

Average Download Speed of all Seeding Algorithms under Attack

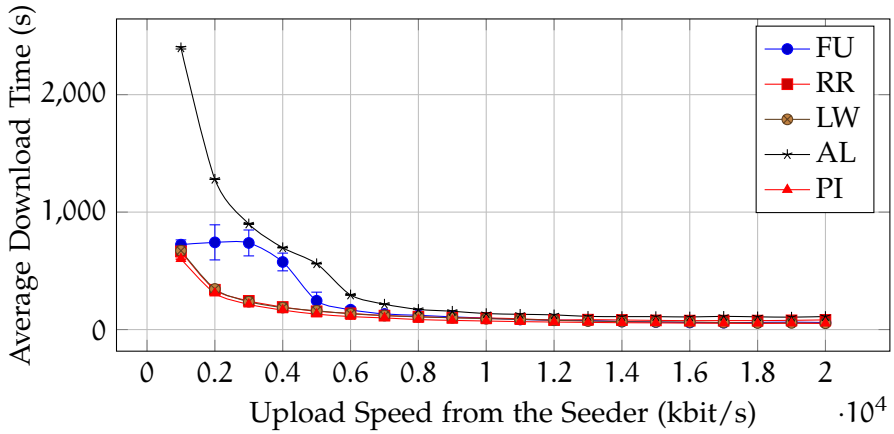


Figure 5.10: Average download speed of the different seeding algorithm with 3 attackers with no upload or download limit and 29 leechers with an upload limit of 20 Mbps. The error bars show 99 % confidence intervals. Source: own representation based on own survey.

In this case, leechers who use AL as a seeding algorithm suffered the most from attacks against the seeder. Their average download time ranged from 105.6–2398.8 s which results in a mean of 397.1 s. The next most vulnerable algorithm was FU. The average download time of leechers with FU from the seeder ranged from 58.5–742.4 s with a mean of 216.6 s. I observed higher deviations when the seeder has a low upload capacity. A low upload capacity suggests that attackers and leechers download at nearly with the same speed. This means, the download speed of FU depends on the efficiency of the attackers and leechers.

The download time of leechers using RR ranged from 76.2–665.7 s with a mean of 148.5 s. After 12 Mbps, RR becomes slower than LW.

The download time of leechers using LW ranged from 53.5–671.8 s with mean of 140.9 s. The seeding algorithm PI is the fastest. The average download time of leechers using PI ranged from 52.5–602.2 s with a mean of 122.2 s.

5.7 COMPARISON WITH RELATED WORK

Dhungel et al. in [Dhu+08a] provides the first investigation of bandwidth and connection attacks on BitTorrent. The study defines bandwidth attacks as situations in which peers try to allocate an upload slot from the seeder as soon as possible to stop the seeder. Their measurements show that bandwidth attacks are ineffective and that is only possible to increase download time by up to 10 %. In [DHW11], the authors came to the conclusion that it is not possible to stop the seeder. However, the current study shows that bandwidth attacks can be effectively launched against seeders. The same authors also studied connection and piece attacks against leechers in detail [DWR09].

In [Lio+06], Liogkas et al. designed and implemented three selfish-peer exploits to obtain bandwidth without sharing pieces with other peers. In the first exploit, the client only downloads pieces from the seeder. Seeders can be easily identified as they advertise themselves by sending a HAVE_ALL message or a complete bitfield. The second exploit attempts to download only from the fastest peers. This exploit observes the frequency of HAVE messages from the victim. This information is exploited to roughly calculate the download rate of a peer. The last exploit introduces fake but seemingly rare pieces to attract high bandwidth leechers. This attack exploits a vulnerability in which a peer can announce pieces which it does not own. They concluded that their exploits delivered significant benefits but also that BitTorrent proved to be quite robust against such attacks. Extending this work, Locher et al. in [Loc+06] developed a selfish BitTorrent client called *BitThief* which never delivers content to other peers. This client exploits optimistic unchoking, does not perform chokes or unchokes, and never announces any pieces. The results the study show that BitThief can succeed in downloading a complete file in all situations. In

rare cases, their client even outperformed the mainline client. In both of these studies, the focus of the attack was the download of a complete file without sharing upload bandwidth. While prior work in this domain focusses on downloading complete files, I instead investigate the effectiveness of attackers who are only interested in degrading system efficiency and not interested in data integrity.

El Defrawy, Gjoka, and Markopoulou in [EGM] show that it is possible to launch a DDoS attack on BitTorrent. A DDoS attack is considered a bandwidth attacks. In this attack, the attacker makes a victim as a trackers. Consequently, all future peers who attempt to contact the victim flood the victim with BitTorrent packets.

Piatek et al. in [Pia+07] measured millions of BitTorrent users and showed that the performance and availability of BitTorrent is quite poor. These measurements motivated the authors to design and implement a new one-hop reputation protocol for P2P networks. In principle, this protocol encourages persistent contribution incentives and rewards contributions. Every client maintains a history of interactions which serve as intermediaries attesting to the behavior of others. While this protocol limits free-riding, it is hard to compare this protocol with the seeding algorithm proposed in this thesis (Section 5.6) because one-hop reciprocation changes standard BitTorrent protocol behaviors. There is, however, a BEP in development to provide an incentive mechanism to users to remain a seeder [Sil15].

5.8 SUMMARY

I considered an important threat against seeding algorithms in BitTorrent and proposed a countermeasure against bandwidth attacks. This novel seeding algorithm for BitTorrent that I call Peer Idol introduces a new message type which contains votes for other peers. I evaluated this algorithm experimentally in terms of performance, security, and stability. The results support the hypotheses that PI is more robust against bandwidth attacks and does decrease in performance in comparison with other algorithms. In the current study's experiment, PI was faster than RR and AL. I have shown that it is not a disadvantage

if votes contain peers which a seeder does not have in its peer set. These peers, however, are saved to a candidate list and can be used for later contact. This reduces dependence on a central tracker and increases the stability and robustness of BitTorrent. In summary, the proposed choking algorithm in seed state implements an incentive mechanism in BitTorrent consecutively through all peers. The news site *torrentfreak* has written an article about this research [Van15a].

6

EXPLOIT BANDWIDTH TO CREATE CONGESTION

6.1 INTRODUCTION

In the previous section, I employed attacks against BitTorrent protocol in terms of protocol. In this section, I present an attack on the BitTorrent protocol in terms of transport. As discussed, BitTorrent Inc. has changed their transport protocol from TCP to uTP to become more Internet Service Provider (ISP) friendly. This protocol introduces the new congestion control algorithm LEDBAT. This algorithm assumes that a receiver always provides correct feedback as this otherwise deteriorates throughput. A misbehaving receiver can exploit this behavior to cause congestion and steal large amounts of a victim's bandwidth. In this section, I introduce three attacks which significantly increase the bandwidth use of a victim and countermeasures for against these attacks.

6.2 ATTACK SCENARIOS

The congestion control LEDBAT from uTP assumes a receiver provides correct feedback, as this would otherwise deteriorate the throughput or yield corrupted data. An attacker who is not interested in data integrity can exploit this behavior to induce a sender to increasingly send packets into a network. The next section examines two possible attack scenarios.

6.2.1 *Congestion Collapse*

A machine which cannot handle the rate of traffic that is arriving must discard incoming packets. This state is called *congested*. The state

in which a network is so heavily congested that is nearly unusable is called *congestion collapse*. These states make it necessary for a transfer protocol which carries mainly bulk data, to have congestion control mechanisms to avoid or react effectively to such situations [FS12, p. 727–728].

Congestion is possible if a victim with high bandwidth capabilities has a machine on its network path which does not have the same bandwidth capacity. The effects of congestion are well-known and include packet loss, queuing delay, and the blocking of new connections. Congestion control in the transport protocol often manages such problems. If a malicious peer can trick the congestion control then it is possible to purposefully create congestion on purpose on a given path. A possible scenario is depicted in Figure 6.1.

Possible Attack Scenario

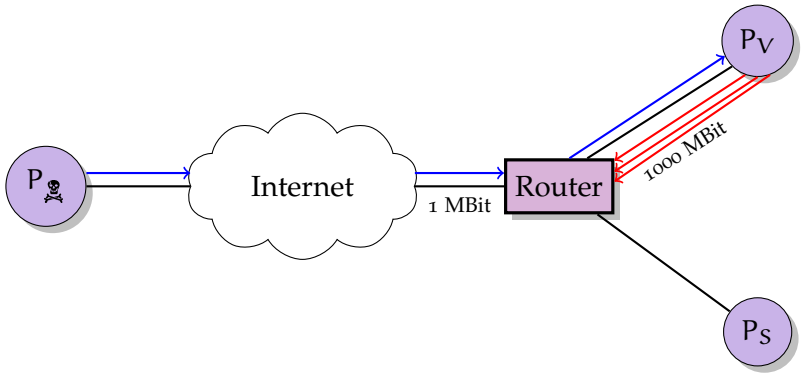


Figure 6.1: A possible scenario where an attacker exploits the congestion control to create congestion on a slow path. The blue lines represent the traffic from an attacker and the red lines represent traffic generated by a victim. Source: own representation.

Figure 6.1 shows an attacker $P_{\text{☠}}$ and a victim P_v with a server P_s that are both behind a router. This router is connected to a 1 Mbit Internet connection. If the attacker $P_{\text{☠}}$ exploits the congestion control of P_V, P_V floods its own Internet connection which may create congestion. Consequently, other clients may have trouble to reach P_V and P_S as they both share the same Internet connection. A slow machine is not necessary on a given network path, it may be enough if the victim simply uses a Digitial Subscriber Line (DSL) or cable modem. Normally, DSL and cable modems have send buffer which is disproportional to their maximum send rate [Nor10]. However, even if there

is no DSL or cable modem, or a slow machine on a given path, it is still possible to steal bandwidth from a peer.

6.2.2 *Steal Bandwidth*

Because bandwidth is a limited resource, it is important to share fairly. If an attacker can exploit congestion controls to gain bandwidth, other peers will receive less bandwidth. If a P2P network uses bandwidth consumption as an incentive mechanism, this becomes a serious problem, as shown in Section 5. In the next section, I discuss attacks which can exploit the congestion control of uTP.

6.3 DETAILS AND EVALUATION OF ATTACKS

I modified the open-source library *libutp* [1] for this evaluation. This library is written by the developer of BitTorrent and it is used by the following BitTorrent clients: uTorrent, Vuze, Mainline and Transmission. Libutp comes with a test program for receiving UTP_RECV and sending files UTP_SEND.

In the next sections, I describe three attacks against uTP together with the results of the experiments which were run under the following conditions. I used a client-server environment. One computer was a sender with an unmodified version of libutp. The client represents a malicious peer with a modified version of libtup. Both computers were running GNU/Linux and connected via a 100 Mbps switch. Additionally, I introduced a 25 ms delay with a 10 ms variance which was distributed normally with NetEm [Hem05]. I chose these values to simulate a connection between two peers with high speed Internet access who are communicating.

1 https://github.com/bittorrent/libutp

6.3.1 *Lying about the One-way Delay*

According to RFC 6817 [Sha+12], the one-way delay measurement only works with the help of the receiver: 'LEDBAT requires that each data segment carries a "timestamp" from the sender, based on which the receiver computes the one-way delay from the sender and sends this computed value back to the sender.' This one-way delay, which I denote as Δt, is calculated by the receiver by subtracting the timestamp t_{snd} from the current time of the receiver t_{rcv}. The receiver returns Δt back to the sender, as shown in Figure 2.11.

The value Δt is only meaningful as a relative value compared to previous values. This is because the clocks are not synchronized at both endpoints. The sender saves all Δt values from the last two minutes in a vector $\vec{h}$. Before these values are included in $\vec{h}$, Δt is normalized with *delay_base*, the lowest value from $\vec{h}$. Normalization is completed to measure the buffering delay on the socket [Nor10]. In LEDBAT, Equation (2.2) is used to decide whether to increase or decrease the send window. The send window is increased by LEDBAT when the lowest value in vector $\vec{h}$ is smaller than 100 ms. Otherwise, the send window remains constant or is decreased.

Exploiting the One-way Delay Measurement

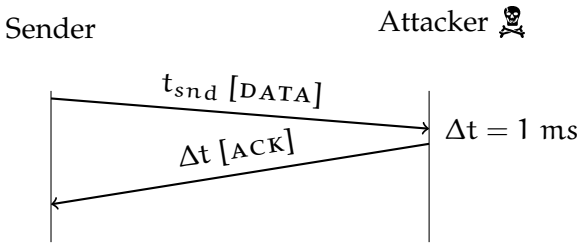


Figure 6.2: One-way delay measurement with a normal receiver and an attacker. Source: own representation.

A malicious receiver can lie about the delay measurement. Figure 6.2 demonstrates how an attacker can pretend that a one-way delay measurement is always $\Delta t = 1$ ms. It is only important that Δt is < 100 ms and constant because the measurements are not interpreted as absolute values. This can create the following situation. Shortly after an attack begins, the *delay_base* is at 1 ms as this is the

lowest value from the last two minutes. All following Δt values are normalized with *delay_base*. Therefore, the vector $\vec{h}$ is filled entirely with zeros. The DELAY_FACTOR makes use of the vector $\vec{h}$ which is partially responsible for the adjustment of the maximal window. Equation (6.1) shows how the variable DELAY_FACTOR is calculated.

$$delay_factor = \frac{100\ ms - min(\vec{h})}{100\ ms} \tag{6.1}$$

where: $\vec{h}$ = history of the last 100 Δt values.

Because all values in $\vec{h}$ are zero, the DELAY_FACTOR is always 1, the highest value. A positive DELAY_FACTOR always increases the window from the sender. The sender receives Δt and increases or decreases its window size MAX_WINDOW according to the pseudo code in Listing 6.1.

Listing 6.1: Source code of the maximal window calculation according to BEP 29 [Nor10].

```
1 scaled_gain = MAX_CWND_INCREASE_PACKETS_PER_RTT * delay_factor *
      window_factor;
2 max_window += scaled_gain;
```

The results of the attack experiment is shown in Figure 6.3 which shows packets per second as dependent on time in seconds. All curves show the bandwidth usage of a file transfer over time. A file transfer with an unmodified receiver has an average value of 632.503 packets/sec. A modified receiver which lies about the one way delay measurement has an average value of 916.507 packets/sec. This shows that the attack just described can increase the bandwidth by approximately 300 packets/sec. This attack is limited, because its increase in bandwidth leads to an increase in packet loss. Every packet loss decreases the send window of the sender. Such a limitation leads to the next possible attack.

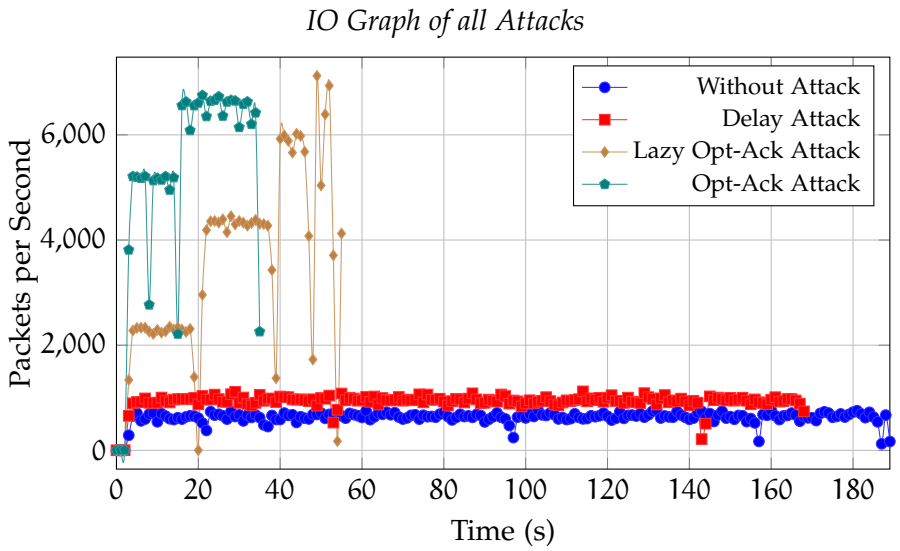


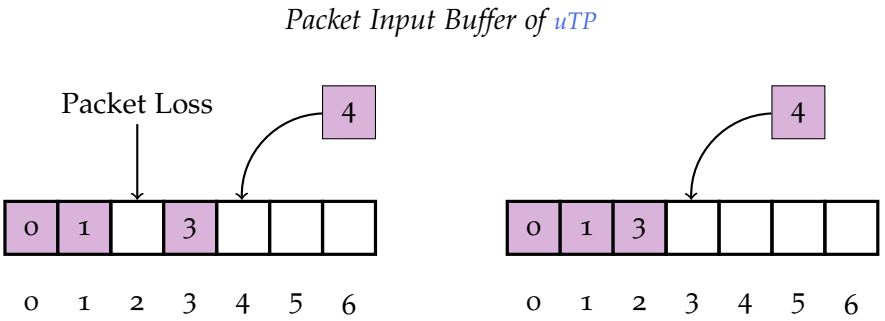
Figure 6.3: File transfer of a 100 MiB file via uTP under the following network conditions: 100 Mbps bandwidth, 25 ms delay, 10 ms variance, and normal distribution. Source: own representation based on own survey.

6.3.2 *Lazy Optimistic Acknowledgment*

Like TCP, uTP also uses packet loss as a sign of congestion and decreases its sending rate. In TCP, when a packet is lost, the sending rate is multiplied by a factor of 0.5. Because this event is less likely in uTP, the sending rate is multiplied by a factor of 0.78 [Nor10]. Again, the sender requires the help of a receiver to understand that a packet is lost. Normally, the receiver sends an SACK with the bitmask of which packets are lost or sends a packet with a duplicated Acknowledgment (ACK) to notify the sender.

The receiver sorts all packets by their sequence number to maintain the integrity of the data. However, a malicious peer can save the packets it receives sequentially. This prevents a gap in the input buffer which is an indicator of a packet loss. Figure 6.4 (a) shows the normal behavior of a receiver. All packets are saved in the input buffer according to their sequence number. The packet with the sequence number 2 is lost which creates a gap between packet 1 and 3. I denote this attack as lazy optimistic ACK (Opt-ACK) attack. Figure 6.4 (b) shows a misbehaving receiver which saves all packets sequentially.

This technique does not create a gap and destroys the order of the file.



(a) Normal (b) Lazy Opt-ACK Attack

Figure 6.4: Comparison of the input buffer of a normal receiver and an attacker. Source: own representation.

A SACK packet is only sent if there is something wrong with the input buffer. Because the input buffer of the modified receiver is always fine, there is no need to send a SACK packet. To inform a sender about the successful acknowledgment of packets, I always send the sender a SACK message with the information that I received all packets. The sender never decreases the sending rate, as a result of misinformation from the receiver. This significantly increases the sending rate of the sender. The experimental results from Figure 6.3 show that the normal file transfer took about 187 seconds. With this attack, the file transfer is complete in ~ 60 seconds. I also increased the average value by 3414.4 packets/sec. This corresponds to an increase of a factor of three. The lazy Opt-ACK attack provides the foundation for the next attack I examine.

6.3.3 *Optimistic Acknowledgment*

At the core of an Opt-ACK attack is the acknowledgment of in-flight packets. When a sender receives an ACK which fits the window, it decreases the CUR_WINDOW by the size of the payload from the acknowledgment packet. The lower the value of the CUR_WINDOW, the more packets a sender is able to send. To acknowledge as many packets as possible, I initiated the bitmask of a SACK packet with

UINT_MAX[2] instead of 0. All bits are set to 1, meaning an attacker pretends to have received all the packets, even those the sender has not yet sent. However, until the acknowledgment arrives to the sender, the sender sends new packets which are automatically acknowledged.

The effect is illustrated in Figure 6.3. The curve of the Opt-ACK attack takes an average value of 37.6 s. This is one-fifth of the download time compared with a file transfer with a normal receiver. The Opt-ACK attack produces an average value of 5073.7 packets per second. Consequently, the Opt-ACK attack increased bandwidth consumption by up to a factor of five.

6.4 DISCUSSION OF THE PROPOSED ATTACKS

The following sections discuss the attacks which were presented.

6.4.1 *Impact of Attacks*

As discussed in Section 6.2, there were two scenarios for an attack, the theft of bandwidth or the creation of congestion collapse. Which of the described attacks can create which forms of damage? To respond to this question, I began a file transfer of a 300 MiB file via uTP without any delay in the current study's 100 Mbps network. Shortly after this, I began a constant UDP stream of 50 Mbps with iPerf [3] for 50 seconds. I first used the unmodified receiver, then modified receiver, from the attacks.

Figure A.1 shows packets per second as dependent on the time. Figure A.1 (a) shows that, when the UDP stream starts, the uTP stream immediately reduces its sending rate to prevent congestion and to prioritize the foreground traffic. Because the delay attack only works when there is a delay and the additional bandwidth consumption is low, a hypothetical Figure would look similar to Figure A.1 (a). Figure A.1 (b) shows the same experiment with a lazy Opt-ACK attack. The sender does not recognize the packet loss and this creates a short

2 Constant which defines the maximum value for an object of type UNSIGNED INT.
3 http://sourceforge.net/projects/iperf/

congestion of between 45–55 s. This corresponds to a packet loss from the constant UDP stream of 7 %.

Figure A.1 (c) shows the results from the experiment with the an Opt-ACK attack. Between 12–37 seconds there is no data from both streams. This corresponds to a packet loss from the UDP stream of 42 %. The constant UDP stream can again send data packets to its destination only when there is a connection timeout from uTP. The experiments show that a delay attack can steal additional bandwidth from a sender. Therefore, with a lazy Opt-ACK and Opt-ACK attack it is possible to create congestion. Such a result necessitates an examination of the limitations of these attacks.

6.4.2 *Comparison of the Attacks*

Figure 6.5 shows all attacks based on bandwidth in Mbps. All values are the average of ten iterations. The first attack increases the bandwidth by up to 1 Mbps. This attack can steal additional bandwidth. The lazy opt-ack attack yields a three fold increase in bandwidth. Without a notification of the packet loss, the sender cannot reduce the window size. The Opt-ACK attack is even more successful and yields a fivefold increase in bandwidth. Both the lazy Opt-ACK and Opt-ACK attack can create congestion.

The limit of these attacks are dependent on the send buffer of the sender. For the test program UTP_SEND, the limit of the send buffer is set to 30.000 bytes, the maximum window size of bytes in flight. The average window size of a normal file transfer is up to 17315.7 bytes. If this value is compared with the window size from the delay attack, the delay attack can apparently increase the average window size by up to 1000 bytes. Both, the lazy Opt-ACK and Opt-ACK attack nearly exhaust the limit of the test program completely. The main difference between the two attacks is that the Opt-ACK attack acknowledges packets faster than the lazy Opt-ACK attack.

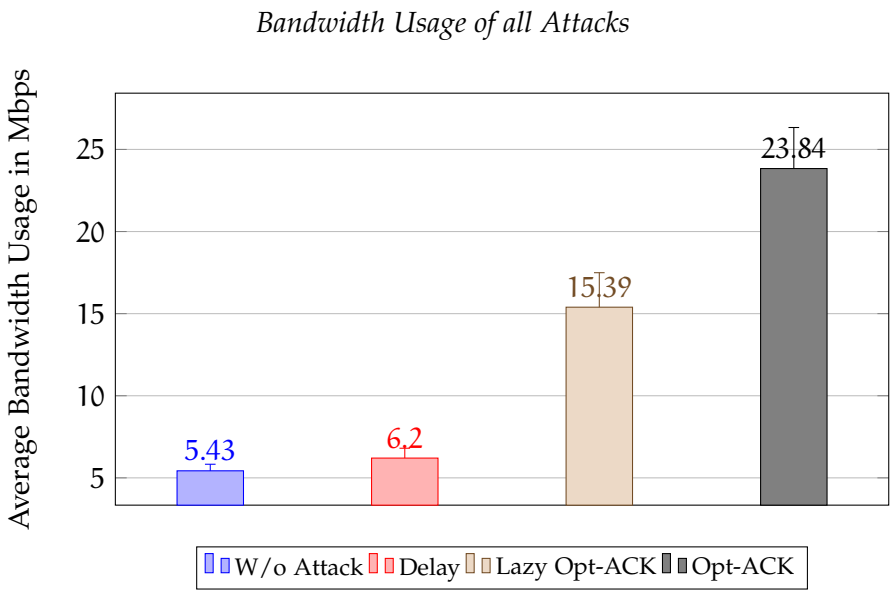


Figure 6.5: Comparison of attacks based on the bandwidth produced. All values are the averages of 10 iterations. The error bars show 95 % confidence intervals. Source: own representation based on own survey.

6.5 COUNTERMEASURES

The attacks that have been presented in this section are often hard to detect because they only differ slightly from behaviors of a normal receiver. Therefore, it is important to have a countermeasure which is efficient and robust against such attacks. In this section, I present a countermeasure against the proposed lazy Opt-ACK and Opt-ACK attack. I evaluate the countermeasure and examine its performance.

6.5.1 *Randomly Skipped Packets*

A sender randomly skips packets and remembers the packets that have been skipped. A normal receiver recognizes a gap in the input buffer and notifies the sender of the missing packet. A receiver does so by either not acknowledging the missing packet or in the form of a SACK packet. The sender then retransmits this packet. An attacker who makes use of the lazy Opt-ACK or Opt-ACK attack does not have this gap, meaning an attacker will nevertheless acknowledge the randomly skipped packet. An attacker betrays itself with this acknowl-

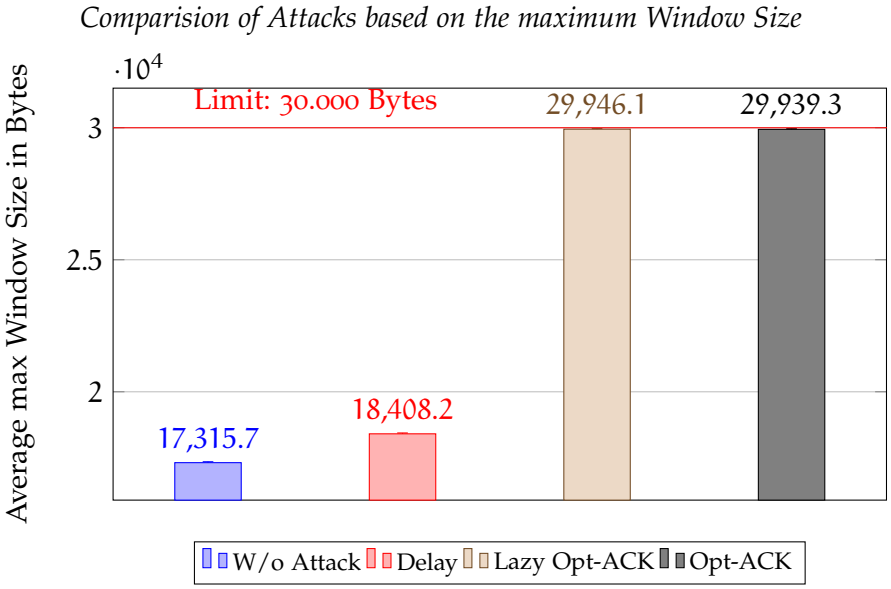


Figure 6.6: All values are averages of 10 iterations. The error bars show 95 % confidence intervals. Source: own representation based on own survey.

edgment. The sender checks if an ACK which has not been sent has been received. A countermeasure of this form is first mentioned by Sherwood, Bhattacharjee, and Braud in [SBB05].

My implementation of this patch used a random value beginning with 500–700 ms. These times were determined in the experiment from Figure 6.7. If the value was smaller it resulted in a lower download speed, if the value was higher it took longer to detect the attack. After the sender recognizes an attack the sender immediately resets the connections.

The next section demonstrates the performance of this countermeasure.

6.5.1.1 *Security Evaluation*

The essence of a countermeasure is the prevention of an attack. To this end, I repeated the experiment from Section 6.3 with both a lazy Opt-ACK and Opt-ACK attack to determine how long it takes a the sender to detect an attack. In the lazy Opt-ACK attack it took approximately 5 s for a sender to detect an attack. In the case of the Opt-ACK attack, it took around 3 s for a sender to detect an attack. The lazy Opt-ACK took longer because the Opt-ACK attack pretends from

the start that it always receives all the packets. There is a close relation between the time in which these attacks are detected and the performance lost. Figure 6.7 illustrates the countermeasure for both attacks.

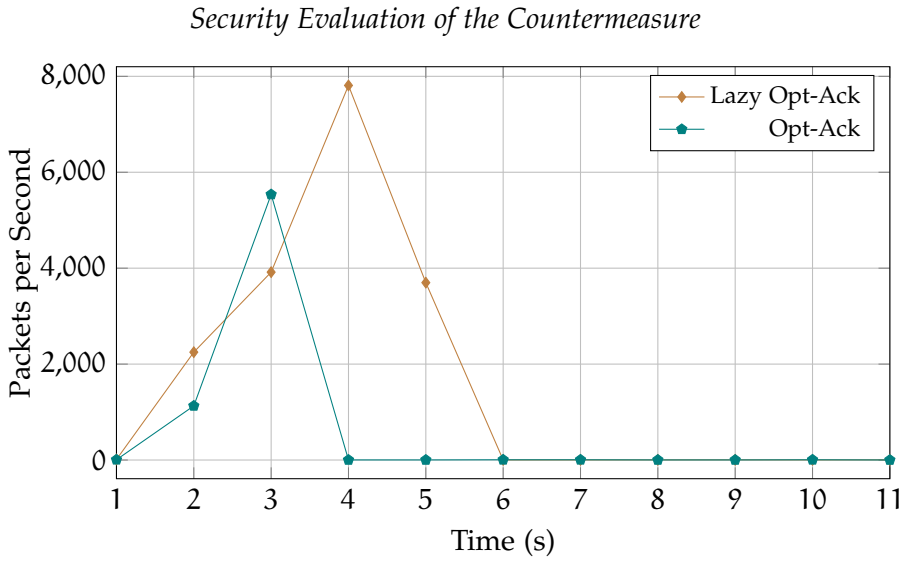


Figure 6.7: File Transfer of a 100 MiB File via uTP under the following network conditions: 100 Mbps bandwidth, 25 ms delay, 10 ms variance and a normal Distribution. It takes approximately 3–5 s to detect both attacks and terminate the connection. Source: own representation based on own survey.

6.5.1.2 *Performance Evaluation*

To evaluate performance loss from the countermeasure wherein the sender randomly skips packet, I setup the following experiment. I wrote a Perl script which automatized the file transfer of a 100 MiB file over uTP. The script increases delay for every iteration by one with a variance of 10 ms and a normal distribution. Every delay value was tested 10 times and the average value was taken. I repeated the experiment for a sender without and a sender with the countermeasures.

Figure 6.8 shows that the difference between a typical sender and patched sender is minimal. To further this, I took the difference of all values and calculated the average value. The average performance loss from the randomly skipped packets countermeasure is 0.448 Mbps. This suggests that the longer the delay is, the smaller the difference between the performance of a normal sender and a patched

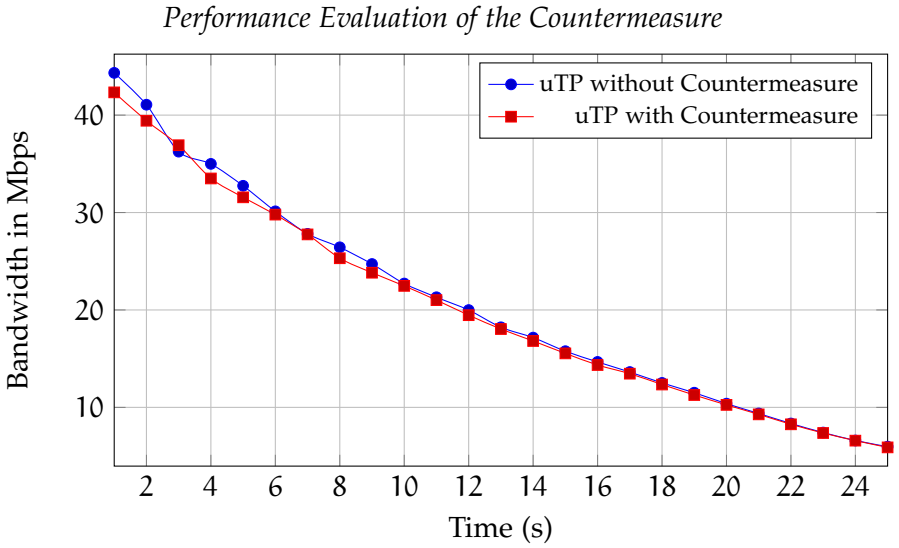


Figure 6.8: File Transfer of a 100 MiB File via uTP under the following network conditions: 100 Mbps bandwidth, 25 ms delay, 10 ms variance and a normal distribution. Source: own representation based on own survey.

sender. The biggest difference is 2.002 Mbps with a 1 ms delay and the smallest difference is 0.015 Mbps with a 24 ms delay.

6.5.2 *Delay Attack*

A countermeasure against the delay attack is more difficult. An attacker can always lie about the one-way delay measurement to induce a sender to send more packets into a network. Congestion control with a one-way delay only works with the help of the receiver and there is no guarantee that a receiver will tell the truth. A potential countermeasure is to make the window calculation less dependent on the one-way delay and include the RTT into the calculation from Listing 6.1. This requires that the countermeasure from Section 6.5.1 is included because the Opt-ACK attack also reduces the RTT. Therefore, a good countermeasure against the delay attack necessitates a complete redesign of LEDBAT congestion control. For this reason, I do not propose a countermeasure against this attack at this point and instead turn to existing research.

6.6 COMPARISON WITH RELATED WORK

In this section, I discuss work related current discussion which provides its foundation. I divide the related work into subsections which examine the exploitation of congestion avoidance algorithms and performance evaluations of LEDBAT.

6.6.1 *Exploitation Congestion Avoidance Control*

Savage et al. in [Sav+99] use a misbehaving TCP receiver to create better end-to-end performance. They named one of their techniques *Optimistic ACKing*. This technique works in the following manner. The receiver sends ACKs which the sender has not yet been sent. This increases the RTT of the sender and in turn increases the send rate. Savage et al. show that these techniques can increase end-to-end performance. They note that these techniques can also be used to generate a SDoS attack.

Sherwood, Bhattacharjee, and Braud in [SBB05] were the first to investigate these techniques from a SDoS perspective. They showed that Opt-ACK attack can cause widespread damage and destabilize a network. The main difference between this Opt-ACK attack against TCP and an Opt-ACK attack against uTP is that uTP does not use RTT to adjust the send rate. However, every valid ACK increases the window size of the sender. As a result of this increase, the sender can send new data which increases the bandwidth. They also engineered a defense strategy which does not require modification in the TCP/IP standard. Unfortunately, this patch has not found its way into the Linux kernel [Hem07]. In contrast, my work provides a performance evaluation of this defense strategy to understand the real performance loss.

6.6.2 *LEDBAT Performance*

Rossi et al. in [Ros+10; RTV10] were the first to evaluate LEDBAT scientifically. They tested LEDBAT in the terms of fairness against competing TCP flows and protocol efficiency in particular. They showed through experimental evidence that LEDBAT reaches some of its design goals such as protocol efficiency, but they also found some issues regarding the fairness of resources.

Abu and Gordon show the impact of delay variability on LEDBAT performance caused by router changes in [AG11]. They came to the conclusion that delay variability can give rise to negative impacts on the throughput.

6.7 SUMMARY

This section proposed the following three attacks for the uTP, a delay attack, a lazy Opt-ACK attack, and an Opt-ACK attack. I provided a detailed evaluation of severity of these attacks in terms of bandwidth consumption and the number of packets and showed the delay attack can steal additional bandwidth and the lazy Opt-ACK and Opt-ACK attack can cause serious congestion. Additionally, I implemented a countermeasure that drops packets randomly to identify misbehaving receivers.

7

REDIRECT BANDWIDTH FROM PEERS TO ARBITRARY VICTIMS

7.1 INTRODUCTION

The objective of a DoS attack in general is to make a service unavailable for legitimate users. Most of these attacks are executed against network connectivity by sending a large amount of network traffic to a given service. These attacks pose a large threat to the Internet. According to a recent global DDoS attack report [Pre14], an average of 28 DDoS attacks occur every hour. The world is facing the next evolution of DDoS attacks.

The company CloudFlare [Pri12] registered a new bandwidth record in 2013 from a DDoS attack. This attack reached 300 Gbps. On average, a DDoS attack can reach around 50 Gbps [13]. A year later, the company reported a new record. This time the impact was around 400 Gbps [Pri14]. In both cases, a new type of DDoS called Distributed Reflective Denial-of-Service (DRDoS) was used.

In the following sections, I first provide background information on this new attack type. I then investigate how the BitTorrent protocol family can be exploited to run such an attack. I then evaluate the impact of this attack and demonstrate countermeasure for BitTorrent against these attacks. Finally, I summarize the contributions of this section.

7.2 BACKGROUND

This section provides a brief overview of DRDoS attacks. Moreover, I highlight vulnerabilities in the BitTorrent protocol that can be exploited.

7.2.1 *Distributed Reflective Denial-of-Service Attacks*

The attacker in a DRDoS attack does not send the traffic directly to the victim. Instead the attacker sends small queries to *reflectors* with a spoofed source IP address of the victim. The reflectors respond to these queries and send these responses to the victim. It is necessary for the exploited protocol to be vulnerable to IP spoofing. If the size of the request packets is smaller than the response packet, this attack *amplifies* the traffic that goes to the victim. Figure 7.1 provides a model of a hypothetical DRDoS attack.

Schematic Diagram of the Threat Model of a DRDoS Attack

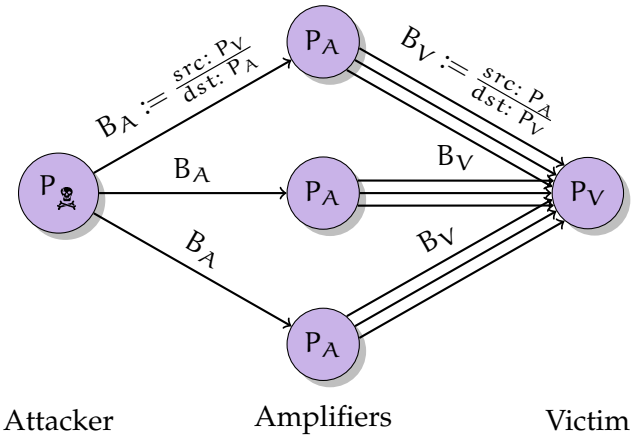


Figure 7.1: Schematic diagram of the threat model of a DRDoS attack. Source: own representation based on [Ros14, p. 2].

The attacker $P_{☠}$ in Figure 7.1 sends forged request packets to the amplifier P_A. This packet is forged, as it does not contain the real source address from the attacker, rather it contains the source address of the victim P_v. An attacker can build its own request packets in user space by making use of *raw sockets*[1]. The amplifier that receives the forged packets responds with B_V to the victim P_V. This attack has several advantages:

- the attacker hides its identity;
- the attack can be initiated by a single attacker but results in a distributed attack;

1 A raw socket is a socket that allows direct sending and receiving of IP packets that the kernel does not support.

- if B_V is larger than B_A, then this attack increases in impact.

To quantify the impact of such an attack, the ratio of B_V and B_A is necessary and is known as the Bandwidth Amplification Factor (BAF) [Ros14, p. 4]:

$$BAF = \frac{|B_v|}{|B_a|}, \tag{7.1}$$

where: $|B_v|$ = size of the amplified payload to the victim,

$|B_a|$ = size of the payload from the attacker.

Both size values do not include Ethernet, IP, or UDP header. This guarantees that BAF values remain valid even if an upper protocol header changes, like in the migration from IPv4 to IPv6. To better understand the impact of a BAF value, consider the following example. A BAF value of 5 means that an attacker with 1 Gbps upload capacity can generate a DRDoS attack that produces 5 Gbps of traffic. Similar to BAF, the Packet Amplification Factor (PAF) is the ratio between the number of packets that are sent from the amplifiers to the victim and the number of packets sent from the attacker to the amplifiers.

7.3 METHODOLOGY

In this section, I provide a brief overview of the details of the current study's analysis of the BitTorrent protocol family.

7.3.1 *Testbed System*

I used the testbed system described in Section 5.4.1 for experimentation as well as smaller testbed system with 3 machines. One machine held the attacker script, one contained the BitTorrent client, and the last machine was a *Raspberry PI*[2] with an open UDP port to analyze network traffic with Wireshark.

2 Is a single-board computer in a size of credit-card.

7.3.2 *uTP Wireshark Dissector Plugin*

Wireshark[3] is an open-source network analyses tool used to capture network traffic and inspect packets. It is, however, not capable of inspecting uTP packets because uTP is a relatively new protocol. Despite this, Wireshark provides a plugin interface for the programming language *Lua*[4]. With this interface it is possible to extend Wireshark and add new protocols. To understand and experiment with uTP it was necessary to write a dissector plugin myself.

7.3.3 *Analyse Script*

To test different BitTorrent clients, I implemented the uTP protocol in a Perl module.

Listing 7.1: Example of the generation of an uTP packet with Perl.

```
my $rawip = Net::RawIP->new({
    ip  => {
        saddr => $SRC,
        daddr => $dst,
    },
    udp => {
        source => $SRC_PORT,
        dest   => $dst_port,
        data   => gen_utp_packet({
            type      => 'st_data',
            vers      => 1,
            extension => 0,
            conn_id   => $conn_id_send,
            seq_nr    => ++$seq_nr,
            ack_nr    => 0,
            wnd_size  => $wnd_size,
            payload   => diffie_hellmann(),
        }),
    },
});

send_packet($rawip1);
```

3 https://www.wireshark.org/
4 http://www.lua.org/

7.4 DETAILS OF THE ATTACK

In this section, I reveal amplification vulnerabilities introduced by the BitTorrent protocol family, including BitTorrent, BitTorrent Sync (BTSync) and uTP.

7.4.1 *Two-way Handshake in uTP*

As described in Section 2.3.3, uTP establishes a connection with a two-way handshake. This allows an attacker to establish a connection with an amplifier using a spoofed IP address. This is possible because the receiver does not verify whether the sender has received the last acknowledgment. To test this attack, I used UCAT attached to libutp. This program is similar to the infamous NETCAT program, except that it uses uTP as its transport protocol.

I setup a receiver (amplifier) that provides an arbitrary file when a connection comes is made and a victim with a *tcpdump*[5] instance. To simulate an attacker, I wrote a Perl script that uses raw sockets. The script sends a forged ST_SYN packet to the amplifier. The amplifier believes this packet to have come from the victim and sends a response (ST_DATA) packet to the spoofed address. Because the forged address does not expect the packet, it does not return an acknowledgment. The receiver runs into timeout and retransmits the lost ST_DATA packet. If four consecutive transmissions have timed out, uTP kills the connection. In this experiment, the receiver sent five packets with a payload size of 1402 bytes, resulting in 7030 sent bytes. According to Equation (7.1), this results in a BAF of 351.5, because the initiator sends only a single ST_SYN packet with a size of 20 bytes.

This gives rise to the following question. Why does the uTP receiver sends only single packet and not more? Micro Transport Protocol does not send more packets, because it implements a slow-start mechanism. The MAX_WINDOW variable in libutp begins with 1382 bytes and increases for every acknowledgment it receives. If the sender

5 A program to capture network traffic exclusively designed for the command line.

does not receive an acknowledgment, it remains to the previous value. To the best of my knowledge, BitTorrent is the only protocol that makes use of uTP. The TOR project, however, evaluated a replacement [LMJ13] but came to the conclusion that is far from trivial to replace TCP with uTP. In the next section, I describe how BitTorrent in connection with uTP can be exploited to run a DRDoS attack.

7.4.2 *Exploiting BitTorrent Handshake via uTP*

After a connection is established, BitTorrent requires a handshake as its first message. This handshake contains the following fields.

PSTRLEN (1 BYTE): Length of the string identifier.

PSTR (19 BYTES): String identifier which is defined in version 1.0 as *BitTorrent protocol*.

RESERVED (8 BYTES): These bytes are reserved to signal the support of different extensions to peers. Each bit can change the behavior of the protocol.

INFOHASH (20 BYTES): Secure Hash Algorithm 1 hash that identifies the torrent file.

PEER ID (20 BYTES): Unique ID that identifies a peer.

If a peer receives a handshake with an infohash with which it does not participate, the peer drops the connection. This means an attacker has to know a valid infohash to exploit the BitTorrent handshake for a DRDoS attack. However, this is a small obstacle, because there are a variety of ways to find peers with valid infohashes, including trackers, DHT, PEX, etc. An attacker can use the BitTorrent handshake to initiate an amplification attack based on the uTP two-way handshake. Figure 7.2 outlines such an attack scenario.

The attacker in Figure 7.2 sends a spoofed ST_SYN packet to the amplifier in *a)*. The amplifier responds to this request with an acknowledgment ST_STATE in *b)*, but to the victim. A connection is established because of the two-way handshake of uTP. The attacker then

DRDoS Attack Scenario with BitTorrent

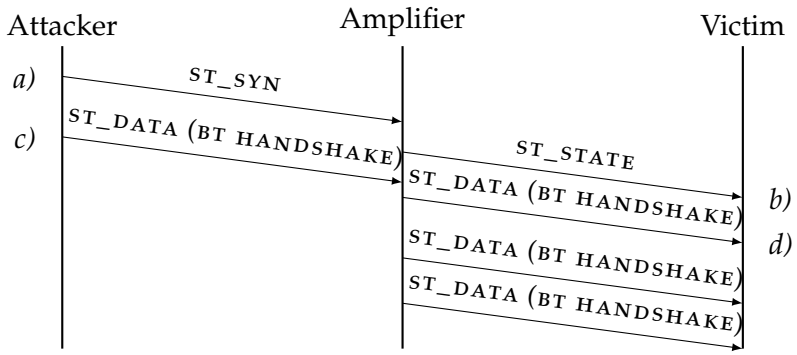


Figure 7.2: In uTP the initiator sends a ST_SYN packet to the receiver. The receiver acknowledges this with a ST_STATE packet. The connection is established (two-way handshake). Source: own representation

sends a BitTorrent handshake message in the first ST_DATA packet in *c)*. Again, the packet contains the source address of the victim.

As discussed, the handshake then requires the infohash of an active torrent from the amplifier. The minimum size of a BitTorrent handshake message is 88 bytes. If the amplifier participates in the torrent, it will respond in *d)* with its own handshake message. Because the handshake in *d)* is un-acknowledged, the amplifier believes the packet is lost and retransmits the handshake 3–4 times until the connection is terminated.

The handshake in *d)* is bigger than the packets from attacker *a)* and *c)* together because of retransmission and *packet stuffing*. Most BitTorrent clients try to put as many BitTorrent messages in one data packet to save additional packets and to be more efficient. In the current study's test, nearly all clients either sent a BITFIELD or multiple *HAVE* messages within the first uTP data packet. Equation (7.2) can be used to measure the impact of an attack that exploits the BitTorrent handshake, abbreviated as *BTH*.

$$\mathrm{BAF}_{\mathrm{BTH}}(p, n) = \frac{20 + 20 + p \times (n + 1)}{20 + 88}, \tag{7.2}$$

where: p = payload size in bytes,

n = number of retransmissions.

All numbers in Equation (7.2) are in bytes. The 40 bytes in the nu-

merator are acknowledgements of two sent packets. The number of retransmissions n is increased by one, because the first regular packet does not belong to the retransmissions. The 20 bytes in the denominator belongs to the ST_SYN packet and the 88 bytes belongs to the ST_DATA packet that contains the BitTorrent handshake.

In the following subsections, I investigate the impact of a possible attack and outline differences between the most commonly used BitTorrent clients listed in Table 5.2.

7.4.2.1 *Mainline and uTorrent clients*

Because mainline BitTorrent[6] version 6.0 is simply uTorrent[7] with a rebranded GUI, I handled both clients together. I tested uTorrent 3.4.2 (built 35702) and mainline BitTorrent 7.9.2 (built 35144). Both clients support *packet stuffing* and place additional BitTorrent messages in the first uTP data packet. In this test, both clients sent one BITFIELD, multiple HAVE messages, and one PORT message. The number of HAVE messages that a client sends depends on the state of the client and the number of pieces the client has downloaded. If a peer is in leech state, it sends the number of pieces that it has already has downloaded, but no more than 24. In seed state, the client always sends 24 HAVE messages, resulting in a BAF of 27.5.

As seen in Section 7.4.2, the BitTorrent handshake includes reserved bytes to indicate different protocol extensions. One disadvantage of this approach is that every time a new extension is developed, the BitTorrent protocol needs to be changed. To solve this, BitTorrent provides LTEP to add new extensions without interfering with the default protocol. If an attacker signals that it supports LTEP, specified in BEP 10 [NSH08], it can further amplify an attack without increasing the size of the handshake. To signal that a peer supports LTEP, a simple bit is set in the extension byte. I observed that in both clients the LTEP handshake is larger than in other clients. This is because uTorrent and Mainline support extensions that are not public, such as UT_HOLEPUNCH and UT_COMMENT. This additional message in

6 http://www.bittorrent.com/downloads/
7 https://www.utorrent.com/

the BitTorrent handshake increases the BAF by up to 39.6. Vuze[8] is the most commonly-used BitTorrent client besides uTorrent and Mainline.

7.4.2.2 *Vuze*

I tested Vuze 5.4.0.0/4 on Windows 7. Without signaling any extensions, Vuze responds with a BitTorrent handshake and a BITFIELD message, that indicates which pieces a peer has downloaded. Vuze retransmits a lost uTP packet four times. This results in a BAF of 13.9. Vuze also supports LTEP, has also designed its own extension protocol called Azureus Message Protocol (AMP) [10a].

If an amplifier runs Vuze with LTEP enabled and an attacker signals that it supports LTEP, the amplifier puts the extension handshake in the first uTP data packet. Compared with uTorrent and Mainline, the handshake is smaller because Vuze does not support the private extensions, which I discussed in Section 7.4.2.1. The BAF increases through LTEP up to 18.7 times.

Vuze uses AMP to transmit a variety of information including normal BitTorrent messages, messages from the chat plugin, or PEX messages. A Vuze client can indicate that it supports AMP by setting the MSB in the first byte from the reserved field in the handshake message. If the attacker sets this bit and a Vuze amplifier supports AMP, then the amplifier adds the following messages to the first uTP packet: AZ_HANDSHAKE, AZ_HAVE, AZ_PEER_EXCHANGE, AZ_REQUEST_HIT, and AZ_STAT_REQ. This increases the handshake by up to 1165 bytes, in turn increasing the BAF_{BTH} by up to 54.3. This value may be even higher if the amplifier shares a large file, because the BITFIELD message depends on the size of the shared file. In our current study's analysis, I used the Ubuntu 14.10 image which is 1.2 GiBs.

8 https://www.vuze.com/

7.4.2.3 *Transmission and LibTorrent*

I tested Transmission[9] 2.84 on Ubuntu 14.04.1. Transmission supports both LTEP and AMP but not *packet stuffing*. This means, that Transmission only sends the handshake message, no matter which extensions are activated. Transmission sends an 88 byte uTP packet to a victim and retransmits a lost packet three times. According to this, an attacker can only achieve a BAF value of 4.0 if the amplifier uses the Transmission client.

Libtorrent[10] is a library written in C++ which is used by over 25 different BitTorrent clients. I tested libtorrent 1.0.2 on Ubuntu 12.04. Like Transmission, libtorrent does not support packet stuffing and only sends a handshake message in the first uTP data packet. However, libtorrent is different from Transmission in terms of the number of retransmissions. Libtorrent resends a lost packet six times, which increases the BAF up to 5.2.

7.4.3 *Exploiting Message Stream Encryption Handshake*

An increasing number of ISPs violate the end-to-end principle of the Internet and discriminate against network applications [Dis+11]. Discriminating means that the ISP limits the traffic of a network application or blocks it completely. Especially bandwidth-hungry P2P applications are shaped, because this traffic creates *transit costs*[11] for an ISP.

To avoid traffic shaping, the BitTorrent community developed Message Stream Encryption (MSE) [14]. The main objective of MSE is to provide payload obfuscation rather than securely encrypt traffic. Brumley and Valkonen showed in [BV08] that MSE has a number of significant weaknesses. It is implemented by most of the BitTorrent clients including uTorrent, BitTorrent mainline, Vuze, Transmission, libtor-

9 http://www.transmissionbt.com/
10 http://www.libtorrent.org/
11 Costs that incurred when network traffic needs to be routed through another network from an ISP.

rent, and BitComet. Figure 7.3 shows a network flow diagram of the Diffie-Hellman handshake of MSE.

Diffie-Hellman Handshake of two BitTorrent Peers

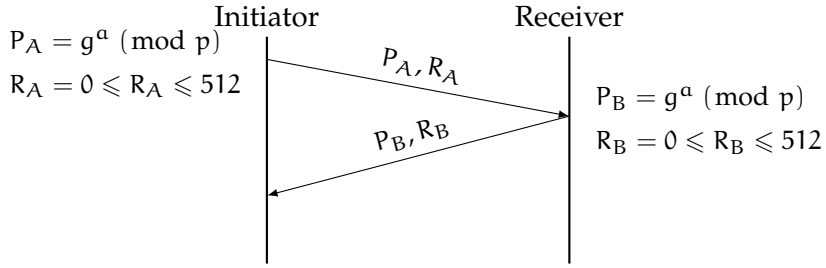


Figure 7.3: Diffie-Hellman handshake of two BitTorrent peers that make use of Message Stream Encryption to avoid traffic shaping. Source: own representation.

The protocol in Figure 7.3 begins with Diffie-Hellman key exchange handshake wherein each peer generates a 768 bit public key, P_I and P_R. To avoid a fixed packet size in each packet, each peer generates random data r_I and r_R with a random length of 0–512 bytes and attaches it to the public key. After the key exchange, the packets are encrypted with RC4. The transport protocol of MSE can either be TCP or uTP. One advantage of this method is that an attacker does not have to know a valid infohash from the amplifier. An attacker can send a spoofed MSE handshake to an amplifier that includes a valid 768 bit public key without random data. Hence, BAF for a client with MSE is:

$$BAF_{MSE}(r,n) = \frac{(116+r)\times(n+1)}{116}, \tag{7.3}$$

where: r = length of the random data (0 bytes $\leqslant r \leqslant$ 512 bytes),

n = number of retransmissions where $n = \{4,5,6\}$.

According to Equation (7.3), BAF_{MSE} ranges from 4, where the amplifier has chosen $r_R = 0$ bytes, to 32.5, where $r_R = 512$ bytes. The payload of the MSE handshake has a high entropy and is therefore hard to detect with an SPI or DPI firewall. However, statistical measurements show good results in detecting MSE [Köh+10; HJ09] but are not widely used. The use of MSE helps to make an attack diffi-

cult to detect and circumvent. To avoid a central tracker, BitTorrent uses a modified DHT protocol based on Kademlia, described in Section 2.2.2.1.

7.4.4 *Exploiting Distributed Hash Table Messages*

The DHT implementation in BitTorrent is divided into two protocols, MLDHT [Loe08] and VDHT [12]. The largest overlay network with users of around 15–27 million [WK13] users per day is MLDHT. Both protocols are not compatible with each other. The BitTorrent protocol uses DHT to find peers that share the same torrent without using a tracker. The following sections examine these two protocols.

7.4.4.1 *Mainline Distributed Hash Table*

The MLDHT protocol supports the following queries: FIND_NODE, PING, GET_PEERS and ANNOUNCE_PEER. All queries and replies are *bencoded*[12]. The PING query attempts to discover if another peer is available. The PING query consists of the peer ID (20 bytes) from the queried peer. Together with the RPC protocol, a ping query has a size of 56 bytes. The response of a PING query has a size of 47 bytes. As such, the PING query does not amplify the bandwidth.

The FIND_NODE query requests the k closest nodes for a specific target. This query is used to find closer nodes to fill buckets without many nodes. The request contains the node ID of both the requesting peer and the target. Together with the Remote Procedure Call (RPC) overhead, the payload of a request packet has a size of 95 bytes. The response contains k of the closest nodes. Because BEP 05 [Loe08] sets $k = 8$, the response varies from 284–332 bytes depending on the implementation. In the study's tests, I found two peers[13] which set $k = 16$. These peers are the bootstrapping peers for most BitTorrent clients. This result in a BAF of 5.2.

12 Is a binary format from BitTorrent to encode structured data (similar to JSON or YAML).

13 ROUTER.UTORRENT.COM and ROUTER.BITTORRENT.COM

The GET_PEERS request is more interesting for an amplification attack as it returns a list of peers for a given infohash. If a peer does not have peers for a specific infohash, it will employ FIND_NODE and return the k closest peers. Therefore, only peers that participate in a swarm are able to send a list of peers in return. According to BEP 05 [Loe08], there is no limitation on the number of peers. In the current study's tests, I found a peer which sends a list of 100 peers in return, resulting in a potential BAF of 11.9. The GET_PEERS request has a WANT option which the requester can set to N4 to request only IPv4 addresses or to N6 to request only IPv6 addresses. If an attacker requests only IPv6 address, the BAF would increase by up to 24.5. With an additional extension, it is possible to further increase this value.

The authors from BEP 33 [10b] describe an extension to MLDHT called DHT *scrapes*. Scrapes are statistics of a swarm such as the number of seeders, leechers, and complete downloads. These statistics are important for decisions to join a swarm without participating in it. These statistics are based on *bloom filters*[14]. To request scrapes, a peer has to send a GET_PEER request that contains the dictionary entry SCRAPE. If the responding peer has database entries for a particular infohash, it returns the statistics in the GET_PEERS response. An attacker that exploits this extension can increase the BAF by up to 13.4 times.

7.4.4.2 *Vuze Distributed Hash Table*

The BAF value of Vuze PING query without any flag is similar to the PING from MLDHT. Vuze, however, supports the Internet coordinate system *Vivaldi* [SB09; Dab+04] which aims to estimate the RTT of other peers without the requirement to send packets to this peer. If the protocol version is $\geqslant$ 10, then the amplifier adds Vivaldi network coordinates to the PING reply packet. This increases the BAF by up to 14.8.

14 Bloom filter is a probabilistic data structure to test if an element is part of a set.

7.4.5 *Exploiting BitTorrent Sync*

BitTorrent Sync[15] is a proprietary P2P protocol from BitTorrent Inc. to synchronize files between different machines. According to a blog post from 2013 [Hon13], BTSync has 1 million users and has synchronized over 30 petabytes of data. This makes it an interesting target for an amplification attack. BTSync uses uTP as its main transport protocol. I investigated three BTSync messages which can be exploited: tracker request, BTSync handshake, and a ping message. For all messages, an attacker needs a valid *BTSync secret*. There are, however, a number of websites[16] where users publish secrets to share their content. An attacker can use these secrets to run an amplification attack.

When a user wants to synchronize a file or directory with BTSync, it generates a unique ID called a BTSync secret. It then contacts a tracker to request peers for this secret. The tracker is run by BitTorrent Inc. and uses the domain `t.usyncapp.com` which is hosted by the Amazon EC2 cloud. The tracker request begins with a uTP ST_SYN packet followed by an ST_DATA packet that contains a bencoded payload which includes peer ID, secret, local address and local port. The tracker responds with a list of peers that also share this secret. The response is larger than the request, however, it highly depends on the number of peers the tracker returns.

The BTSync handshake resembles the BitTorrent handshake. The first data packet contains the secret and a 16 byte NONCE value which may be random. The peer responds with a 160 byte public key and a 16 byte salt value. This creates to a BAF of 10.8.

The BTSync PING starts with the string BSYNC followed by a zero byte and a bencoded dictionary. The dictionary contains the command PING, 20 byte long secret, and 20 byte long peer ID. The UDP payload of this packet is altogether with the overhead of the dictionary 76 bytes. The other side also responds with a PING message, but not just one. It sends 117 PING messages and 12 uTP ST_SYN pack-

15 https://www.getsync.com/
16 http://www.reddit.com/r/btsecrets/ and http://btsynckeys.com/

ets to the requester. A screenshot of this is shown in Figure 7.4. This occurs with BTSync versions 1.4 and 2.0.105 on all platforms.

Response from a BitTorrent Sync Node to single Request

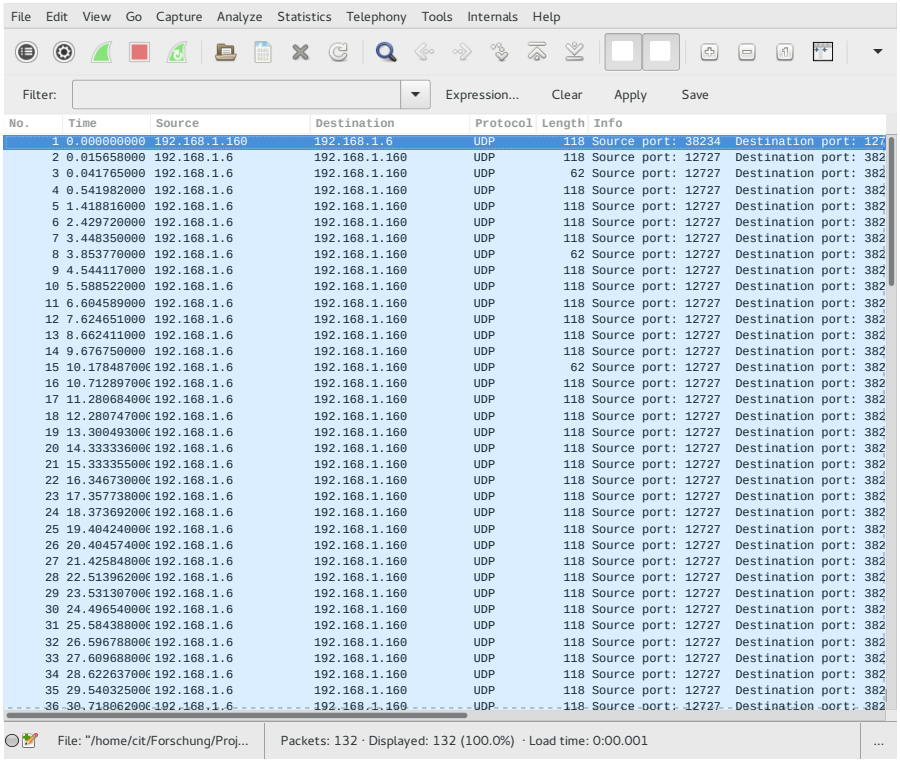


Figure 7.4: Screenshot of the network analyses tool *Wireshark* that displays the responses from a BitTorrent Sync node to a single PING request. Source: own representation.

7.4.6 *Discussion*

To summarize this discussion, I provide a list of all BAF and PAF values in Table A.1. The protocol analysis shows that BitTorrent is highly vulnerable to DRDoS attacks. An attacker is able to amplify BitTorrent traffic by 4–54.3 times. If an attacker knows the peer ID from the amplifier, the attacker is able to predict the BitTorrent client and begin a target-oriented attack. This is possible through the peer ID convention which is defined in BEP 20 [Haro8b]. Even if a client does not support uTP, an attacker can still exploit DHT or MSE with a lower BAF value. This means that nearly every client can be exploited, making such an attack efficient and robust against *peer churn*, independent arrival

and departure of peers. The next section examines the experimental evaluation of these exploits.

7.5 EXPERIMENTAL EVALUATION

In this section, I show that the attacks which I have presented here are efficient, robust, and difficult to circumvent.

7.5.1 *Efficiency*

To test the efficiency of these attacks, I performed an experiments in the testbed system described in Section 4.1 with one attacker, one victim, and 31 amplifiers. All amplifiers were uTorrent server version 3.3 with a private torrent of 1 GiB in seed mode.

The attacker in the first experiment runs a *scapy*[17] script which sends forged uTP packets to the amplifier. The script first sends 31 uTP ST_SYN packets to the amplifier, waits 1 second and then sends 31 uTP data packets which contains the BitTorrent handshake. The script then waits for 120 seconds and repeats this process. It is important to wait attacks, because the amplifiers would otherwise terminate the connection with a ST_FIN packet. I determined the 120 seconds waiting time experimentally. The attacker script does not need to save the state of each peer, making it quite efficient. An I/O graph is shown in Figure 7.5.

Figure 7.5 shows that after the attacker has sent the forged packet, the amplifier sends traffic to the victim. The first peak of the attack is the highest, because both the acknowledgments for the ST_SYN packets and the handshake arrive the victim. The following peaks are the retransmissions of the handshake and the acknowledgments. During this experiment, I noticed that uTorrent for Linux behaves differently than in a Windows client. In Linux, the BAF was lower than with a Windows client. The amplification factor of this attack was 14.6.

17 http://www.secdev.org/projects/scapy/

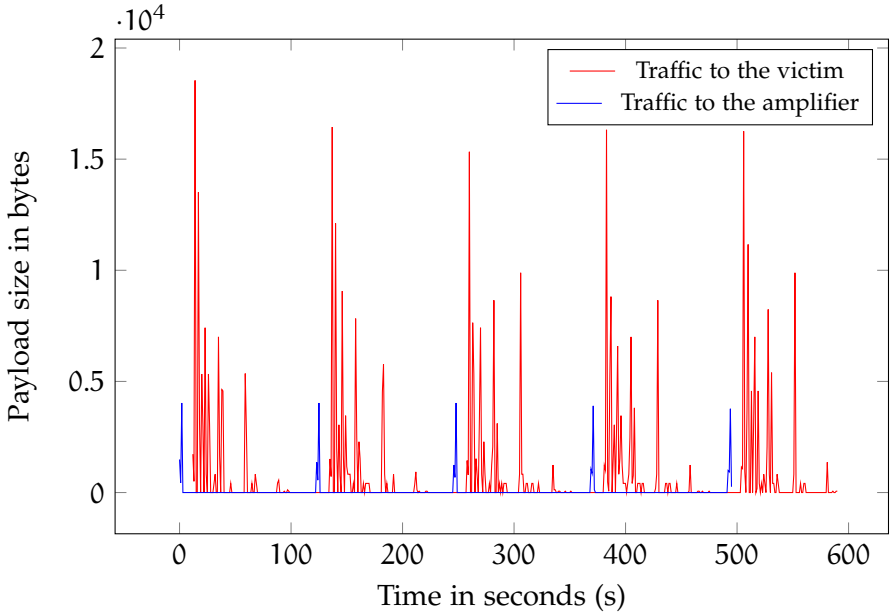


Figure 7.5: Amplification attack with one attacker, 31 amplifiers and one victim. The blue line shows the payload size that the attacker sends to the amplifier. The red line shows the payload size that the amplifier sends to the victim. Source: own representation based on own survey.

7.5.2 *Robustness*

To evaluate the amplifier churn and robustness of the detected vulnerabilities in a real-world attack, I wrote a BitTorrent crawler which consists of two modules. The first module used the MLDHT network to find new peers and the second module established a connection via uTP to the find peers and exchange the BitTorrent handshake message. Overall, I found 9.6 million possible amplifiers when 2.1 million peers responded to the MLDHT requests. The next subsection describes the architecture of the BitTorrent crawler and presents the results from the use of this crawler.

7.5.2.1 *Architecture of a BitTorrent Crawler*

I programmed the BitTorrent crawler in the programming language *Elixir*[18], which is a functional, fault-tolerant and concurrent language

18 http://elixir-lang.org/

built on top of the Erlang Virtual Machine (EVM). Figure 7.6 shows the architecture of this crawler.

Architecture of the BitTorrent Crawler

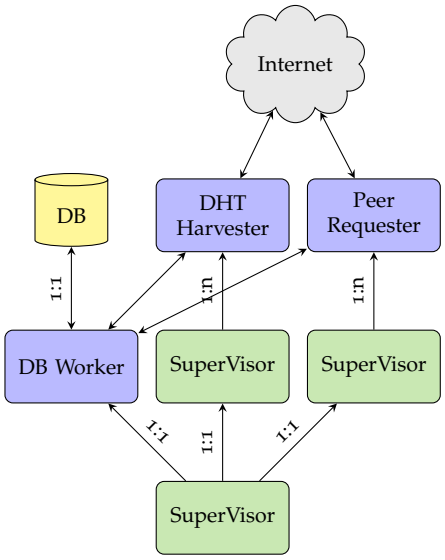


Figure 7.6: Architecture and supervision tree of the BitTorrent Crawler. Every rectangle is a process. DB denotes the database. Source: own representation.

The green rectangles in Figure 7.6 are supervisors. Supervisors are processes that observe other processes. If an observed process crashes unexpectedly, the supervisor automatically restarts the process. The blue rectangles are workers. These perform the actual work. The DB (database) worker is a process which is connected to the study's PostgreSQL database. The supervisors of the DHT harvester and the peer requester start n process each. I set $n = 5$ to completely fill the async-threads of EVM.

Before I began the crawler, I filled a database table 'torrents' with the complete magnet database from *Piratebay* from 13th February 2012. This database comprised 1.6 million unique infohashes. Each MLDHT harvester process selects an un-requested infohash and sends a GET_PEERS request with the infohash to a bootstrapping node. This node returns a list of nodes that are closer to the infohash which the harvester saves in the database. The requester then selects an un-requested node and sends it a GET_PEERS request. Additionally, the harvester process saves meta information from the responses, including payload size, version of the BitTorrent client, and the number

of nodes. The harvester takes another infohash if it has requested 1000 peers.

The peer requester selects peer information from the MLDHT harvester and sends a ST_SYN packet to a peer. If this peer responds within 10 seconds with a ST_STATE packet, the crawler sends a BitTorrent handshake with all extensions enabled to that peer. This process then waits for a reply and saves all responses to the database.

7.5.2.2 *Mainline Distributed Hash Table network*

The BitTorrent crawler collected over 9.6 million peers via MLDHT beginning from January 1st, 2015 and continuing until February 1st, 2015. Of these peers, 2.1 million responded to the GET_PEERS request. This corresponds 21.9 % of peers. Of these responsive peers, 67.8 thousand peers included participating peers, or 3.2 %. The rest returned only k neighbors. Figure 7.7 shows a histogram of payload sizes from MLDHT responses.

Histogram of the Payload Size from the MLDHT Responses

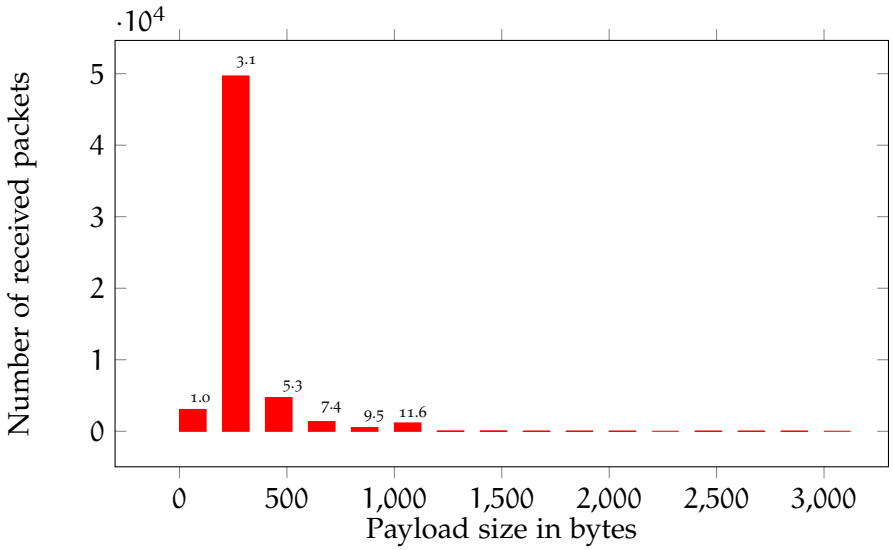


Figure 7.7: Histogram of the payload size from the Distributed Hash Table responses which are caused by GET_PEERS requests. The numbers above the bars are the average Bandwidth Amplification Factor values. Source: own representation based on own survey.

The mean of all received values is 665.6 bytes which results in a BAF of 7. The biggest response had a size of 3344 bytes providing a BAF of 35.2 and the smallest responds had a size of 82 bytes providing a

BAF of 0.9. With such observations, the conclusion can be made, that MLDHT can be exploited but most responses do not produce a high BAF value. Therefore, I used the peer information gathered from the MLDHT network to request a BitTorrent handshake to test the efficacy.

7.5.2.3 *Micro Transport Protocol Distribution*

In this section, I analyze responses from BitTorrent handshake messages. The BitTorrent crawler sent a ST_SYN packet to every participating peer in the database. If a peer responded within 10 seconds, the crawler sent a BitTorrent handshake to this peer. I collected 10,417 handshakes via uTP. I first address the question of the distribution of the payload size of the first data packet because this is crucial for the impact of the amplification attack. Figure 7.8 shows the distribution of the payload size from all BitTorrent handshakes which were received.

Histogram of the BitTorrent Handshake Size from uTP Responses

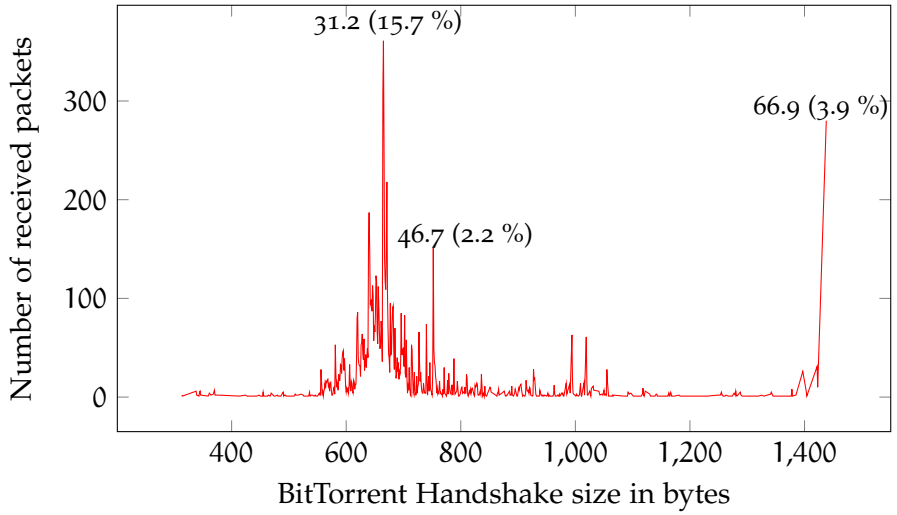


Figure 7.8: Numbers above the peaks are Bandwidth Amplification Factor (BAF) values; the numbers in brackets are the relative frequency. It can be observed that three packet sizes stand out: 665 bytes, 1000 bytes, and 1438 bytes. These three packet sizes would be the most often response in an amplification attack. Source: own representation based on own survey.

The histogram in Figure 7.8 shows three peaks in the distribution. The first peak comprises 15.7 % of all handshakes and would result in a 665 bytes and a 5 times retransmission with a BAF of 31.2. The next

peak is around 1000 bytes with a frequency of 2.2 %. This results in a BAF of 46.7. The third peak has the highest payload size of 1438 bytes and comprises 3.9 % of all handshakes. Retransmitted five times, this would result in a BAF of 66.9.

The mean of all received packets is 810.8 bytes. The maximum value is 1438 bytes and the min value is 313 bytes. All BAF values are calculated with retransmission of 4 packets (total of 5 packets), because this is the default for uTP. I also analyzed all LTEP message that were in the first uTP data packet. A LTEP message contains a dictionary entry with the client version. Table 7.1 evaluates these messages.

VERSION	COUNT	PERCENTAGE
uTorrent 3.4.2	8616	82.7
BitTorrent 7.9.2	1641	15.8
uTorrent 3.4.1	107	1.0
uTorrent 3.4	27	0.3
BitTorrent 7.9.1	14	0.1
Unknown	9	0.1
BitTorrent 7.9	2	0.0

Table 7.1: Client software and version number of the inspected Libtorrent Extension Protocol messages. Source: own representation based on own survey.

It can be seen that the latest uTorrent and BitTorrent version where on top of the list in Table 7.1. Only 0.1 % of the BitTorrent clients were unknown, because they were not transmitting an LTEP message. Because I crawled only the MLDHT network, Vuze is not in the list. In the next subsection, I discuss the difficulty of circumventing a DRDoS attack that exploits BitTorrent.

7.5.3 *Evadability*

The attacker which I discussed in Section 7.1 exploits DNS and Network Time Protocol (NTP) for an amplification attack. Both protocols use well-known ports for their services. Therefore, it is easy to block the reflected traffic with an SPI firewall. BitTorrent, however,

uses dynamic ports which make it harder to block. The payload of the BitTorrent handshake contains the string '*BitTorrent protocol*', and in the case of BTSync the string '*BTSYNC*'. In both cases, the victim needs DPI hardware to block an attack from these protocols.

If an attacker exploits MSE, the victim cannot defend itself neither with an SPI firewall, nor a DPI firewall. Message Stream Encryption does not have a static port number because it is part of BitTorrent and does not have static values in the payload. Table 7.2 compares the defense mechanism of different BitTorrent protocols. In the next section, I discuss how the impacts of the proposed attacks can be reduced or avoided.

	DNS	NTP	BTH	MLDHT	VDHT	BTSYNC	MSE
SPI	X	X					
DPI			X	X	X	X	

Table 7.2: Comparison of amplification vulnerabilities where an X denotes the firewall technology with which it can be defended. Source: own representation based on own survey.

7.6 COUNTERMEASURES AND MITIGATION STRATEGIES

In this section, I discuss countermeasures against the vulnerabilities which have been presented. The root problem of these vulnerabilities is IP source address spoofing. The Spoofer[19] project measures the the susceptibility of IP spoofing around the world from 2005 to the present [Bev+13; Bev+09]. According to the latest measurements from 2015, 26.1 % of all autonomous systems allow spoofing and 15.5 % allow partial spoofing. Anti-spoofing filtering techniques like *Ingress address filtering* and *unicast reverse path* are effective against IP spoofing but are not used everywhere. The lack of these mitigation strategies can also be seen in the impacts of the amplification attacks discussed in Section 7.1. A solution must follow two parallel paths: global ISP coordination to prevent further IP spoofing, and protocol defense and mitigation mechanisms to avoid protocol exploitation.

19 http://spoofer.cmand.org/

7.6.1 *Three-way Handshake over uTP*

The proposed attacks in this study are possible because uTP establish a connection with a two-way handshake. This design failure yields to the discussed vulnerabilities which an attacker can exploit by sending spoofed BitTorrent or uTP handshakes. The ideal countermeasure would be to introduce a three-way handshake to uTP, as in TCP. This would prevent the attacks because the receiver would not accept a data packet without the last acknowledgment from the three-way handshake. The disadvantage of this approach is that it is a significant change in the protocol and would require global coordination with all BitTorrent developers.

7.6.2 *Disable Packet Stuffing for the First Data Packet*

Nearly all BitTorrent client support packet stuffing wherein the clients put BitTorrent messages in the first uTP packet. To mitigate attacks, it is possible to disable packet stuffing and only accept the BitTorrent handshake in the first data packet. This would not prevent the attack but would reduce the impact to a BAF value that is equal to the number of retransmission.

7.6.3 *Enforcing a Valid ACK as a Third Packet*

Another mitigation strategy is to enforce acknowledgment in the first uTP data packet of the first ST_STATE packet. This is similar to a three-way handshake except that the last acknowledgment is also the first data packet. This prevents the attacks for all BitTorrent protocols and does not require a protocol change. Protocols, however, which do not require a first data packet like UCAT, as discussed in Section 7.4.1, would still be vulnerable.

7.6.4 *Distributed Hash Table*

User Datagram Protocol tracker, MLDHT, and VDHT already have countermeasure against index poisoning attacks [NR06]. This prevents an attacker from adding other peers except from themselves to the DHT network with the ANNOUNCE_PEER or STORE call. It does this, by requiring a token that a peer requests from a previous GET_PEERS or FIND_NODE query. This token is valid for 10 minutes and prevents spoofing attacks but only for the ANNOUNCE_PEER or STORE query. To prevent amplification attacks it would be necessary to request the token in the PING query because it has the smallest BAF value. All other queries would need to require this token. This would prevent spoofing attacks against DHT. The disadvantage of this mechanism is that it always requires two DHT queries to request new peers which slows down bootstrapping time.

7.7 COMPARISON WITH RELATED WORK

The ancestor of amplification attacks is the *smurf attack* discussed in [CA98] in 1998. In a smurf attack, attackers send forged Internet Control Message Protocol (ICMP) echo requests to an amplifier using an IP broadcast address. A network packet with a broadcast address in its destination field is addressed to all hosts in a network. For instance, the broadcast address 192.168.1.255 can reach 254 hosts (in Classless Inter-Domain Routing (CIDR) notation 192.168.1.0/24) in the network. All hosts which have received the forged message will send an ICMP echo reply message to the victim. The amplification takes place because the attacker sends a single packet to a host and the whole network replies with a responds. This generates a DDoS attack. As a countermeasure, the amplifier blocks packets that are addressed to an IP broadcast address.

Rossow in [Ros14] provides the first broad investigation of UDP based protocols. The study found 14 protocols that are vulnerable, including MLDHT. I reproduced these results and completed a more thorough investigation of the BitTorrent family which includes uTP,

MSE, BTSync, MLDHT and Vuze DHT with different queries and different clients.

Kührer et al. in [Küh+14a] analyzed the amplifier magnitude for DNS, SNMP, Simple Service Discovery Protocol (SSDP), Character Generator Protocol (CharGen), Quote of the Day (QOTD), NTP and NetBIOS with an Internet-wide scan. Additionally, they developed a remote spoofer test to check if a network allows IP spoofing. They found that more than 2,000 networks that lacked egress filtering. In a follow-up, Kührer et al. in [Küh+14b] investigated the vulnerability of TCP is. Despite its three-way handshake, there are TCP/IP implementations that do not strictly follow the standard practice in which the implementation sends data before the three-way handshake is complete.

7.8 SUMMARY

In this chapter, I have shown that BitTorrent and BTSync are vulnerable to DRDoS attacks. With peer-discovery techniques like trackers, DHT or PEX, an attacker can collect millions of amplifiers. An attacker needs only a valid infohash or secret, which can be easily obtained by torrent search engines to exploit vulnerabilities. I have shown that the most used BitTorrent clients, uTorrent, Mainline and Vuze, are highly vulnerable and the traffic of these clients can be amplified by up to a factor of 50. In the case of BTSync, amplification by a factors of up to 120 is possible with only a single packet. An easier amplification target is a MSE handshake because the attacker does not require a valid infohash. The BAF value of this attack ranges from 4–32.5. Such attacks are not only easy for an attacker, they are also hard to detect because the payload of the handshake has a high entropy value.

I followed a responsible disclosure process and contacted the security team at BitTorrent Inc. one month before a research paper about this chapter was published. They decided to enforce a valid ACK as third packet as countermeasure against the presented attacks [De 15; Ave15]. They noted that they also want to include a countermeasure for MLDHT, however, at the time of writing this thesis, this counter-

measure is still in progress. A couple of news sites covered the presented vulnerabilities [Van15b; Goo15].

Part III

CONCLUSION

"Education has nothing to do with filling a pail,
rather it has everything to do with igniting a flame"

—Heraclitus

8

CONCLUSION

8.1 RESEARCH HYPOTHESIS AND OBJECTIVES

This thesis presents an analysis of bandwidth attacks for the incentive driven P2P protocol BitTorrent. In this section, I reflect on both my research hypothesis and objectives.

8.1.1 *Research Hypothesis*

In the introduction, Section 1.3, I presented the hypothesis that securing the bandwidth consumption of P2P applications improves the security of the P2P swarm. I divided this hypotheses into two sub-hypotheses. I will evaluate the sub-hypotheses in the next paragraphs.

The first sub-hypotheses 1.2 stats that *P2P protocols that do not have bandwidth security policies pose a security threat to both the swarm and the Internet*. All attacks presented in this thesis can create tangible damage. The damage begins with a significantly slower download speed for participating peers, the opposite of the intentions of the BitTorrent protocol. This damage arises if malicious peers exploit vulnerabilities on an application level. The damage continues with possible congestion in a specific network path which results in a DoS attack in which other peers cannot reach the victim (seeder or leecher). Finally, damage occurs because an attacker can redirect bandwidth from any BitTorrent client that is using uTP to an arbitrary victim (DRDoS). This shows that BitTorrent pose a security threat to the swarm and to the Internet.

The second sub-hypotheses 1.1 states that one of *the most used P2P protocols BitTorrent has not secured bandwidth enough*. I found vulnerabilities on both an application and transport level. In terms of the

application level, I showed that the default seeding algorithm FU is easily exploitable by malicious peers. The allowed fast extension allows malicious peers to request pieces repeatedly and therefore steal bandwidth. For these attacks, I presented a new seeding algorithm PI for the allowed fast extension, a specification change is needed. In terms of transport, however, I have shown that malicious peers can exploit uTP and its congestion control algorithm LEDBAT to create congestion on a specific network path. The countermeasure against uTP and LEDBAT attacks requires the occasional dropping of a random packet to identify malicious peers. Because uTP uses a two-way handshake, I have shown in Section 7 that a malicious peer can redirect traffic from a BitTorrent client to an arbitrary victim. These vulnerabilities strongly suggest that BitTorrent has not done enough to secure bandwidth. The two sub-hypothesis together build the main hypothesis of this thesis: *Securing the bandwidth consumption of P2P protocols improves the security of the P2P swarm.*

8.1.2 *Thesis Objectives*

In the introduction, Section 1.3, I presented three objectives of this thesis. The first objective was to investigate the impact of bandwidth attacks against application and transport layer of the P2P protocol BitTorrent. I investigated bandwidth attacks against application and transport layer in a small testbed system with 32 peers and a global testbed system with 300 peers and presented my results in Section 5.4.

The second objective was to understand the aspects of bandwidth attacks related to security in a BitTorrent swarm. I showed in Section 5.4, that the chocking algorithm is mainly responsible of how vulnerable BitTorrent is to bandwidth attacks.

The last objective was to propose systematic improvements to the found vulnerabilities. I proposed effective countermeasures which mitigate or prevent the attacks to all found vulnerabilities. Therefore, all objectives were archived. I conclude this section with a summary of the achievements of this thesis and recommendations for future research.

8.2 SUMMARY OF THE ACHIEVEMENTS

I began this analysis with an investigation of the BitTorrent protocol itself, in particular different choking algorithms in the seed state and the allowed fast extension. I evaluated seeding algorithms experimentally in terms of performance, security and stability, and found out that the default seeding algorithm FU and AL are highly vulnerable to bandwidth attacks. During this investigation, I found a programming error in the RR implementation of *libtorrent* which could be exploited by malicious peers. Together with the maintainer of *libtorrent* I fixed the programming error in RR. I also found that RR and LW are harder to exploit, but still exploitable with a sybil attack. These results motivated me to propose a novel seeding algorithm PI that is more robust against bandwidth attacks without the loss of performance. In the case of the allowed fast extension, I found that it has a logical issue, because an attacker can request the allowed fast set repeatedly. I proposed two countermeasures against this attack, one of which was included in *libtorrent*. From this point, I continued analysis on the transport protocol.

I investigated BitTorrent's new transport protocol uTP and its congestion control algorithm LEDBAT. Congestion is an important issue, especially for transferring bulk data as with the BitTorrent protocol. I proposed three attacks against uTP, namely, delay attack, lazy opt-ack attack and opt-ack attack and evaluated their impact in terms of bandwidth consumption and number of packets. I have shown that an attacker that is using a delay attack can steal additional bandwidth and therefore has an advantage because of the reciprocal nature of BitTorrent. An attacker, however, that is not interested in data integrity can create significant congestion with a lazy opt-ack and an opt-ack attack. I implemented a countermeasure which drops packets randomly to check if the receiver is lying in the acknowledgment. I continued in my analysis of the uTP with a detailed examination of the two-way handshake.

Finally, I investigated if and how the BitTorrent protocol family is vulnerable to DRDoS attacks. In comparison to TCP, uTP uses a two-

way handshake instead of a traditional three-way handshake. An attacker that makes use of IP spoofing can exploit this handshake to inject a first data packet into any BitTorrent peer. An attacker can craft a special BitTorrent handshake to redirect traffic from an amplifier to any victim on the Internet. I have shown in the case of BitTorrent that an attacker can amplify the traffic by up to a factor of 50. With the extension MSE, an attacker does not need a valid infohash and can amplify the traffic by up a factor of 4–32.5. I have shown that even the P2P file synchronization protocol BTSync which also uses uTP as its main transport protocol is vulnerable to these attacks. An attacker can amplify BTSync traffic up to 120 times. I proposed countermeasures which are included in all BitTorrent clients that were vulnerable to our attacks.

8.3 FUTURE DIRECTIONS

This thesis presented an analysis of bandwidth attacks in the mainline BitTorrent protocol. There are, however, a variety of extensions that may be exploitable but were not covered in this thesis, including PEX [The15] an extension for peers to send metadata files [HN12]. Both extensions provide the possibility for additional information via messages. There are also extensions in uTorrent that are undisclosed as of yet, including UT_HOLEPUNCH and UT_COMMENT. The BitTorrent client *Vuze* also has its own extensions. Because all of these extensions are widely used, a security analysis would be an important contribution.

This thesis also presented a novel seeding algorithm PI as a countermeasure against bandwidth attacks. There is a proof-of-concept called *joystream*[1] that combines BitTorrent with the P2P crypto currency Bitcoin. It would be relevant to examine how well BitCoin could work as an incentive mechanism in a situation wherein a user could receive bitcoins by being a seeder and provide bitcoins to other peers to download files.

1 http://joystream.co/

Part IV

APPENDIX

A

APPENDIX

CATEGORY	DESCRIPTION	BAF	PAF
	ucat	351.5	6
BitTorrent	uTorrent w/o extensions	27.6	3.5
	Mainline w/o extensions	27.8	3.5
	uTorrent with LTEP	39.6	3
	Mainline with LTEP	39.6	3
	Vuze w/o extensions	13.9	2
	Vuze with LTEP	18.7	2
	Vuze with AMP	54.3	3.5
	Transmission w/o extensions	4.0	3.5
	Transmission with LTEP	4.0	3.5
	Transmission with AMP	4.0	3.5
	Libtorrent w/o extensions	5.2	4
	Libtorrent with LTEP	5.2	4
MLDHT	ping	0.8	1
	find_node with K = 8	3.1	1
	`get_peers` with 100 peers (IPv4)	11.9	1
	`get_peers` with 100 peers (IPv6)	24.5	1
	`get_peers` with scrapes	13.4	1
VDHT	ping	0.8	1
	ping with Vivaldi coordinates	14.9	1
BTSync	ping	120.2	129

Table A.1: Summary of all BAF and PAF values of the inspected BitTorrent messages. Source: own representation based on own survey.

Listing A.1: Source code of PI implemented as a libtorrent extension.

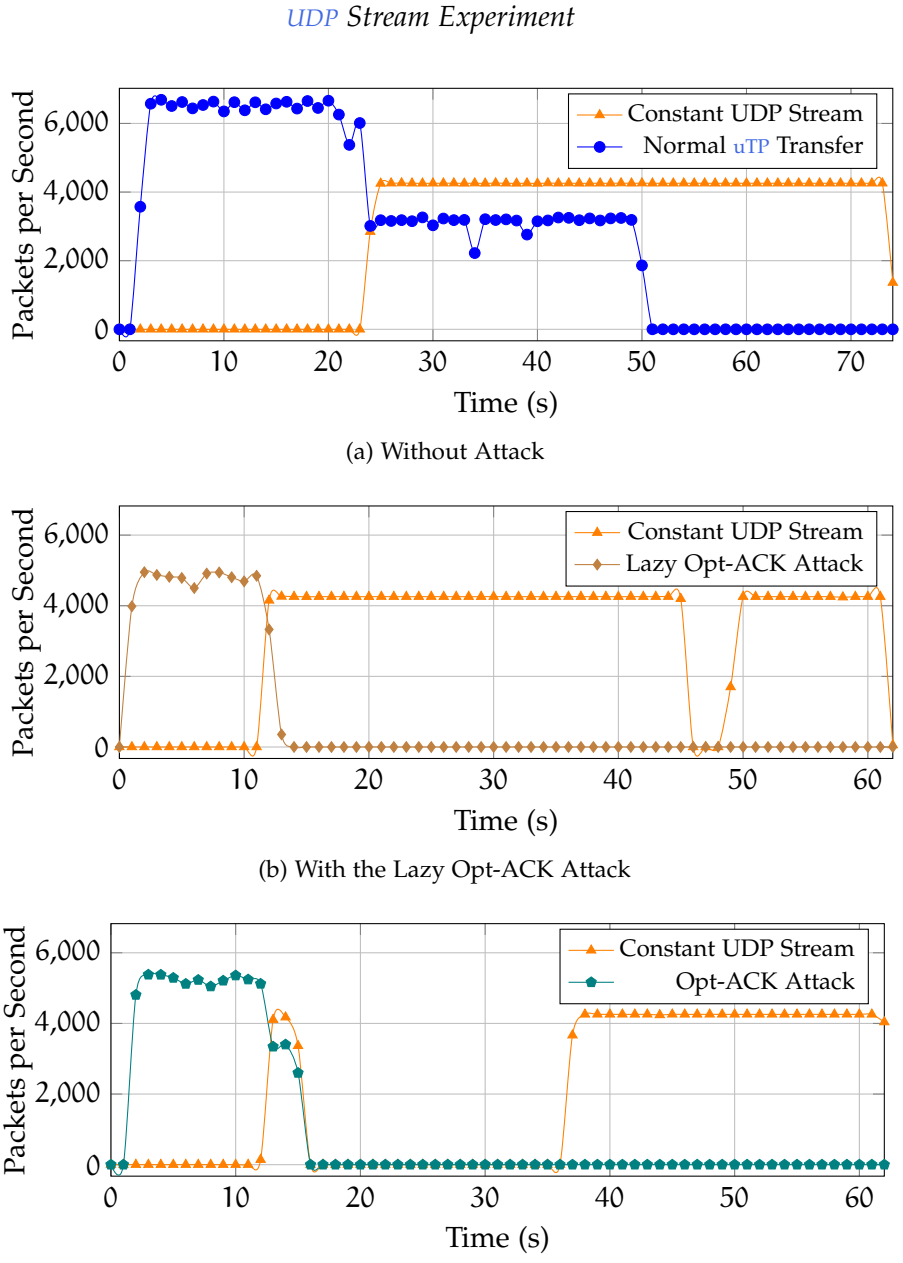
```
1 #include "libtorrent/pch.hpp"
2
3 #ifndef TORRENT_DISABLE_EXTENSIONS
4
5 #ifdef _MSC_VER
6 #pragma warning(push, 1)
7 #endif
```

```
8
9  #include <boost/shared_ptr.hpp>
10 #include <boost/bind.hpp>
11
12 #ifdef _MSC_VER
13 #pragma warning(pop)
14 #endif
15
16 #include "libtorrent/peer_connection.hpp"
17 #include "libtorrent/bt_peer_connection.hpp"
18 #include "libtorrent/bencode.hpp"
19 #include "libtorrent/torrent.hpp"
20 #include "libtorrent/extensions.hpp"
21 #include "libtorrent/broadcast_socket.hpp"
22 #include "libtorrent/socket_io.hpp"
23 #include "libtorrent/peer_info.hpp"
24 #include "libtorrent/random.hpp"
25
26 #include "libtorrent/extensions/peer_idol.hpp"
27
28 #ifdef TORRENT_VERBOSE_LOGGING
29 #include "libtorrent/lazy_entry.hpp"
30 #endif
31
32 namespace libtorrent {
33
34     class torrent;
35
36     namespace {
37         const char extension_name[] = "peer_idol";
38
39         struct peer_idol_plugin: torrent_plugin {
40
41             torrent& m_torrent;
42
43             peer_idol_plugin(torrent& t) : m_torrent(t) {}
44
45             ~peer_idol_plugin() {}
46
47             virtual boost::shared_ptr<peer_plugin> new_connection
                    (peer_connection* pc);
48         };
49
50
51         struct peer_idol_peer_plugin : peer_plugin {
52             peer_idol_peer_plugin(torrent& t, bt_peer_connection&
                     pc)
53                 : m_peer_idol_extension_id(23)
54                 , m_torrent(t)
55                 , m_pc(pc)
56                 , m_10_second(0)
57                 , m_full_list(true) {
58             }
59
60             virtual void tick() {
```

```
61                  if (++m_10_second <= 10)
62                      return;
63
64                  // only send the votes to the seeder
65                  if (m_pc.is_seed()) {
66                      send_best_peers();
67                  }
68
69                  m_10_second = 0;
70              }
71
72              // can add entries to the extension handshake
73              virtual void add_handshake(entry& h) {}
74
75              void send_best_peers() {
76                  // copy peers from the torrrent in a
                        peer_connection
77                  // vector in order to sort it in the next step
78                  std::vector<peer_connection*> peers;
79                  for (torrent::peer_iterator i = m_torrent.begin()
                        ,
80                       end(m_torrent.end()); i != end; ++i) {
81                      peer_connection* peer = *i;
82                      peers.push_back(peer);
83                  }
84
85                  // sort all peers according to their upload rate
86                  std::sort(peers.begin(), peers.end()
87                            , boost::bind(&peer_connection::
                                payload_download_compare, _1, _2));
88
89                  entry pid;
90                  std::string& pla = pid["added"].string();
91                  std::back_insert_iterator<std::string> pla_out(
                        pla);
92
93                  // check if enough peers are available
94                  if (peers.size() >= 3) {
95
96                      for (int i = 0; i < 3; ++i) {
97                          tcp::endpoint  remote = peers.at(i)->
                                remote();
98
99                          if (remote.address().is_v4()) {
100                             detail::write_endpoint(remote,
                                    pla_out);
101                         }
102
103                     }
104
105                 }
106
107                 std::vector<char> pid_msg;
108                 bencode(std::back_inserter(pid_msg), pid);
109
```

```
110                 char msg[6];
111                 char* ptr = msg;
112 
113                 detail::write_uint32(1 + 1 + pid_msg.size(), ptr)
                        ;
114                 detail::write_uint8(bt_peer_connection::
                        msg_extended, ptr);
115                 detail::write_uint8(m_peer_idol_extension_id, ptr
                        );
116 
117                 m_pc.send_buffer(msg, sizeof(msg));
118                 m_pc.send_buffer(&pid_msg[0], pid_msg.size());
119 
120             }
121 
122             virtual bool on_extended(int length, int msg, buffer
                    ::const_interval body) {
123                 if (msg != m_peer_idol_extension_id) return false
                        ;
124 
125                 lazy_entry pid_msg;
126                 error_code ec;
127                 int ret = lazy_bdecode(body.begin, body.end,
                        pid_msg, ec);
128                 if (ret != 0 || pid_msg.type() != lazy_entry::
                        dict_t) {
129                     return true;
130                 }
131 
132                 lazy_entry const* p = pid_msg.dict_find_string("
                        added");
133                 char const* in = p->string_ptr();
134 
135                 for (int i = 0; i < 3; ++i) {
136                     tcp::endpoint adr = detail::read_v4_endpoint<
                            tcp::endpoint>(in);
137 
138                     for (torrent::peer_iterator i = m_torrent.
                            begin(),
139                          end(m_torrent.end()); i != end; ++i)
                                 {
140                         peer_connection* peer = *i;
141                         if (peer->remote() == adr) {
142                             peer->votes++;
143                         }
144                     }
145 
146                 }
147             }
148 
149             int m_peer_idol_extension_id;
150 
151             typedef std::vector<std::pair<address_v4::bytes_type,
                     boost::uint16_t> > peers4_t;
152             peers4_t m_peers;
```

```
153
154             torrent& m_torrent;
155             bt_peer_connection& m_pc;
156
157             int m_10_second;
158             bool m_full_list;
159         };
160
161         boost::shared_ptr<peer_plugin> peer_idol_plugin::
                new_connection(peer_connection* pc)
162         {
163             if (pc->type() != peer_connection::
                    bittorrent_connection)
164                 return boost::shared_ptr<peer_plugin>();
165
166             bt_peer_connection* c = static_cast<
                    bt_peer_connection*>(pc);
167             return boost::shared_ptr<peer_plugin>(new
                    peer_idol_peer_plugin(m_torrent, *c));
168         }
169
170     }
171 }
172
173 namespace libtorrent
174 {
175     boost::shared_ptr<torrent_plugin> create_peer_idol_plugin(
            torrent* t, void*)
176     {
177
178         return boost::shared_ptr<torrent_plugin>(new
                peer_idol_plugin(*t));
179     }
180 }
181
182
183 #endif
```



(a) Without Attack

(b) With the Lazy Opt-ACK Attack

(c) With the Opt-ACK Attack

Figure A.1: File transfer of a 300 MiB File via uTP and parallel to that, a constant UDP stream of 50 Mbps under the following network conditions: Bandwidth: 100 Mbps half duplex and Delay: 0 ms. Source: own representation based on own survey.

BIBLIOGRAPHY

[10a] *Azureus messaging protocol - VuzeWiki*. Oct. 2010. URL: https://wiki.vuze.com/w/Azureus_messaging_protocol (visited on 05/02/2016).

[10b] *BEP 33: DHT Scrapes*. Tech. rep. BitTorrent Inc., Jan. 2010. URL: http://www.bittorrent.org/beps/bep_0033.html (visited on 05/02/2016).

[11] *Enemies of the Internet*. Tech. rep. Reporters without Borders, Mar. 2011. URL: http://viewsdesk.com/wp-content/uploads/2011/03/Internet-Enemies_2011.pdf (visited on 05/02/2016).

[12] *Distributed Hash Table*. Tech. rep. Vuze, Oct. 2012. URL: https://wiki.vuze.com/w/Distributed_hash_table (visited on 05/02/2016).

[13] *Quarterly Global DDoS Attack Report Q1 2013*. Tech. rep. Prolexic, 2013. URL: http://cybersafetyunit.com/download/pdf/DDoS_Attack_Report_Q113_041613.pdf (visited on 05/02/2016).

[14] *Message Stream Encryption Format Specification Version 1.0*. en. May 2014. URL: http://wiki.vuze.com/w/Message_Stream_Encryption (visited on 05/02/2016).

[AG11] Amuda James Abu and Steven Gordon. 'Impact of delay variability on LEDBAT performance.' In: *Advanced Information Networking and Applications*. IEEE, 2011, pp. 708–715. DOI: 10.1109/AINA.2011.98.

[AH00] Eytan Adar and Bernardo A. Huberman. 'Free Riding on Gnutella.' In: *First Monday* 05.10 (2000). DOI: 10.5210/fm.v5i10.792.

[Ave15] Christian Averill. *Mitigating DRDoS Vulnerability in the BitTorrent Ecosystem*. Aug. 2015. URL: http://blog.bittorrent.com/2015/08/27/mitigating-drdos-vulnerability-in-the-bittorrent-ecosystem/ (visited on 05/02/2016).

[BA05] Nabhendra Bisnik and Aluhussein Abouzeid. 'Modeling and Analysis of Random Walk Search Algorithms in P2P Networks.' In: *Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems*. IEEE, 2005.

[Bev+09] Robert Beverly, Arthur Berger, Young Hyun, and K Claffy. 'Understanding the Efficacy of Deployed Internet Source Address Validation Filtering.' In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. 2009, pp. 356–369. DOI: 10.1145/1644893.1644936.

[Bev+13] Robert Beverly, Arthur Berger, Young Hyun, and K Claffy. *Initial Longitudinal Analysis of IP Source Spoofing Capability on the Internet*. Tech. rep. Internet Society, 2013. URL: http://www.internetsociety.org/doc/initial-longitudinal-analysis-ip-source-spoofing-capability-internet (visited on 05/02/2016).

[BMB08] M. P. Barcellos, R. B. Mansilha, and F. V. Brasileiro. 'TorrentLab: investigating BitTorrent through simulation and live experiments.' In: *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*. July 2008, pp. 507–512. DOI: 10.1109/ISCC.2008.4625749.

[Bra97] Scott Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. Tech. rep. 2119. Mar. 1997. URL: htt

p://www.ietf.org/rfc/rfc2119.txt (visited on 05/02/2016).

[BV08] Billy Bob Brumley and Jukka Valkonen. 'Attacks on Message Stream Encryption.' In: *Proceedings of the 13th Nordic Workshop on Secure IT Systems*. Oct. 2008, pp. 163–173.

[BYL09] John F. Buford, Heather Yu, and Eng Keong Lua. *P2P Networking and Applications*. Academic Press, 2009. ISBN: 978-0123742148.

[CA98] *CA-1998-01: Smurf IP Denial-of-Service Attacks*. CERT. Oct. 1998. URL: https://www.cert.org/historical/advisories/CA-1998-01.cfm (visited on 05/02/2016).

[Cam09] Gonzalo Camarillo. *Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability*. Tech. rep. 5694. Nov. 2009. URL: http://www.ietf.org/rfc/rfc5694.txt (visited on 05/02/2016).

[Cas+02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. 'Secure routing for structured peer-to-peer overlay networks.' In: *ACM SIGOPS Operating Systems Review* 36 (Dec. 2002), pp. 299–314. ISSN: 01635980. DOI: 10.1145/844128.844156.

[CC05] Tom Chothia and Konstantinos Chatzikokolakis. 'A Survey of Anonymous Peer-to-Peer File-Sharing.' In: *Proceedings of Embedded and Ubiquitous Computing – EUC 2005 Workshops*. Springer Berlin Heidelberg, 2005, pp. 744–755. DOI: 10.1007/11596042_77.

[CGM08] Alix L. H. Chow, Leana Golubchik, and Vishal Misra. 'Improving BitTorrent: a simple approach.' In: *Proceedings of the 7th international conference on Peer-to-peer systems*. IPTPS'08. 2008, p. 8.

[Cha+09] Olivier Chalouhi, Alon Rohter, Paul Gardner, and AronM. *Changelog: Azureus: Vuze 4.3.0.0 Released*. Nov. 2009. URL: http://sourceforge.net/p/azureus/news/2009/11/azureus-vuze-4300-released-/ (visited on 05/02/2016).

[Cha81] David Chaum. 'Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms.' In: *Communications of the ACM* 24 (Feb. 1981), pp. 84–90. DOI: 10.1145/358549.358563.

[Che+08] Zhijia Chen, Yang Chen, Chuang Lin, Nivargi V., and Pei Cao. 'Experimental Analysis of Super-Seeding in BitTorrent.' In: *Communications, 2008. ICC '08. IEEE International Conference on*. May 2008, pp. 65–69.

[Cho+12] Tom Chothia, Marco Cova, Chris Novakovic, and Camilo Gonzlez Toro. 'The Unbearable Lightness of Monitoring: Direct Monitoring in BitTorrent.' In: *8th International Conference on Security and Privacy in Communication Networks*. SecureComm, 2012. DOI: 10.1007/978-3-642-36883-7_12.

[Chu+04] Yang-hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. 'Early Experience with an Internet Broadcast System Based on Overlay Multicast.' In: *Proceedings of the Usenix Technical Conference*. USENIX, 2004, pp. 1–15.

[CJ89] Dah-Ming Chiu and Raj Jain. 'Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks.' In: *Computer Networks and ISDN Systems* 17.1 (June 1989), pp. 1–14. DOI: 10.1016/0169-7552(89)90019-6.

[Cla+01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. 'Freenet: A Distributed Anonymous Information Storage and Retrieval System.' In: *International Workshop On Designing Privacy Enhancing Technologies: Design Issues In Anonymity And Unobservability*. Springer-Verlag New York, Inc., 2001, pp. 46–66.

[Cli00] Clip2.com, Inc. *Gnutella: To the Bandwidth Barrier and Beyond*. 2000. URL: http://www.xml.com/pub/r/662 (visited on 05/02/2016).

[Coh03] Bram Cohen. 'Incentives build robustness in BitTorrent.' In: *Workshop on Economics of Peer-to-Peer systems*. 2003. URL: http://bittorrent.org/bittorrentecon.pdf (visited on 05/02/2016).

[Coh08a] Bram Cohen. *Discussion of BEP 6: Fast Extension*. 2008. URL: https://web.archive.org/web/20080829225534/http://forum.bittorrent.org/viewtopic.php?pid=384 (visited on 05/02/2016).

[Coh08b] Bram Cohen. *The Bittorrent Protocol Specification*. Tech. rep. Bittorrent, Inc., Feb. 2008. URL: http://bittorrent.org/beps/bep_0003.html (visited on 05/02/2016).

[Cot13] Roger Les Cottrel. *How Bad Is Africa's Internet?* Jan. 2013. URL: http://spectrum.ieee.org/telecom/internet/how-bad-is-africas-internet (visited on 05/02/2016).

[Cue+10] Ruben Cuevas, Michal Kryczka, Angel Cuevas, Sebastian Kaune, Carmen Guerrero, and Reza Rejaie. 'Is content publishing in BitTorrent altruistic or profit-driven?' In: *Proceedings of the 6th International Conference*. ACM, 2010, 11:1–11:12. DOI: 10.1145/1921168.1921183.

[Dab+04] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. 'Vivaldi: a decentralized network coordinate system.' In: *Proceedings of the 2004 Conference on Applications,*

Technologies, Architectures, and Protocols for Computer Communications. ACM Press, 2004, pp. 15–26. DOI: 10.1145/1015467.1015471.

[De 15] Francisco De La Cruz. *DRDoS, UDP-Based protocols and BitTorrent*. Aug. 2015. URL: http://engineering.bittorrent.com/2015/08/27/drdos-udp-based-protocols-and-bittorrent/ (visited on 05/02/2016).

[Dhu+08a] Prithula Dhungel, Xiaojun Hei, Di Wu, and K.W. Ross. *The seed attack: Can bittorrent be nipped in the bud?* Tech. rep. Department of Computer and Information Science, Polytechnic Institute of NYU, 2008, pp. 1–20.

[Dhu+08b] Prithula Dhungel, Di Wu, Xiaojun Hei, Brad Schonhorst, and Keith W. Ross. 'Is BitTorrent Unstoppable?' In: *International Workshop on Peer-to-Peer Systems (IPTPS)* (2008). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.414.971&rep=rep1&type=pdf (visited on 05/02/2016).

[Dhu+08c] Prithula Dhungel, Di Wu, Brad Schonhorst, and Keith W. Ross. 'A measurement study of attacks on bittorrent leechers.' In: *International Workshop on Peer-to-Peer Systems (IPTPS)*. Citeseer, Feb. 2008.

[DHW11] Prithula Dhungel, Xiaojun Hei, and Di Wu. 'A measurement study of attacks on bittorrent seeds.' In: *(ICC), 2011 IEEE*. IEEE, June 2011, pp. 1–5. ISBN: 978-1-61284-232-5. DOI: 10.1109/icc.2011.5963011.

[Dic13] Dictionary.com, ed. *Collins English Dictionary - Complete & Unabridged 10th Edition*. Feb. 2013. URL: http://dictionary.reference.com/browse/peer (visited on 05/02/2016).

[Dis+11] Marcel Dischinger, Massimiliano Marcon, Saikant Guha, Krishna P. Gummadi, Ratul Mahajan, and Stefan Saroiu.

'Glasnost: Enabling End Users to Detect Traffic Differentiation.' In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*. 2011.

[Dou02] John R. Douceur. 'The Sybil Attack.' In: *IPTPS*. 2002, pp. 251–260. DOI: 10.1007/3-540-45748-8_24.

[DWR09] Prithula Dhungel, Di Wu, and Keith W. Ross. 'Measurement and mitigation of BitTorrent leecher attacks.' In: *Elsevier Computer Communications* 32.17 (2009), pp. 1852–1861. DOI: 10.1016/j.comcom.2009.07.006.

[EGM] Karim El Defrawy, Minas Gjoka, and Athina Markopoulou. 'BotTorrent: Misusing BitTorrent to Launch DDoS Attacks.' In: *Proceedings of the 3rd USENIX Workshop on Steps to reducing unwanted traffic on the Internet*.

[FS12] Kevin Fall and William Richard Stevens. *TCP/IP Illustrated, Volume 1 Second Edition. The Protocols*. Addison-Wesley, 2012. ISBN: 978-0-321-33631-6.

[Goo15] Dan Goodin. *How BitTorrent could let lone DDoS attackers bring down big sites*. Aug. 2015. URL: http://arstechnica.com/security/2015/08/how-bittorrent-could-let-lone-ddos-attackers-bring-down-big-sites/ (visited on 05/02/2016).

[Gre02] Matthew Green. 'Napster Opens Pandora's Box: Examining How File-Sharing Services Threaten the Enforcement of Copyright on the Internet.' In: *Ohio State Law Journal* 63.2 (2002), pp. 799–819. URL: http://moritzlaw.osu.edu/students/groups/oslj/files/2012/03/63.2.green_.pdf (visited on 05/02/2016).

[GRS99] David Goldschlag, Michael Reed, and Paul Syverson. 'Onion Routing for Anonymous and Private Internet Connections.' In: *Communications of the ACM* 42 (1999), pp. 39–41.

[Haro8a] Dave Harrison. *Discussion of BEP 6: Fast Extension*. 2008. URL: https://web.archive.org/web/20080927063343/http://forum.bittorrent.org/viewtopic.php?id=13 (visited on 05/02/2016).

[Haro8b] David Harrison. *BEP 20: Peer ID Conventions*. Tech. rep. BitTorrent Inc., Feb. 2008. URL: http://bittorrent.org/beps/bep_0020.html (visited on 05/02/2016).

[Hazo8] Greg Hazel. *Announcements: uTorrent 1.9 alpha 15380*. Dec. 2008. URL: https://web.archive.org/web/20081206022119/http://forum.utorrent.com/viewtopic.php?pid=377209 (visited on 05/02/2016).

[HCo8] David Harrison and Bram Cohen. *BEP 0006: Fast Extension*. Tech. rep. BitTorrent, Inc., 2008. URL: http://bittorrent.org/beps/bep_0006.html (visited on 05/02/2016).

[Hei+15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. 'Eclipse Attacks on Bitcoin's Peer-to-Peer Network.' In: *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 129–144. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman (visited on 05/02/2016).

[Hemo5] Stephen Hemminger. 'Network emulation with NetEm.' In: *Linux Conf Au*. Citeseer, 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.1687&rep=rep1&type=pdf (visited on 05/02/2016).

[Hemo7] Stephen Hemminger. *Re: fixing opt-ack DoS against TCP-Stack*. Linux-Net Archive. Jan. 2007. URL: http://lkml.indiana.edu/hypermail/linux/net/0701.1/0003.html (visited on 05/02/2016).

[HJ09] Eric Hjelmvik and Wolfgang John. 'Statistical protocol identification with SPID: Preliminary results.' In: *Proceedings of the Swedish National Computer Networking Workshop*. 2009.

[HKZ07] Jerome Harrington, Corey Kuwanoe, and Cliff C. Zou. 'A BitTorrent-driven distributed.' In: *Proceedings of the 3th International Conference on Security and Privacy in Communications Networks*. IEEE, 2007, pp. 261–268. DOI: 10.1109/SECCOM.2007.4550342.

[HN12] Greg Hazel and Arvid Norberg. *BEP 0009: Extension for Peers to Send Metadata Files*. Tech. rep. BitTorrent Inc., Oct. 2012. URL: http://www.bittorrent.org/beps/bep_0009.html (visited on 05/02/2016).

[Hof08a] John Hoffman. *BEP 0016: Superseeding*. Tech. rep. BitTorrent Inc., Feb. 2008. URL: http://www.bittorrent.org/beps/bep_0016.html (visited on 05/02/2016).

[Hof08b] John Hoffman. *BEP 12: Multitracker Metadata Extension*. Tech. rep. Bittorrent, Inc., Feb. 2008. URL: http://bittorrent.org/beps/bep_0012.html (visited on 05/02/2016).

[Hon13] Lou Hong. *Introducing BitTorrent Sync 1.2*. Nov. 2013. URL: http://blog.bittorrent.com/2013/11/05/introducing-bittorrent-sync-1-2/ (visited on 05/02/2016).

[Isd+10] Tomas Isdal, Michael Piatek, Arvind Krishnamurthy, and Thomas Anderson. 'Privacy-Preserving P2P Data Sharing with OneSwarm.' In: *Proceedings of the ACM SIGCOMM 2010 conference*. 2010. DOI: 10.1145/1851182.1851198.

[JW12] Roger G. Johnston and Jon S. Warner. *The Dr. Who Conundrum*. Security Management. 2012. URL: https://sm.asisonline.org/Pages/The-Dr-Who-Conundrum.aspx (visited on 05/02/2016).

[Kano1] Gene Kan. *Gnutella*. Ed. by Andy Oram. O'Reilly & Associates Incorporation, 2001, pp. 94–122. ISBN: 0-59600110-X.

[KM02] Tor Klingberg and Raphael Manfredi. *Gnutella 0.6 Protocol Specification*. Tech. rep. Network Working Group, 2002. URL: http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html (visited on 05/02/2016).

[Knu74] Donald E. Knuth. 'Computer Programming as an Art.' In: *Communications of the ACM* 17.12 (Dec. 1974), pp. 667–673.

[Köh+10] Christopher Köhnen, Christian Überall, Florian Adamsky, Veselin Rakocevic, Muttukrishnan Rajarajan, and Rudolf Jäger. 'Enhancements to Statistical Protocol IDentification (SPID) for Self-Organised QoS in LANs.' In: *Proceedings of the 19th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, Aug. 2010, pp. 1–6. ISBN: 978-1-4244-7114-0. DOI: 10.1109/ICCCN.2010.5560139.

[Kry+11] Michal Kryczka, Rubén Cuevas Rumın, Roberto Gonzalez, Ángel Cuevas, and Arturo Azcorra. 'FakeDetector: A measurement-based tool to get rid out of fake content in your BitTorrent Downloads.' In: *CoRR* abs/1105.3671 (2011). URL: http://arxiv.org/abs/1105.3671 (visited on 05/02/2016).

[Küh+14a] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. 'Exit from Hell? Reducing the Impact of Amplification DDoS Attacks.' In: *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*. Aug. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kuhrer (visited on 05/02/2016).

[Küh+14b] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. 'Hell of a Handshake: Abusing TCP for Reflective Amplification DDoS Attacks.' In: *Proceedings of the 8th USENIX Workshop on Offensive Technologies*. USENIX Association, Aug. 2014.

[Le +10] Stevens Le Blond, Arnaud Legout, Fabrice Lefessant, Walid Dabbous, and Mohamed Ali Kaafar. 'Spying the world from your laptop: identifying and profiling content providers and big downloaders in BitTorrent.' In: *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*. 2010. URL: https://www.usenix.org/legacy/event/leet10/tech/full_papers/LeBlond.pdf (visited on 05/02/2016).

[Leg+07] Arnaud Legout, Nikitas Liogkas, Eddie Kohler, and Lixia Zhang. 'Clustering and sharing incentives in BitTorrent systems.' In: *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 2007, pp. 301–312.

[Len11] Lenovo. *IdeaCentre Q180 (3110) data sheet*. 2011. URL: https://www.lenovo.com/shop/americas/content/pdf/system_data/q180_tech_specs.pdf (visited on 05/02/2016).

[Lio+06] Nikitas Liogkas, Robert Nelson, Eddie Kohler, and Lixia Zhang. 'Exploiting BitTorrent For Fun (But Not Profit).' In: *Proceedings of the 5th International Workshop on Peer-to-Peer Systems*. Feb. 2006, pp. 1–1.

[LMJ13] Karsten Loesing, Steven J. Murdoch, and Rob Jansen. *Evaluation of a libutp-based Tor Datagram Implementation*. Tech. rep. TOR, Oct. 2013. URL: https://research.torproject.org/techreports/libutp-2013-10-30.pdf.

[LNR06] J. Liang, N. Naoumov, and K. W. Ross. 'The Index Poisoning Attack in P2P File Sharing Systems.' In: *Proceedings of the 25th IEEE International Conference on Computer Communications*. IEEE, 2006, pp. 1–12. DOI: 10.1109/INFOCOM.2006.232.

[Loc+06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. 'Free Riding in BitTorrent is Cheap.' In: *Proceedings of the 5th Workshop on Hot Topics in Networks*. 2006, pp. 85–90.

[Loc+10] Thomas Locher, David Mysicka, Stefan Schmid, and Roger Wattenhofer. *Poisoning the Kad Network*. Ed. by David Hutchison et al. Vol. 5935. Springer Berlin Heidelberg, 2010, pp. 195–206. ISBN: 978-3-642-11321-5 978-3-642-11322-2.

[Loe08] Andrew Loewenstern. *BEP 5: DHT protocol*. Tech. rep. BitTorrent, Inc., 2008. URL: http://bittorrent.org/beps/bep_0005.html (visited on 05/02/2016).

[LUM06] Arnaud Legout, Guillaume Urvoy-Keller, and Pietro Michiardi. 'Rarest first and choke algorithms are enough.' In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 203–216. DOI: 10.1145/1177080.1177106.

[Man+10] Pere Manils, Chaabane Abdelberi, Stevens Le-Blond, Mohamed Ali Kâafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. 'Compromising Tor Anonymity Exploiting P2P Information Leakage.' In: *CoRR* abs/1004.1461 (2010).

[Med11] Media Defender. Media Defender. Mar. 2011. URL: http://www.mediadefender.com (visited on 05/02/2016).

[MH01] Nelson Minar and Marc Hedlund. *A Network of Peers—Peer-to-Peer Models Through the History of the Internet*. Ed.

by Andy Oram. O'Reilly & Associates Incorporation, 2001, pp. 3–20. ISBN: 0-59600110-X.

[Mir+05] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet denial of service: attack and defense mechanisms*. March. Prentice Hall PTR, 2005. ISBN: 0-13-147573-8.

[MM02] Petar Maymounkov and David Mazières. 'Kademlia: A Peer-to-peer Information System Based on the XOR Metric.' In: *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*. 2002, pp. 53–65.

[MS07] Peter Mahlmann and Christian Schindelhauer. *Peer-to-Peer Netzwerke*. Springer-Verlag, 2007. ISBN: 978-3-540-33991-5.

[Nak09] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Tech. rep. Bitcoin, 2009. URL: https://bitcoin.org/bitcoin.pdf (visited on 05/02/2016).

[Nam09] Yuri Namestnikov. *The economics of botnets*. Tech. rep. Kaspersky Lab, 2009. URL: https://www.securelist.com/en/analysis/204792068/The_economics_of_Botnets (visited on 05/02/2016).

[Nie10] Heiko Niedermayer. 'Architecture and Components of secure and anonymous Peer-to-Peer Systems.' PhD thesis. Technische Universität München, 2010.

[Nor09] Arvid Norberg. *Local Peer Discovery Documentation*. uTorrent Forum. Oct. 2009. URL: https://forum.utorrent.com/topic/54306-local-peer-discovery-documentation/ (visited on 05/02/2016).

[Nor10] Arvid Norberg. *BEP 29: uTorrent transport protocol*. Tech. rep. BitTorrent, Inc., 2010. URL: http://bittorrent.org/beps/bep_0029.html (visited on 05/02/2016).

[NR06] Naoum Naoumov and Keith Ross. 'Exploiting P2P systems for DDoS attacks.' In: *Proceedings of the International Workshop on Peer-to-Peer Information Management*. 2006, pp. 47–53. DOI: 10.1145/1146847.1146894.

[NSH08] Arvid Norberg, Ludvig Strigeus, and Greg Hazel. *Extension Protocol*. Tech. rep. Bittorrent, Inc., Feb. 2008. URL: http://bittorrent.org/beps/bep_0010.html (visited on 05/02/2016).

[Ora01] Andy Oram, ed. *Peer-to-Peer – Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates Incorporation, 2001. ISBN: 0-59600110-X.

[Pet+06a] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. 'Experiences Building PlanetLab.' In: *Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI)*. 2006. URL: https://www.usenix.org/legacy/events/osdi06/tech/full_papers/peterson/peterson.pdf (visited on 05/02/2016).

[Pet+06b] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. 'Experiences Building PlanetLab.' In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. 2006, pp. 351–366.

[Pet05] Kim Peterson. *BitTorrent file-sharing program floods the Web*. The Seattle Times. Jan. 2005. URL: http://seattletimes.nwsource.com/html/businesstechnology/2002146729_bittorrent10.html (visited on 05/02/2016).

[Pia+07] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. 'Do incentives build robustness in BitTorrent?' In: *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*. 2007, pp. 1–14. URL: https://www.

usenix.org/legacy/event/nsdi07/tech/piatek/piatek.pdf (visited on 05/02/2016).

[Pir12] The PirateBay. *No more torrents=no changes anyhow*. 2012. URL: http://thepiratebay.se/blog/208 (visited on 05/02/2016).

[PLL11] Swagatika Prusty, Brian Neil Levine, and Marc Liberatore. 'Forensic Investigation of the OneSwarm Anonymous Filesharing System.' In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. ACM, 2011, pp. 201–214. DOI: 10.1145/2046707.2046731.

[Pre14] Chris Preimesberger. *DDoS Attack Volume Escalates as New Methods Emerge*. May 2014. URL: http://www.eweek.com/security/slideshows/ddos-attack-volume-escalates-as-new-methods-emerge.html (visited on 05/02/2016).

[Pri12] Matthew Prince. *Deep Inside a DNS Amplification DDoS Attack*. Oct. 2012. URL: http://blog.cloudflare.com/deep-inside-a-dns-amplification-ddos-attack/ (visited on 05/02/2016).

[Pri14] Matthew Prince. *Technical Details Behind a 400Gbps NTP Amplification DDoS Attack*. Feb. 2014. URL: https://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack/ (visited on 05/02/2016).

[RLD10] A. Rao, A. Legout, and W. Dabbous. 'Can Realistic BitTorrent Experiments Be Performed on Clusters?' In: *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*. Aug. 2010, pp. 1–10. DOI: 10.1109/P2P.2010.5569970.

[Ros+10] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. 'LEDBAT: the new BitTorrent congestion control protocol.' In: *Computer Communications and Networks (ICCCN)*. IEEE, 2010, pp. 1–6. DOI: 10.1109/ICCCN.2010.5560080.

[Ros14] Christian Rossow. 'Amplification Hell: Revisiting Network Protocols for DDoS Abuse.' en. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS 14)*. 2014, pp. 1–5. ISBN: 1-891562-35-5. DOI: 10.14722/ndss.2014.23233.

[Rou+04] Mema Roussopoulos, Mary Baker, David Rosenthal, TJ Giuli, Petros Maniatis, and Jeff Mogul. '2 p2p or not 2 p2p?' In: *Workshop on Peer-to-Peer Systems*. 2004, pp. 1–6. DOI: 10.1007/978-3-540-30183-7_4.

[RTV10] Dario Rossi, Claudio Testa, and Silvio Valenti. 'Yes, we LEDBAT: Playing with the new BitTorrent congestion control algorithm.' In: *Passive and Active Measurement*. Springer, 2010, pp. 31–40. DOI: 10.1007/978-3-642-12334-4_4.

[San+11] F.R. Santos, W.L. da Costa Cordeiro, L.P. Gaspary, and M.P. Barcellos. 'Funnel: Choking Polluters in BitTorrent File Sharing Communities.' In: *Network and Service Management, IEEE Transactions on* 8.4 (2011), pp. 310–321. ISSN: 1932-4537. DOI: 10.1109/TNSM.2011.110311.110104.

[Sav+99] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. 'TCP congestion control with a misbehaving receiver.' In: *ACM SIGCOMM Computer Communication Review* 29.5 (Oct. 1999), pp. 71–78. ISSN: 0146-4833.

[SB09] Moritz Steiner and Ernst W. Biersack. 'Where is my Peer? Evaluation of the Vivaldi Network Coordinate System in

Azureus.' In: *Proceedings of the 8th International IFIP-TC 6 Networking Conference*. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-01399-7_12.

[SBB05] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. 'Misbehaving TCP receivers can cause Internet-wide congestion collapse.' In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, 2005, pp. 383–392. ISBN: 1595932267. DOI: 10.1145/1102120.1102170.

[Sch01] Rolf Schollmeier. 'A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications.' In: *Peer-to-Peer Computing, 2001. Proceedings.* 2001, pp. 2–3. DOI: 10.1109/P2P.2001.990434.

[Sha+12] Stanislav Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. *RFC 6817: Low Extra Delay Background Transport (LEDBAT)*. Tech. rep. IETF, 2012. URL: https://tools.ietf.org/html/rfc6817 (visited on 05/02/2016).

[Sia07] Ka Cheung Sia. *DDoS vulnerability analysis of BitTorrent protocol*. Tech. rep. University of California, Los Angeles, 2007, pp. 1–8. URL: http://oak.cs.ucla.edu/~sia/pub/cs239spring06.pdf (visited on 05/02/2016).

[Sil15] Steven Siloti. *BEP: Persistent Indirect Reputation (Draft)*. Tech. rep. BitTorrent Inc., May 2015. URL: https://github.com/ssiloti/bep-persistent-credit/blob/master/one-hop-rep.rst (visited on 05/02/2016).

[Sin+06] A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. 'Eclipse Attacks on Overlay Networks: Threats and Defenses.' In: *Proceedings of the 25th IEEE International Conference on Computer Communications (IEEE INFOCOM 2006)*. IEEE, 2006, pp. 1–12. DOI: 10.1109/INFOCOM.2006.231.

[Sir+07] Michael Sirivianos, Jong Han, Park Rex, and Chen Xiaowei Yang. 'Free-riding in BitTorrent Networks with the Large View Exploit.' In: *Proceedings of the 6th International Workshop on Peer-To-Peer Systems*. 2007, pp. 1–7.

[SJ06] Saul Stahl and Paul E. Johnson. *Understanding Modern Mathematics*. Jones & Bartlett Pub (Ma), 2006. ISBN: 978-0763734015.

[STR07] Xin Sun, Ruben Torres, and Sanjay Rao. 'DDoS Attacks by Subverting Membership Management in P2P Systems.' In: *Proceedings of the 3rd IEEE Workshop on Secure Network Protocols*. IEEE, Oct. 2007, pp. 1–6. DOI: 10.1109/NPSEC.2007.4371618.

[SW05] Ralf Steinmetz and Klaus Wehrle. *What Is This 'Peer-to-Peer' About?* Ed. by Ralf Steinmetz and Klaus Wehrle. March. Springer-Verlag, 2005, pp. 9–16. ISBN: 978-3-540-29192-3.

[The15] The 8472. *BEP 0011: Peer Exchange (PEX)*. Tech. rep. BitTorrent Inc., Oct. 2015. URL: http://www.bittorrent.org/beps/bep_0011.html (visited on 05/02/2016).

[Use08] User:Amc1. *BitTorrent Peer Exchange Conventions*. Vuze. Feb. 2008. URL: http://wiki.theory.org/BitTorrentPeerExchangeConventions (visited on 05/02/2016).

[Use14] User:Limaner. *Wikimedia: DHT Example*. Licence: CC BY-SA 3.0. June 2014. URL: https://commons.wikimedia.org/wiki/File:Dht_example_SVG.svg (visited on 05/02/2016).

[Van10] Ernesto Van Der Sar. *BitTorrent Makes Twitter's Server Deployment 75x Faster*. TorrentFreak. July 2010. URL: https://torrentfreak.com/bittorrent-makes-twitters-server-deployment-75-faster-100716/ (visited on 05/02/2016).

[Van11a] Ernesto Van Der Sar. *Facebook Uses BitTorrent, and They Love It*. TorrentFreak. June 2011. URL: https://torrentfreak.com/facebook-uses-bittorrent-and-they-love-it-100625/ (visited on 05/02/2016).

[Van11b] Ernesto Van Der Sar. *uTorrent Keeps BitTorrent Lead, BitComet Fades Away*. TorrentFreak. Sept. 2011. URL: http://torrentfreak.com/utorrent-keeps-bittorrent-lead-bitcomet-fades-away-110916/ (visited on 05/02/2016).

[Van12] Ernesto Van Der Sar. *BitTorrent Traffic Increases 40 % in Half a Year*. 2012. URL: https://torrentfreak.com/bittorrent-traffic-increases-40-in-half-a-year-121107/ (visited on 05/02/2016).

[Van15a] Ernesto Van der Sar. *Attackers can 'Steal' Bandwidth from BitTorrent Seeders, Research finds*. Aug. 2015. URL: https://torrentfreak.com/attackers-can-steal-bandwidth-bittorrent-users-research-finds-140819/ (visited on 05/02/2016).

[Van15b] Ernesto Van der Sar. *BitTorrent Can Be Exploited for DoS Attacks, Research Warns*. Aug. 2015. URL: https://torrentfreak.com/bittorrent-can-be-exploited-for-dos-attacks-research-warns/ (visited on 05/02/2015).

[VLO10] Quang Hieu Vu, Mihai Lupu, and Beng Chin Ooi. *Peer-to-Peer Computing: Principles and Applications*. Springer-Verlag, 2010. ISBN: 978-3-642-03513-5.

[VV08] Kashi Venkatesh Vishwanath and Amin Vahdat. 'Evaluating Distributed Systems: Does Background Traffic Matter?' In: *Proceedings of the USENIX Annual Technical Conference*. 2008. URL: http://cseweb.ucsd.edu/~vahdat/papers/usenix08.pdf (visited on 05/02/2016).

[WH10] Scott Wolchok and J. Alex Halderman. 'Crawling BitTorrent DHTs for fun and profit.' In: *Proceedings of the 4th*

USENIX Workshop on Offensive Technologies. USENIX Association, 2010, pp. 1–8. URL: https://jhalderm.com/pub/papers/dht-woot10.pdf (visited on 05/02/2016).

[WK12] Liang Wang and Jussi Kangasharju. 'Real-World Sybil Attacks in BitTorrent Mainline DHT.' In: *Proceedings of the IEEE Global Communication Conference (GlobeCom)2012*. 2012. DOI: 10.1109/GLOCOM.2012.6503215.

[WK13] Liang Wang and Jussi Kangasharju. 'Measuring large-scale distributed systems: case of BitTorrent Mainline DHT.' In: *Proceedings of the 13th IEEE International Conference on Peer-to-Peer Computing*. IEEE, Sept. 2013, pp. 1–10. ISBN: 978-1-4799-0515-7. DOI: 10.1109/P2P.2013.6688697.

[Wu+10] Di Wu, Prithula Dhungel, Xiaojun Hei, Chao Zhang, and Keith W. Ross. 'Understanding Peer Exchange in BitTorrent Systems.' In: *Proceedings of the 10th International Workshop on Peer-to-Peer Systems*. 2010, pp. 1–8.

[YuK12] Ricky Kwok Yu-Kwong. *Peer-to-Peer Computing: Applications, Architecture, Protocols, and Challenges*. Chapman & Hall/CRC Computational Science Series, 2012. ISBN: 978-1-4398-0934-1.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of August 25, 2016 (`classicthesis` version 1.1).

DECLARATION

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

London, August 2016

Florian E. W. Adamsky