Technical Report
KN–2010–DiSy–02

Distributed System Laboratory

Analysis and Efficient Classification of P2P File Sharing Traffic

Thomas Zink **Marcel Waldvogel**
Distributed Systems Laboratory
Department of Computer and Information Science
University of Konstanz – Germany

This project has been funded by Huawei Technologies Co., LTD.

Abstract. Since the advent of P2P networks they have grown to be the biggest source of internet traffic, superseding HTTP and FTP. For service providers P2P traffic results in increased costs for both infrastructure and transportation. Interest is high to reliably identify the type of service to ensure quality of service. In this document we analyze P2P network architectures and give an overview of existing identification mechanisms. In addition we devise a simple identification scheme suitable for implementation in resources restricted environments with limited computational power and memory. The scheme is based on behavior analysis and as such is not prone to traffic obfuscation techniques.

Table of Contents

List of Figures

List of Tables

1 Introduction

Peer-to-peer (P2P) systems form overlay networks to provide distributed storage that allows queries and sharing. The overlay networks are dynamic by nature and can be of complex topologies. According to [1] P2P systems can be classified into three types.

- Centralized
- Decentralized but unstructured
- Decentralized but structured

Centralized. A central directory server holds the locations of all data present in the P2P network. Nodes query the server to retrieve a list of nodes that provide the desired content. Centralized systems do not scale well and also have a single point of failure. *Napster* introduced this architecture. *BitTorrent* also features a form of centralization in terms of trackers, though this has been loosened by introducing decentralized trackers.

Decentralized but unstructured. These systems neither provide a centralized directory nor any control about network topology and content placement. Nodes dynamically join the network and query neighbors for content. Neighbors are randomly selected from all available nodes. Queries are usually propagated using floods or random walks within a certain radius defined by a TTL. Unstructured networks are very robust towards changes but queries do not scale since the load on every node grows linearly with the numbers of queries. Nearly all P2P systems currently in use are decentralized and unstructured including *Gnutella*, *FastTrack/KaZaA* and *eDonkey*.

Decentralized but structured. Structured overlay networks are prominent in current research. They were primarily designed to improve data discovery and have constraints both on the network topology and content locations. Content is not placed randomly but at specific locations. The structure is usually built using a *distributed hash table* (DHT) where content is identified by a key and the associated value is a pointer to the target location. Nodes form a graph that maps the keys to values. A well known example is *Kademlia* which is also featured by the *eMule* network. A widespread prejudice states that structures overlays are to complex to maintain. However, as shown in [2] this need not be the case.

2 Related Work

Multiple different approaches for P2P traffic identification have been suggested based on signatures, patterns, payload or behavior. This section presents a summary of different identification mechanisms and exercises their applicability and dis-/advantages.

2.1 Transport layer identification

Previous work tries to identify P2P traffic [3] and super nodes [4] on the transport layer using flow identification techniques based on heuristics using the 5-tuple {source IP, destination IP, source port, destination port, protocol}. Traffic identification works as follows.

1. Source/destination IP pairs that use both TCP and UDP are flagged as P2P candidates. Only few other applications like DNS, IRC, online gaming... have similar behavior and are eliminated if identified using a lookup table of port numbers.
2. {source IP, source port} and {destination IP, destination port} pairs are examined. Pairs for which the number of distinct IPs and distinct ports are equal are considered P2P.
3. To decrease the risk of false-positives the identified P2P flows are matched against the behavior of known applications like eMail, DNS, HTTP, malware and online gaming.

The authors prove that P2P traffic can be identified with 95% accuracy using their heuristics.

Advantages and Limitations.

+ Protocol independent. Works on proprietary and encrypted protocols.
+ Non-intrusive. Requires little changes to flow identification schemes and does not strongly affect flows.
- No P2P application identification.
- Must be matched against the behavior of other well known applications to reduce false-positives.

To identify super nodes the previously identified P2P traffic is filtered using the following conditions.

1. Well-known super node ports. Super nodes usually use reserved well known port numbers.
2. Forwarding queries from peers. If egress timestamp − ingress timestamp < expected maximum forwarding time and the payload size of the egress packet = payload size of the ingress packet, it is considered to be a forward query.
3. Once a super node is identified, all nodes the query is forwarded to are marked as super nodes.

The authors conclude that the algorithm could not be verified because they were not able to provide a super node themselves so this approach to identify super nodes is to be considered experimental.

2.2 Application Signature Identification

[5] presents an application signature based identification scheme to identify P2P download traffic. The authors only concentrate on TCP transfer traffic of well known P2P protocols. They use fixed-offset string matching to identify P2P TCP segments. The following signatures are used for identification (we omit 'dead' protocols for simplicity).

Gnutella

– First string following TCP/IP headers is `GNUTELLA`, `GET`, `HTTP`

– If first string is `GET` or `HTTP` there must follow a field with either `User-Agent:`, `UserAgent:` or `Server:`.

ed2k

– First byte after TCP/IP headers is the ed2k marker `0xe3`.
– next four bytes as integer is equal to packet size - size of headers + 5.

BitTorrent

– First 20 bytes in TCP payload equal `0x13BitTorrent Protocol`.

Only the first packets at the beginning of the transfer phase need to be analyzed. Since fixed-offset string matching is considerable cheap the overall cost is reasonably small. The authors show that more than 99% of P2P traffic could be identified with their method.

Advantages and Limitations.

+ Reasonably small cost.
- Does not work on unknown, proprietary or encrypted protocols, unless the encryption sequences are captured.

2.3 Match Characteristic Strings

Similar to [5] Karagiannis *et al.* propose a string based characterization of P2P traffic identification [6],[7]. The authors analyze both TCP and UDP traffic and exercise multiple stages of identification. The following table summarizes the string against the packet payload is matched (again 'dead' protocols are omitted, for a complete overview see [7]).

P2P protocol	Strings	Transport Protocol
ed2k	`oxe3`, `oxc5`	TCP/UDP
BitTorrent	`GET /announce?info_hash`, `GET /torrents`	TCP
	`GET TrackPak`, `0x13BitTorrent`	TCP
	`0x00000005`, `0x0000000d`, `0x00004009`	TCP
Gnutella	`GNUTELLA`, `GIV`, `GET /uri-res/`, `GET /get/`	TCP
	`X-{Versio,Dynami,Query,Ultrap,Try,...}`	TCP
	`GND`	UDP

Table 1. Strings used for P2P identification

To identify P2P traffic the following algorithm is used.

1. If src/dst port matches well known P2P ports the flow is flagged as P2P.
2. Payload is compared to the string table. In case of a match the flow is flagged as P2P, else the flow is flagged as non-P2P.
3. If a UDP flow is flagged as P2P from 2, src/dst IPs are hashed into a table of IPs. All flows that contain an IP of this table are also flagged as P2P, even if there is no payload match. Flows with src/dst port of other well known applications are excluded.
4. If a TCP flow is flagged as P2P, src/dst IPs are hashed into a second table of IPs. All flows that contain an IP of this table are flagged as possible-P2P if they have been identified from 2. Flows with src/dst port of other well known applications are excluded.

2.4 Algorithm Behavior Identification

[8] proposes a scheme to identify BitTorrent peers using the behavior of BitTorrent's choke algorithm. Algorithm behavior analysis could also be exercised to other protocols but is protocol and revision specific. As such it is not further investigated. Interested readers are referred to the original paper.

3 P2P Protocol Analysis

This section analyses the properties of prominent P2P protocols and aims to find commonalities among different P2P networks that can be exploited for automatic identification of P2P traffic. The focus lies on answering the following questions.

- Identify signatures in P2P control and data flows, both for plain and encrypted protocols. How does encryption affect identification.
- Does identification of control flows allow automatic identification of data flows and vice versa.
- DPI is expensive. Are there other signature or template based approaches for identification to reduce complexity.
- Distinguish different types of unknown P2P traffic even after identification of most flows. Emphasize on encrypted protocols.

We first present shared properties of P2P applications, problems in identification and how they can be addressed. Then strategies, which have been previously proposed by former research are summarized. After that our analysis of significant P2P protocols follows.

3.1 Peer-to-Peer Application Properties

Shared Properties. With the exception of *Kademlia* all popular P2P systems use an unstructured overlay network. Though all protocols have default ports, random ports are possible and thus port numbers cannot be used for (positive) application identification. Also both TCP and UDP are common protocols to transport P2P packets. Though the networks are classified as being unstructured it is common to have a semi-structured architecture based on super peer-like entities.

The following common properties can be identified among different P2P networks.

- P2P systems cause at least two types of traffic: signaling and transfer.
- Bootstrapping: nodes connect to already known nodes to get information about possible download locations. To do so, the hosts keep local information of already known peers and probe them for availability.
- Handshaking: all P2P systems use some kind of handshaking to initiate connections.
- Ping, Pong, Keep Alive: Pings and pongs are common, as well as some kind of periodic keep alive messages.
- Special peers: Queries are usually forwarded to special peers with certain requirements. These are either super nodes or ultra peers. After that, peers connect to other peers for direct download.
- Usually control messages are sent using UDP while the actual file exchange is done using TCP connections.
- A number of connections is kept open, even if there are no active file transfers.
- On some protocols, the peer will constantly probe other peers, resulting in a high rate of opening connections.

Challenges in identifying P2P traffic. There are a number of challenges involved in identifying P2P traffic.

- Large number of self organizing hosts.
- Arbitrary ports for signaling and transfer traffic, and NAT makes identification on transport layer difficult.
- Multiple different flows – signaling, transfer – per application.
- High speeds.

– Encryption and obfuscation complicates payload based identification strategies.

To address these problems different methods can be exercised.

– Sampling: Capture only a fraction of the packets. Since the frequency of (especially) signaling traffic is high, only a fraction of these packets need to be captured to identify ip/port pairs. This can significantly reduce the bandwidth needed for identification.
– Association: Signaling traffic suffices to identify the ports on which the peers listen. Thus, TCP traffic to those ports can be tagged as data transfer traffic and does not need further effort.
– Matching few bytes of signaling packets to fixed strings is much cheaper as full DPI and enables identification of non-encrypted protocols.

3.2 Encryption, Obfuscation and Caching

To avoid detection and traffic shaping measurements from ISPs, P2P systems started integrating traffic encryption and obfuscation measurements into their systems. The main reason for any ISP to shape P2P traffic are either legal issues or financial issues. P2P traffic consumes vast amounts of resources and produce additional cost due to high inter-network traffic. Thus, ISPs started shaping or prohibiting P2P traffic much to the dismay of their customers. The P2P community responds with traffic obfuscation methods. This section describes common features of protocol encryption. For details how encryption is implemented in specific protocols see the corresponding sections.

Methods. Common techniques include using handshaking to exchange key information (like BitTorrent's Diffie-Hellman-Merkle key exchange) or – as with Skype – proprietary encoding of the payload and multistage encryption using hashing and multiple encryption algorithms.

– Use global information and random seeds to generate keys.
– Use packet information and random seeds to generate keys.
– Use handshaking to exchange key information.

These techniques are most prominent in Skype and also many BitTorrent clients. In general it is much easier to encrypt peer-to-peer TCP connections than UDP signaling traffic. Since UDP is connectionless ping and keep-alive messages can only be encrypted using packet internal information whereas TCP connections allow key exchange methods.

In general P2P systems must somehow exchange key information. Signaling traffic is likely to be unencrypted or weakly encrypted by using packet internal information or simple encoding schemes.

Dispute and Caching. This approach is subject to much dispute even in the P2P community. BitTorrent's inventor Bram Cohen attacks encryption on his internet journal [1]. The main points of his critic are:

1. Large traffic that looks like noise is not more difficult to identify than unencrypted flows.
2. Incompatibility issues between clients.
3. ISP content caching is obliterated by encryption/obfuscation.
4. Obfuscation is unprofessional, hostile and harmful.

To allow ISPs to reduce traffic cost, BitTorrent Inc. and CacheLogic developed the *Cache Discovery Protocol*. ISPs can automatically detect requested content, cache and seed it to their customers thus reducing extra-network traffic and cost. Caching is also widely considered to be legal, since it is not equivalent to copying. Naturally, obfuscation renders caching useless, which is one of the reasons why Bram Cohen has a strong averseness to it.

[1] http://bramcohen.livejournal.com/29886.html

Detecting Encrypted and Obfuscated Traffic. There are multiple approaches on how to identify encrypted flows.

- Exploit the fact that encrypted traffic looks like noise.
- Exploit knowledge about packet sizes, transport protocols and transmission rates.
- Use man-in-the-middle attacks to intercept key exchanges.
- Intercept retrieval of global information needed to set up encryption keys.

The chosen approach mainly depends on the methods used for the specific application. Skype for example, is heavily encrypted and looks nearly completely like noise which is at the time of this writing a unique feature. Even VPNs – like IPsec – have specific signatures which make them detectable. Of course, if more applications use fully encrypted communication it would be impossible to distinguish those applications based solely on noise. For suggestions of how to identify specific encrypted protocols, see the corresponding sections.

Modern traffic identification systems that claim to reliably identify applications use a combination of signature based DPI and application behavior analysis. They provide regular signature updates to keep up with the evolution of P2P and other applications.

3.3 Gnutella

Gnutella is a direct successor of *Napster* and the first P2P application that introduced a decentralized overlay network. In the *Gnutella* network all peers function as clients and servers alike, called SERVENTS. It was introduced by Nullsoft and soon made open-source. Today the most prominent *Gnutella* client is *LimeWire*[2]. Since *Gnutella* is open-source it is well documented and developers frequently exchange and implement RFCs.

Terminology. Throughout the next paragraphs the following terms will be used.

servent Any peer in the Gnutella network
client A client that wishes to connect, issue queries, download files
server A peer that responds to queries and uploads files.
ultrapeer A special peer that serves as a mediator between leafes
leaf Any peer that is not an ultrapeer
user agent The Gnutella client software running on a peer.

Overlay Network. The *Gnutella* overlay has significantly change from *v0.4* to *v0.6*. Older *Gnutella v0.4* servents can both issue and respond to queries. Queries are distributed by floods. A servent broadcasts the query to all neighbors which forward the query similarly up to a maximum radius. Figure 1 depicts the *Gnutella v0.4* architecture.

This design had certain flaws which could lead to bottlenecks. *Gnutella v0.6* introduces ULTRAPEERS which implement special services like flow control and Pong-caching. Other peers are called *leaves* and keep only a small number of connections to ultrapeers. Figure 2 shows the *Gnutella v0.6* network. Each node decides independently to become an ultrapeer based on local information which includes firewalling, operating system, bandwidth, uptime, RAM and CPU power. Leaves only forward messages to all it's ultrapeers while an ultrapeer relays messages to all other connected ultrapeers and only to leaves if they decide the leaf can satisfy the request.

[2] http://www.limewire.com/

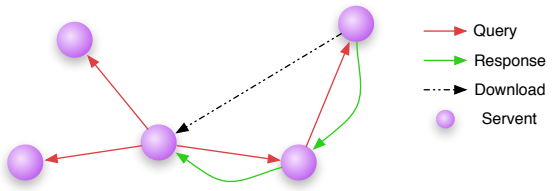

Fig. 1. Gnutella v.04 network

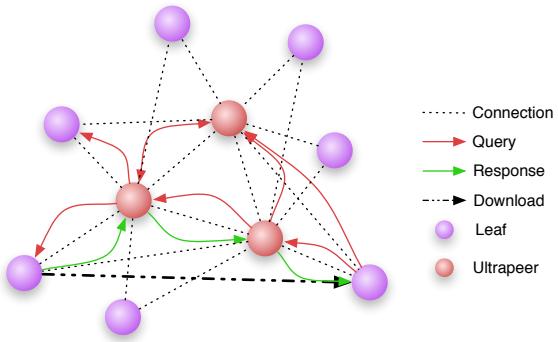

Fig. 2. Gnutella v.06 network

Connect and Participate. To connect to the network a peer must be aware of ultrapeers that are already connected to the *Gnutella* network. A peer keeps a local cache of well known *Gnutella* servents. The joining peer rapidly sends UDP `PING` messages (see below) to known servents to discovery which are actually online. On reply (`PONG`), the peer tries to connect using the *Gnutella* handshake (see below). After connecting to an ultrapeer the client propagates its file list to the ultrapeers and can start exchanging binary messages.

A *Gnutella* session can thus be described as follows.

Retrieve ultrapeers Client connects to ultrapeer databases and retrieves recent ultrapeer lists. Usually manual or automatic HTTP download.

Discovery Client rapidly sends UDP `PING` messages to find online ultrapeers. Source port is user-agent listen port.

Ultrapeer Connection Open TCP connection to online ultrapeers. Immediately perform *Gnutella* handshake (see below). On error, receive new ultrapeers and try again.

Message Exchange User defined `QUERY` messages are forwarded to all connected ultrapeers, then between ultrapeers and finally to leaves suitable for the query. `QUERYHIT`s can either be directed at the responsible leaf or it's ultrapeer using UDP.

Client-Client Communication Users can browse through shared folders of each other. Requests and file lists are exchanged using UDP.

Data Transfer Downloads are done using direct HTTP connections between servents. The client initiates the TCP connection and requests files using HTTP GET requests. The server answers with HTTP OK.

Protocol Specification. The current *Gnutella* protocol is version 0.6 [3]. A Gnutella servent connects to the network by establishing connections with ultrapeers already present in the network. There are multiple methods to retrieve lists for connected hosts like manual or automatic downloads. Connecting clients must first establish a TCP connection and then proceed with the Gnutella handshake.

Gnutella Handshake.

1. Client establishes TCP connection.
2. Client sends `GNUTELLA CONNECT/0.x`, where x is the protocol version. The connection request also carries client information like supported features, user-agent, the client IP and port.
3. Server answers either with `GNUTELLA/0.x 200` in case the connection is accepted or with `GNUTELLA/0.x 503` when no leaf slots are available. In that case the Ultrapeer replies with an `X-Try-Ultrapeers` header bearing {`IP:Port`} tuples of known ultrapeers.
4. A client that wishes to connect sends `GNUTELLA/0.x 200`.
5. Both peers can start to exchange binary messages at will.

Note that the user agent opens multiple ports for connections to ultrapeers and send queries from different ports than it accepts connections.

An observed client connection request looks like follows (all IPs and ports are replaced by random numbers).

```
GNUTELLA CONNECT/0.6
X-Max-TTL: 3
X-Dynamic-Querying: 0.1
X-Requeries: false
X-Query-Routing: 0.1
User-Agent: LimeWire/5.1.2
Vendor-Message: 0.2
X-Ultrapeer-Query-Routing: 0.1
GGEP: 0.5
Listen-IP: 207.112.39.83:17784
Accept-Encoding: deflate
Pong-Caching: 0.1
X-Guess: 0.1
X-Ultrapeer: False
X-Degree: 32
X-Locale-Pref: en
Remote-IP: 19.179.51.127
```

An example ultrapeer 503 reply:

```
GNUTELLA/0.6 503 No Leaf Slots
X-Try-Ultrapeers: 201.170.245.144:39697,214.204.30.68:19678,67.225.145.85:31727
```

An example ultrapeer 200 reply:

```
GNUTELLA/0.6 200 OK
Listen-IP: 172.38.35.223:32758
Remote-IP: 48.105.117.144
User-Agent: LimeWire/4.18.8
```

[3] http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html

```
X-Ultrapeer: True
X-Degree: 24
X-Query-Routing: 0.1
X-Ultrapeer-Query-Routing: 0.1
X-Max-TTL: 3
X-Dynamic-Querying: 0.1
X-Locale-Pref: en
GGEP: 0.5
Bye-Packet: 0.1
X-Try-Ultrapeers: 199.146.247.162:24971,186.65.239.213:18912,118.200.6.233:11669
```

Gnutella Messages. Messages are exchanged using UDP. Source port is the user agent's listen port. Each message has a 23 byte header as depicted in figure 3.

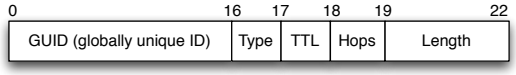

Fig. 3. Gnutella message header

`GUID` A globally unique message ID. Byte 8 should contain `0xff` for all modern clients. Byte 15 should contain `0x00` since it is reserved for future use.

`Payload Type` The type of message (see below)

`TTL` Time-To-Live. Number of times the message will be forwarded in the overlay before it is removed.

`Hops` Number of times the message has been forwarded in the overlay.

`Payload Length` The length of the message in bytes from end of header. In little endian. This number plus 23 equals the message size in bytes and are the only reliable source for the client to identify the actual message content.

The following message types are allowed.

`0x00 PING` Used to actively discover hosts on the network. A servent receiving a Ping message is expected to respond with one or more Pong messages.

`0x01 PONG` The response to a Ping. Includes the address of a connected Gnutella servent, the listening port of that servent, and information regarding the amount of data it is making available to the network.

`0x80 QUERY` The primary mechanism for searching the distributed network. A servent receiving a Query message will respond with a Query Hit if a match is found against its local data set.

`0x81 QUERYHIT` The response to a Query. This message provides the recipient with enough information to acquire the data matching the corresponding Query.

`0x40 PUSH` A mechanism that allows a firewalled servent to contribute file-based data to the network.

`0x02 BYE` An optional message used to inform the remote host that the servent is closing the connection, and the reason for doing so.

Query Routing Protocol QPR The QPR is the protocol used in an ultra peer scenario to route queries in the network. Each ultrapeer maintains a local hash table mapping files to connected peers. Peers use QRP to update the ultrapeers hash table as well as query files.

Other Protocols. Multiple extension to the protocol exist, such as GGEP (Gnutella Generic Extension Protocol), HUGE and XML. Reserved bytes in the message payload indicate which extensions the peers support and affects both signaling as well as data transfer traffic.

Traffic We have identified different types of traffic which are summarized in table 2. Here 'rnd' means a port randomly chosen on initiating the connection while 'def' denotes the user agent's defined listen port which is either left on the default value or chosen by the user.

description	peer-2-ultrapeer connection	messages	data transfer
type	control	control	data
transport layer	TCP	UDP	TCP
application layer	QRP	gnutella	HTTP/HUGE
direction	p2up	p2p, p2up, up2p	p2p
src port	rnd	def	rnd
dst port	def	def	def
purpose	join participate query routing	exchange messages	exchange data
handshake	Gnutella (join)	no	TCP
header	no ?	message header	no
signature	handshake `GNUTELLA`	paket size >= 23 B B8 should be `0xff`, rarely is B15 should be `0x00` B16 in { `0`, `1`, `2`, `0x40`, `0x80`, `0x81` } B19-22 (little endian): Gnutella payload length (= *UDP payload length* − 23)	should start with `GET`, rarely does
frequency	high at start infrequently afterwards	$\approx 7/sec$ frequent	only during transfer high amount of traffic

Table 2. Analysis of captured Gnutella traffic

Identification The following algorithm shows how to identify IP/port pairs of *Gnutella* peers.

1. If `protocol` = `TCP` & `relative sequence number` = 4 & `bytes`$(0, 7)$ = `GNUTELLA`, then a leaf Gnutella handshake message is caught. The scr IP can be added to the list of Gnutella peers, the src port is a random port for this connection. The dst IP and port can be added to the Gnutella candidate peers.
2. If the response messages first 8 bytes equal `GNUTELLA`, the dst IP and port can be added to Gnutella peers.
3. It is possible but expensive to further dissect the response message to identify other Gnutella peers.
4. If `protocol` = `UDP` & `length` ≥ 23 & `byte`$(15) = 0$
& `byte`$(16) \in \{0, 1, 2, 0x40, 0x80, 0x81\}$ & `bytes`$(19, 22)$ = `length` $- 23$, then src IP/port and dst IP/port are Gnutella peers. These ports are also the listening ports on which data transfer flows. And can be used to identify connecting hosts as Gnutella peers.

3.4 BitTorrent

BitTorrent started in 2001 with a simple post on a forum: "My new app, BitTorrent, is now in working order, check it out here" [4]. It has been developed by Bram Cohen who describes it in [9]. Since then it has grown to be the most popular file sharing tool currently in use.

Overlay Network. The *BitTorrent* network consists of the following entities:

.torrent aka metainfo file. File with meta-data about the download file.
Webserver An ordinary web server that hosts the .torrent file.
Tracker A centralized server that maintains a list and the states of peers actively downloading and uploading the file. It mediates between peers and answers requests. A new extension [5] also allows decentralized trackers based on a Kademlia DHT.
Peer Any host participating in the *BitTorrent* network.
Seed Any peer that has all parts of the shared file. At least one seed for each file must be present in the network.
Leech Any peer that is not a *Seed*. The term is generally used negatively for users downloading a file and not seeding afterwards.

Figure 4 shows the network.

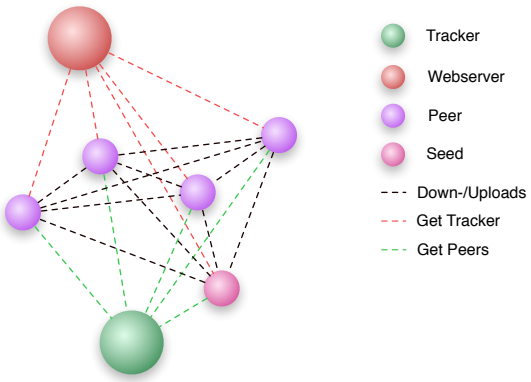

Fig. 4. BitTorrent network.

bencoding. Nearly all messages as well as the .torrent file are encoded using a scheme known as *bencoding*. It defines the following formats.

`byte strings` <len in base 10 ascii>:<string>
`integer` i<integer in base 10 ascii>e
`list` l<bencoded values>e
`dictionary` d<bencoded string><bencoded value>e

[4] http://finance.groups.yahoo.com/group/decentralization/message/3160
[5] http://www.bittorrent.org/beps/bep_0005.html

Protocol Specification. The trackers use a simple protocol on top of HTTP. A downloader sends a tracker-`GET` request which contains its random ID, IP, Port and information about downloaded and uploaded amount of data. The tracker responds with a (randomly) generated list of peers that offer the file. The peers then connect to each other exchanging pieces of the file. Peer connections are symmetrical. Files are split into pieces of commonly 2^{18} byte. An extension allows UDP connections for tracker requests. It is supported by a small number of clients, including Azureus/Vuze.

Peers communicate and exchange data over TCP. The *BitTorrent* protocol is also known as "peer wire protocol". After the TCP handshake a peer must immediately send the *BitTorrent* handshake message, which is easily detectable. Then data or message exchange can commence. The handshake message is 49+len(pstr) Bytes long and has the following fields.

`pstrlen` 1B, length of the following string
`pstr` string identifier
`reserved` 8B, for extensions. Each bit can be used to state support for different extensions.
`info_hash` 20B, SHA1 hash of the info key of the .torrent file.
`peer_id` 20B, unique peer ID. Includes the client ID and client version.

The current protocol implementations all use pstrlen = `0x13` and
pstr = `BitTorrent Protocol`. All other messages using the peer wire protocol have the following structure.

`message length` 4B, length of the message
`type` 1B, type of message, not present for keep-alive
`payload` optional message content

There are multiple messages peers can exchange. Some of them are used to exchange status information or request and find file pieces. Depending on the type of message the payload is present or not which usually contains index information for file pieces.

In addition to the peer wire protocol the DHT extension also implements the so called KRPC protocol used to query for files and nodes in the DHT. It is a simple RPC mechanism which is transported over UDP. Each KRPC message is sent within one UDP packet without any retransmits. The UDP packet contains a bencoded dictionary that encodes query arguments and return values. There are three types of KRPC messages, query, response and error. They start with the following string sequences.

`query` `d1:a`
`response` `d1:r`
`error` `d1:e`

Query messages transport query strings as arguments (as indicated by the key a) inside a dictionary. Responses also are dictionaries with response strings. Error messages contain a list of error codes and strings

Traffic. Table 3 summarizes the flows we could identify in a few hours long *BitTorrent* session. We used the official *BitTorrent* client on multiple virtual machines and shared and seeded freely available files. We do not analyze Tracker traffic, since it is far to rare to be used for flow identification. In the table, 'rnd' means a port randomly chosen on initiating the connection while 'def' denotes the user agent's defined listen port which is either a default value or chosen by the user.

description	KRPC messages	messages/data
type	signal	signal, transfer
transport layer	UDP	TCP
application layer	peer wire/KRPC	peer wire
direction	in/out	in/out
src port	def	rnd
dst port	def	def
purpose	exchange messages	exchange message/data
handshake	no	bittorrent
header	message header	message header
observed signatures	size >= 5 B B0-4: `d1:{a,r,e}`	size >= 49 B B0-19: `0x13BitTorrent Protocol`

Table 3. Captured BitTorrent Traffic

Encryption and Obfuscation Many *BitTorrent* clients – excluding the official one – implement methods for encryption and obfuscation, known as *Message Stream Encryption (MSE)/Protocol Encryption PE*. The official *BitTorrent* client is able to accept but does not reply with encrypted packets. *BitTorrent* uses the infohash field of the .torrent file combined with a Diffie-Hellman-Merkle key exchange to set up an RC4 encrypted connection. Using the infohash from the .torrent file prevents man-in-the-middle attacks. *BitTorrent* only encrypts TCP traffic transported using the peer wire protocol. Signaling traffic, which is transported via UDP is unencrypted since no key exchange is possible.

Identification

1. If `protocol` = `TCP` & `rel.seq.num.` = 4 & `bytes`$(0, 4)$ = `0x13Bit`, then a BitTorrent Handshake is caught and data transfer is initiated. Src IP, dst IP/port can be tagged as BitTorrent peers. The src port is random and specific for that connection.
2. If `protocol` = `UDP` & `length` ≥ 5 & `bytes`$(0, 4) \in$ `d1:{a,r,e}` a KRPC message is caught and the src IP/port and dst IP/port can be added to known BitTorrent hosts. These ports also listen for data transfer.

Since *BitTorrent* UDP traffic is always unencrypted it is easy to identify *BitTorrent* peers solemnly on capturing signaling traffic. This allows easy identification of IPs and listening ports. Any traffic to these peers can be classified as *BitTorrent* traffic.

3.5 FastTrack/KaZaA

FastTrack is a proprietary protocol introduced by Sharman Networks that gained much attention in 2003. It uses an architecture similar to *Gnutella v0.6* consisting of SUPERNODES (SN) and ORDINARY NODES (ON), however, historically it precedes *Gnutella v0.6*. The protocol has not been documented, but the ON-SN protocol has been largely reverse-engineered [6] with SN-SN communication still being unknown. Documentation is also very out-dated ranging back to 2004. [10] extensively analyses the *KaZaA/FastTrack* network.

As other P2P networks, *KaZaA* has been subject to lawsuits initiated by the RIAA. This lead to *KaZaA* being sold and turned into a payment-based online music download service. The original *KaZaA Multimedia Desktop*, which was famous for its addition of malware is no longer available.

[6] http://cvs.berlios.de/cgi-bin/viewcvs.cgi/gift-fasttrack/giFT-FastTrack/PROTOCOL?revision=1.19

The once popular *KaZaA lite* client has also been discontinued. There have been several attempts to resurrect the *KaZaA/FastTrack* network, including *Kazaa Lite Resurrection* and *MLDonkey*, which once provided connectivity to *FastTrack*. However, the current state is that *KaZaA/FastTrack* is effectively dead.

Network. SNs are special nodes that are highly available, have large bandwidth, and high processing power. SNs must volunteer to get elected, they are not automatically spawned as in the *Gnutella v0.6* network. Each ON connects to at least one SN and uploads meta-data of all files they want to share with other nodes. ONs query SNs for files which broadcast the queries to other SNs and check the query for matches in their meta-data base and respond with qualified hosts. Figure 5 depicts the network architecture.

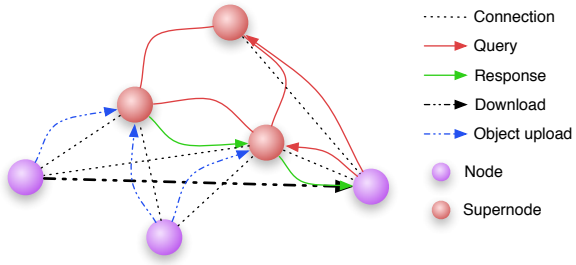

Fig. 5. FastTrack network.

Protocol Specification. As already mentioned little is known about the *FastTrack* protocol specification and documentation is out-dated. In addition connections between ordinary nodes and super nodes is encrypted and uses random ports and TCP as well as UDP. *KaZaA* uses HTTP-like headers for file transfers which can be easily distinguished from ordinary HTTP traffic since they are plain text.

Traffic. Unfortunately, we were not able to connect to the *FastTrack* network. All the original clients ceased existence. Clients still in development like *MLDonkey* removed *FastTrack* connectivity. We tried several days to connect to the *KaZaA* network using the most recent version of *Kazaa Lite Resurrection*, however, we had no luck in joining the network due to a too small number of connected hosts. Thus, we were not able to actively identify different types of *KaZaA* traffic. The only type of messages we could capture were UDP packets presumably used for discovery. All were sent from the user agent's *KaZaA* port to different host IPs/ports and had the following 12 byte content: `0x27 00 00 00 29 80 KaZaA.`.

Identification. Apart from discovery UDP packets we were not able to capture any *KaZaA* traffic. Thus we can not derive any means for identification except looking for 12 byte UDP packets ending with the string `KaZaA.`. However, since *KaZaA/FastTrack* is effectively dead and has no impact on today's internet traffic any effort of identifying it's traffic is probably wasted.

3.6 Skype

Skype has been developed in 2003 by the same people that also invented *KaZaA*. Thus, it shares several features with P2P protocols, especially the overlay network. However, it is used mostly in a one-to-one or one-to-few contact scenario, typically between people knowing each other, and rarely for filesharing. *Skype* traffic is heavily encrypted and obfuscated. The binary as well as network traffic have been reverse engineered in [11], [12]. There have been previous attempts to identify *Skype* traffic which will be introduced in the following sections.

Overlay Network. The *Skype* overlay network is relatively similar to the *KaZaA* network. It consists of super-nodes which can be used to relay traffic behind NATs and firewalls. Any node can become a super-node based on availability, bandwidth and similar criteria. [11] states that there are $\approx$ 200 super-nodes hardcoded into the *Skype* binary that change on every release. It is currently unclear, whether the super-nodes are also used as a distributed dictionary of nodes and skype IDs, or if a centralized approach is taken.

Encryption and Obfuscation *Skype* uses multistage encryption based on hashing, AES, RSA and RC4 to encrypt it's messages. It uses both TCP and UDP as transport layer protocol. In case of TCP the whole message is encrypted. In case of connectionless UDP, the message cannot be fully encrypted but only obfuscated, that is, the user agent must extract header information to encrypt/decrypt messages. In addition, the UDP port must be fixed.

Skype UDP. Skype UDP packets are encrypted with RC4. The key is calculated using elements from the datagram, including src/dst IP, a Skype packetID and parts of the actual payload. The following fields can be identified in a Skype UDP message.

`ID` 16-bit random identifier.
`FUN` 3 random bits followed by a 5 bit function (payload) type. Five different types have been identified: {`0x02`, `0x03`, `0x07`, `0x0f`, `0x0d`}.

Skype TCP. The whole TCP stream is encrypted using RC4. The seed is sent in the first 4 bytes. In addition the data can be fragmented and sent over multiple packets. Data is packed using arithmetic compression very similar to huffman encoding prior to encryption.

Traffic. [13] identifies three types of traffic caused by *Skype* which can be categorized into E2E (end-to-end or peer-to-peer) and E2O (skypeout messages over PSTN gateways). Depending on the type of traffic packets show different characteristics which provide first clues for identification.

E2E over UDP The five `FUN` bits are deterministic. Other bits appear completely random.
E2O over UDP The first four bytes are deterministic and represent a *Connection Identifier* (`CID`). The `CID` is likely to change during connection setup but is stable afterwards. All other bytes are cyphered and random. The port is fixed to 12340.
E2E/E2O over TCP The whole message is cyphered and completely random.

In addition the following observations are worth mentioning.

- Packet size depends on the codec used and is stable throughout the conversation.
- Transmission rate must be fast and constant to maintain a good quality of service.

In essence, this means that *Skype* traffic consists of rapid, equal-sized packets, transported via UDP/TCP. With the exception of a few bits depending on the type the packets appear to be completely random.

Identification There have been previous attempts to identify *Skype* traffic. A rudimentary guideline is given in [11],[12], who try to detect *Skype* by the first UDP NAck packet, which can be identified by length and the `FUN` bits. Further effort is done in [13]. The authors propose three different *Skype* classifiers, based on methods prominent in data mining applications that exploit statistical information and probabilities. They are usually quite expensive and cannot easily be done in hardware. Another approach that tries to solve complexity and bandwidth problems is proposed in [14]. Here, a signature based on deltas between packets of a flow is used for identification.

Chi-Square Classification.
The chi-square classifier of [13] tests whether the inspected flow behaves like a *Skype* flow. To determine the chi-square error, groups of bits of the packet are examined over time and compared to the expected behavior stated in section 3.6. Random bits are expected to follow a specific distribution and the error can be relatively easy computed.

Naïve Bayes Classification.
Bayes classification is a standard information mining application and also prominent in statistical analysis. It examines certain features and based on the values of the features computes a probability that the examined object belongs to a specific class. In this case the authors are interested in message size and transmission rate of the packets. Based on the codec used the packets have a distinct size. The rate may not drop below a certain threshold. Using these features the probability that a flow is either *Skype* or not can be computed.

Payload Based Classification.
Skype makes payload based classification difficult due to encryption and obfuscation. However, there is some information both in the payload as well as in the headers that can be used for identifying *Skype* flows. The Classifier uses payload information introduced in section 3.6 to identify E2E and E2O packets. Then timing information is taken into account to classify the flows.

Uncooperative Identification.
In [14], the author bases his scheme on the following observations.

- *Skype* flows have a constant rate (isochronicity).
- *Skype* flows have a constant packet size.
- Packets in *Skype* flows appear random, with the exception of certain bits.

The identification of *Skype* traffic works in multiple stages.

- Flows are defined only by the tuple {dst IP, dst port}. Since the Source IP/port can be easily forged and NAT and proxy networks obfuscate the true source they are not taken into account when identifying flows.
- Incoming packets that are bigger in size than a certain threshold are filtered out.
- Remaining packets are hashed into a table of flows and each flow is associated with a timer and a counter.
- Flows in which the number of packets in a given interval exceed and under-run specific thresholds over a period of time (30-100 packets per second), are removed from consideration.
- Remaining flows are matched against a pre-computed signature. The signature is based on changes of 4 bit wide blocks of the packet payload within the flow. Each 4 bit block is seen as an integer value. The delta between packets is derived by simple subtraction. Then the mean and standard derivation of the deltas is computed and used as a signature. As shown in the thesis [14], different protocols show a specific signature – including Skype – which can be used to distinguish them from other protocols.

The methodology works on high speed due to early filtering of flows and uses multiple tricks to reduce memory requirements (multiple hashing, counting bloom filters, aggregated timestamps). Furthermore it is easy to implement in hardware. The downside is, that it is not well tested and only applied to actual *Skype* voice conversation. However, the use of filtering and flow signatures is intriguing and a good starting point for detecting encrypted flows like *Skype* or maybe other encrypted P2P protocols.

3.7 Kademlia

Kademlia [15] is a decentralized and structured P2P system that uses a DHT and a novel XOR based distance metric. It has been integrated in *eMule*, *mlDonkey*, and certain BitTorrent clients and has since gained a lot of attention both in the file-sharing community and in academics.

Overlay. Each node is assigned a unique 160 bit NODEID. Files are also identified by a 160 bit KEY (usually a SHA-1 hash of it's content). Files are stored at nodes with a CLOSE NodeID, where close is determined by XOR(NodeID,key). Each node keeps a list of {IP,Port,NodeID} triples for nodes of distance between 2^i and 2^{i+1} called k-buckets. This list is updated when a node receives a request from another node.

Protocol. The protocol consists of four RPCs.

`PING` probe a node to check it is alive.
`STORE` instruct a node to store a {key,value} pair.
`FIND_NODE` query a node with a NodeID. The recipient returns {IP,Port,NodeID} triples.
`FIND_VALUE` like `FIND_NODE`. If recipient has received a `STORE` it returns the value.

For all RPCs the recipient echoes a 160 bit random RPC ID which can include a PING. *Kademlia* nodes use UDP packets to communicate and exchange messages.

Lookups and Joining. Nodes perform a recursive `node lookup`. They select α nodes from their k-bucket an initiate parallel `FIND_NODE` RPCs. Of the k retrieved nodes closest to the target it recursively selects another α nodes to query. Nodes that do not respond in a given time are removed from consideration. Retrieval if values works similar but the `FIND_VALUE` RPC is used. Each node refreshes it's buckets by periodically performing node searches for random IDs. Joining a network requires a node to connect to an already connected node which is added to the k-bucket. The joining node then performs a node lookup of it's own ID and refreshes all k-buckets for hosts further away than the nearest neighbor. In addition all nodes have to periodically (every hour) republish keys. The exception is if a node recieved a `STORE` RPC for a key, it is not republished in the next hour.

3.8 eD2k/Overnet

Overnet was a two-layer P2P network consisting of clients and servers. It has been taken down in 2006 due to pressure from RIAA and other parties. Since then pressure on Overnet/eD2k servers has increased. Nevertheless *eDonkey* and derivates are still popular in some parts of the world and are often only superceded by *BitTorrent*. However its popularity is shrinking fast. The *Kademlia* protocol has since been integrated into *eMule* and *MLDonkey* and is considered to be the future [7]. The original client *eDonkey2000* was developed by MetaMachine. It is discontinued since 2005 and has been replaced by others, most notably *eMule, aMule, MLDonkey*. The server software has been reverse engineered and is known as LUGDUNUM.

[7] http://www.amule.org/wiki/index.php/FAQ_ed2k

Network. Clients must know the IP of a server to connect to. As in *FastTrack*, the clients send meta-data to the server to register files they offer for sharing. Clients can then query and request files using unique identifiers. The server responds with download locations, and clients connect to other clients for downloading. Most notably for ed2k clients is that they define two different listening ports one each for TCP and UDP traffic. So there is no correlation between TCP and UDP traffic and both seem to be random on each side of the conversation.

3.9 OpenNAP/WinMX

OpenNAP is an extended open source re-implementation of the original *Napster* protocol. It is thus client/server based. *WinMX* started as an *OpenNAP* client for Windows. A proprietary protocol called WPNP has been introduced by Frontcode. *WinMX* has practically been shutdown by the RIAA. Community efforts try to keep the network alive but it is only slightly popular, and overshadowed by *BitTorrent* and others. The protocol is not documented and to the best of our knowledge has not been subject to academic research.

3.10 Contrasting Protocols

Non-P2P/filesharing protocols that have similar properties under certain circumstances include, but are not limited to the following.

- HTTP
- Gaming Protocols
- TeamSpeak (in-game voice communications)
- VoIP

4 Efficient P2P Classification

Contemporary traffic classification systems rely on complex deep packet (DPI) inspection or statistical/behavior analysis (BA) and require vast amounts of resources. These systems are usually highly integrated and feature a multi-layered approach that combines different methods.

We present a simple identification algorithm, which is targeted on resource restricted environments like physical interface cards with few amounts of memory and little processing power. The algorithm is not intended to replace complex traffic classification systems but to provide fast and efficient pre-classification to single out interesting flows and thus reduce the pressure in later identification stages.

The algorithm exploits the observation that P2P applications usually use both TCP and UDP in parallel for signal and transfer traffic. Depending on the point of identification in the network, either core or edge, the view of network traffic between hosts is different.

4.1 Classify large flows as P2P on Edge

Since an edge router connects the provider's network (PN) to foreign networks (FN) we must distinguish between incoming and outgoing traffic. The edge router can detect all connections from clients in the PN but has very limited information about connections from clients in FNs.

Outgoing connections Nodes participating in P2P networks that are attached to the PN produce signal traffic to multiple nodes in one or more FNs. File transfers are initialized from the PN and produce massive amounts of incoming TCP traffic. Thus the edge router has a complete view of all signaling and transfer traffic originating from a PN node. Figure 6 depicts P2P traffic from a sample PN node.

In this scenario signal traffic has both a fixed source IP and port while the destination IPs and

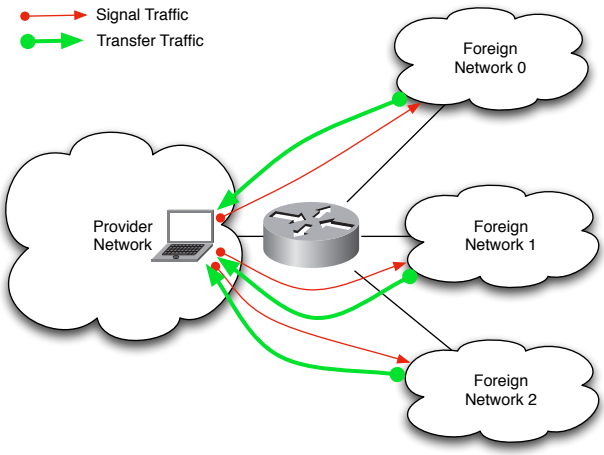

Fig. 6. Outgoing P2P traffic.

ports appear random. The TCP transfer traffic has a random source port but a fixed destination port for each distinct destination IP. For large incoming TCP flows which have been initialized from the PN there are many UDP signal packets from the destination PN node to FN nodes

where a fraction must have destination IP and port equal to the TCP flows source IP and port. In addition transfer flows are only acknowledged with TCP ACK packets without payload having a constant size. This allows to differentiate between HTTP and P2P traffic. Table 4 summarizes the properties.

The following algorithm is a simple scheme for identification.

properties	signal traffic	transfer traffic
outgoing src port	fix	rnd
outgoing dst port	fix	fix
incoming src port	fix	fix
incoming dst port	fix	rnd

Table 4. Properties of P2P traffic originating from PN nodes.

```
if in.TCP >> out.TCP and out.TCP.ACK:
if exist out.UDP:
if out.UDP.dstport == out.TCP.dstport:
tag flow as possible P2P
```

Incoming Connections For incoming P2P traffic the same rules as for outgoing traffic apply. However, the rate of signal and transfer packets is much lower thus thresholds must be chosen appropriately. Figure 7 shows P2P traffic of FN nodes.

For TCP transfer traffic that flows out of the PN there still must be signal packets that have

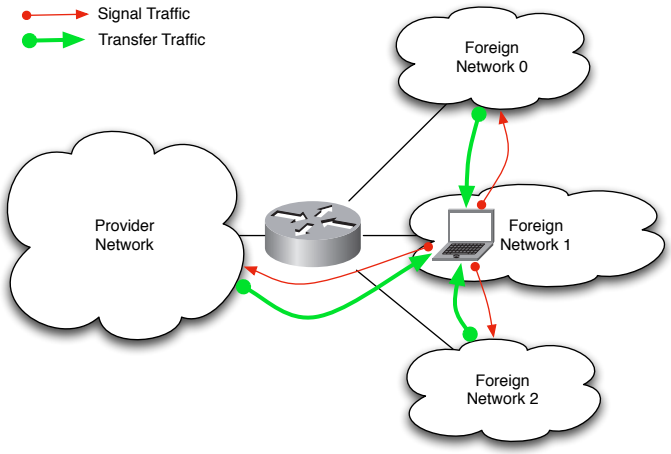

Fig. 7. Incoming P2P traffic.

been sent by the downloading FN node but the source port is different. Usually there are outgoing signal packets to the FN node from the uploading host where the outgoing source port equals the incoming destination port. But these messages might have a rather long time gap. So it is advisable

to check for both incoming and outgoing UDP packet for participating nodes. Table 5 shows the flow properties.
The following algorithm shows a simple scheme for identification similar to that of outgoing con-

properties	signal traffic	transfer traffic
outgoing src port	fix	fix
outgoing dst port	fix	rnd
incoming src port	fix	rnd
incoming dst port	fix	fix

Table 5. Properties of P2P traffic originating from FN nodes.

nections.

```
if out.TCP >> in.TCP and in.TCP.ACK:
if exist out.UDP or in.UDP:
if in.TCP.dstport == (out.UDP.srcport or in.UDP.dstport):
tag flow as possible P2P
```

One weakness of this scheme is that it relies on equal TCP and UDP port numbers. This is not the case for all P2P protocols. The ed2k clients usually define different port numbers for TCP and UDP traffic and thus can not be detected. So this is an easy exploit for P2P designers to prevent identification.

4.2 Identification Algorithm

The limitations in logic and memory on the target platform do not allow complex computations or data structures. A simple algorithm based on the assumptions presented in previous sections can be easily implemented in hardware and requires little amount of memory. The idea is to use a combination of simple behavior analysis and payload matching. Figure 8 shows a flow chart of the algorithm.

The algorithm is based on the assumption that P2P applications use both TCP and UDP on the same port to connect to or receive connections from different machines. For every {Address, Port} pair that engages in a UDP conversation a record of the form {ts, certain} is kept in a dedicated identification table. The timestamp ''ts'' is used to match the entry against a timeout to check if it is recent enough and can be considered for the currently observed packet. It is also needed for aging and removing old entries from the identification table. The field ''certain'' is a boolean value that indicates whether it is certain that the host is engaged in a P2P conversation on this specific port or not. Every host that has UDP traffic is added to the identification table with the packet's timestamp (or an abbreviation of it) and certain set to 0. These entries are marked as ''possible'' P2P candidates. Algorithm 1 explains the algorithm in more detail.

The signatures for pattern matching are up to four bytes short and few in number. Only a small set (on average 3) of signatures are needed for a set of P2P protocols. They can easily be stored in registers and matched in parallel to avoid stressing the memory and reduce timing. Note that signature matching works on each packet and can run independently from and parallel to the flow table updates. The identification algorithm itself then runs on not yet identified flows and uses information from the flow table (specifically the type).

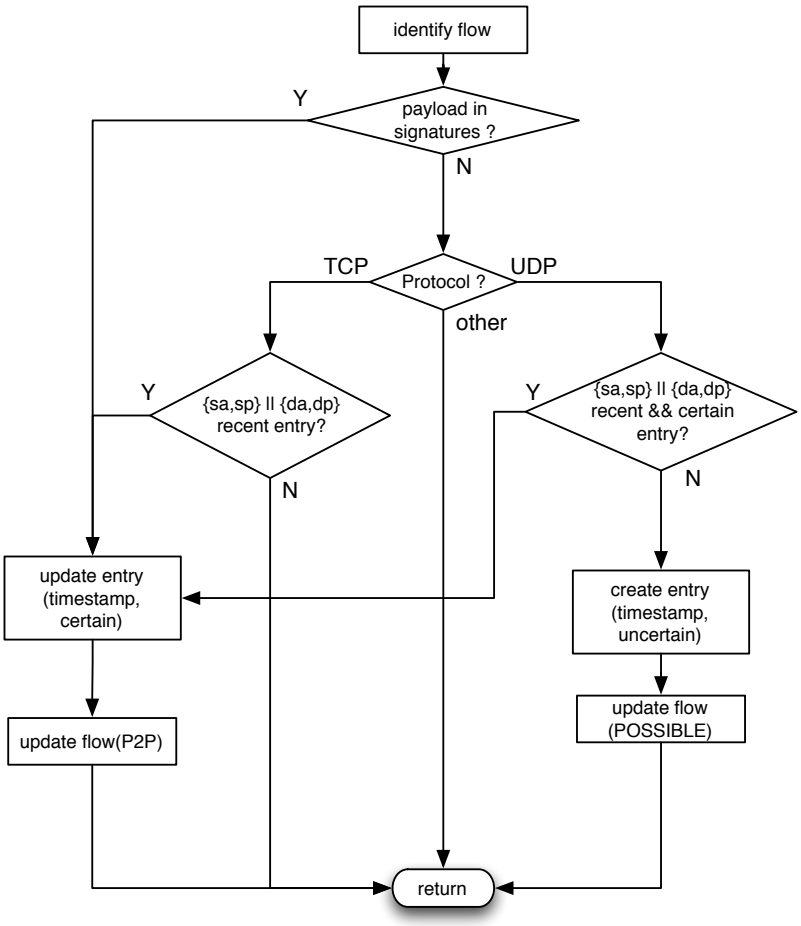

Fig. 8. Identification Algorithm flow chart.

Whether an entry is supposed to be recent depends on the state of the certain bit. An entry that is certainly identified as a P2P related entry (certain=1) will have a longer timeout. This is based on the observation that hosts engaging in P2P conversations are usually connected longer to the network and especially to down/upload partners. Thus, two different timeouts must be kept, one short timeout for entries that are not certain and one long timeout for entries that are certain. Listing 1 shows a possible pseudo c implementation of the timeouts and recent function.

It is crucial to find suitable values for the timeouts to prevent tainting innocent hosts. Machines may just have connected to the network and received the IP of a previously identified host. Thus a timeout that is set too high will falsely classify these hosts as P2P hosts. On the other hand, a timeout that is set too low will not find related P2P connections.

4.3 Pattern matching

For pattern matching only the first four bytes of the packet payload are needed. These four bytes are compared to a small set of well known P2P strings using exact matches. The signatures to compare to are shown in listing 2.

Algorithm 1: Flow identificaion Algorithm

```
Data: received packet P, flow flow
Result: flow type
if identified(flow) then return;
payload = getPayloadBytes(P);
if payload in signatures then
    updateEntries(src,sport, dst,dport, ts, 1);
    setType(flow, P2P);
    return;
ts = getTimestamp(P);
ip_p = getProtocol(P);
if ip_p == UDP then
    if recentCertain(src,sport, ts) or recentCertain(dst,dport, ts) then
        updateEntries(src,sport, dst,dport, ts, 1);
        setType(flow, P2P);
        return;
    updateEntries(src,sport, dst,dport, ts, 0);
    setType(flow, POSSIBLE);
    return;
if ip_p == TCP and (recent(src,sport, ts) or recent(dst,dport, ts)) then
    updateEntries(src,sport, dst,dport, ts, 1);
    setType(flow, P2P);
    return;
return ;
```

```
1  #define SHORT 5 // 5 secs short timeout (certain=0)
2  #define LONG (5*60) // 5 mins long timeout (certain=1)
3  bool recent (age, certain) {
4    time_t delta = certain ? LONG : SHORT;
5    return ((time()-age) <= delta);
6  }
```

Listing 1. Timeouts and recent function

```
1  const char *signatures[] = {
2    "\x13""Bit","d1:a","d1:r","d1:e",
3    "GNUT","GIV ","GND ","GO!!","MD5 ",
4    "\x27\x00\x00\x00","\xe3\x19\x01\x00","\xc5\x3f\x01\x00"
5  };
```

Listing 2. Signatures for pattern matching

Since exact matches are relatively cheap all patterns can be stored in 32 bit registers which can be compared to the 4 byte packet content in parallel. The logic or of the individual result indicates whether there was a match or not.

4.4 Data Structures

The identification algorithm only needs a relatively small table that holds records for every host that engages in a UDP conversation or which is part of a flow otherwise identified as P2P (e.g. by

DPI or signature matches). The table is a dictionary style structure indexed by a key and storing an associated value.

The key is a pair of {Address, Port}. The hash of the key is used to index the table. Listing 3 shows the key definition.

```
1  typedef struct key_s {
2    uint32_t addr;
3    uint16_t port;
4  } key_t;
```

Listing 3. Key type definition

The stored values can be reduced to the pair {timestamp, certain flag} and can both be encoded in a single value with a few bits. Since entries are kept only a short amount of time and are regularly removed during aging, there is no need to keep exact timestamps. Instead, only the time offset of the entry in milliseconds relative to the timestamp of the last aging cycle is stored. In addition, a small counter value is needed that keeps track of the number of aging cycles that the value survived.

```
1  #define d_age 30000 // 30 secs aging
2  #define d_short 5000 // 5 secs short timeout
3  #define d_long 5 * 60 * 1000 // 5 mins long timeout
4
5  typedef struct value_s {
6    uint8_t age : 5;
7    uint8_t cycles : 4;
8    uint8_t certain : 1;
9  } value_t;
```

Listing 4. Type definitions of the table's entries

Let o the offset in milliseconds, c the number of cycles, d_{age} the aging interval in millisecond. Then the real age of an entry in milliseconds can be derived using the following equation.

$$\texttt{realage} = o + c * d_{age}$$

When an entry is created at timestamp T_{ts} and the last aging occurred at T_{age}, then

$$o = \lceil (T_{ts} - T_{age}) \cdot 10^3 \rceil$$

On each aging cycle, the counter of the entry is incremented by 1 if the entry remains in the table. This way, the age of an entry can efficiently be encoded without the need to keep 64 bit timestamp values. The total number of bits needed for an entry e is

$$|e| = |o| + |c| + 1$$

where $|x|$ is the length of x in bits. The number of bits needed for the offset o and counter c fields are derived as follows

$$|o| = \lceil \log d_{age} \rceil$$

$$|c| = \lceil \log \frac{d_{long}}{d_{age}} \rceil$$

where d_{long} is the long timeout in milliseconds.

Listing 4 shows an example type definition of the value type using an aging interval of 30 seconds, a short timeout of 5 seconds and a long timeout of 5 minutes. The resulting table entry only needs 10 bits space. Thus, in a typical 32 bit wide memory three entries could be stored per memory word. So even with several millions of entries the table is only a few MiB in size.

Storing only the age and certain bit can lead to false positives if the hash function produces collisions for different keys. There are multiple ways of how to define a suitable hash function or even a set of hash functions and use multiple hashes to significantly reduce the probability of hash collisions and false positives. An overview is given in [16].

5 A guide for hardware implementation

The hardware implementation needs to make efficient use of the DFI engine and limit memory accesses to both the flow and identification tables. A typical implementation in hardware or on an FPGA vastly utilizes parallelism of multiple engines to reduce computation time.

In our proposed design we split the flow inspection into two parts. One is the pattern matching engine (further denoted PAT engine), the other is the DFI engine, that executes the flow identification algorithm. This separation is possible and useful since both parts operate on different information of the packet. The PAT engine uses only the first four bytes (further referred to as content) of the packets payload. The DFI engine needs the five tuple { src, sport, dst, dport, protocol} from the packet's headers as well as the timestamp of the packet. A more complex design with more resources can include more sophisticated DPI and Statistical Packet Inspection (SPI) engines that would run in parallel. The design is depicted in figure 9.

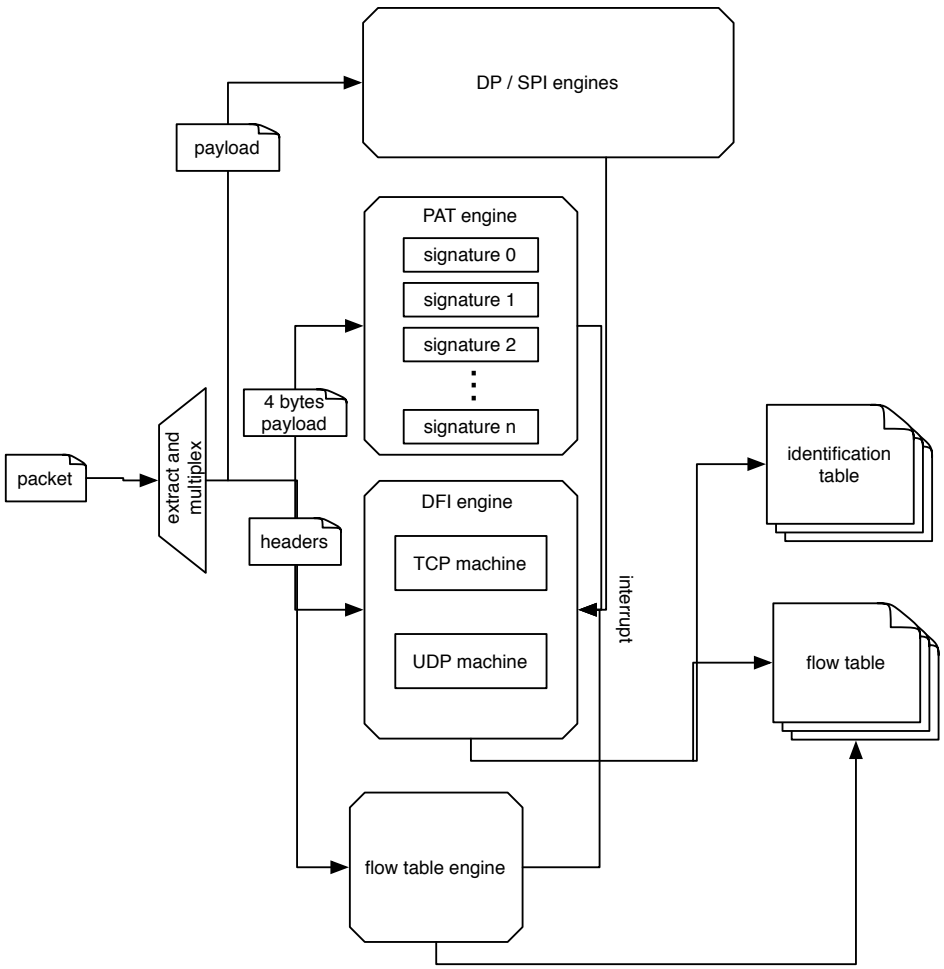

Fig. 9. Possible design of the engines.

Upon arrival of a packet, the header information and the content are extracted and sent to the PAT and DFI engines which can operate in parallel. The content is immediately processed by the string matching part of the DFI engine. The result will either be a match, in which case the P2P protocol can be identified, or a mismatch, in which case the protocol will be unknown. Note, that the string matching only needs to check a small set of equal sized strings, which can all run in parallel and which is much faster than full-fledged DPI.

Meanwhile, the header information is used by the DFI engine to access the identification table and retrieve the table entries if available. These entries should be copied to local registers for later usage, together with their indexes. Note, that it suffices to retrieve one of the entries. So if the first one is found there is no need for a second table access. The DFI module should have an interrupt wire that can cause an instant interrupt and reset the DFI states. Reasons for an interrupt could be, that the flow has been identified by PAT or DPI engines or that the flow table returns that the flow is already identified. Depending on the packet's protocol, either UDP or TCP, one of the identification machines is used. So the protocol field of the packet's header triggers execution of either the UDP or the TCP machine, but never both. The DFI engine needs access to both the identification as well as the flow table to query and update information. It also needs one internal register (alternatively two smaller ones) to mirror the two identification entries and keep the indexes. The TCP and UDP machines both can access these registers to read and compare the age and certain values of the entry. If identification leads to changes of the entries, they have to be written to the table, but only, if there has been a change or a new entry is created.

Logic Requirements

PAT engine. The PAT engine needs n 4 byte registers for n protocol signatures. The incoming 4-byte packet payload is compared to the n signatures in parallel. The comparators can be implemented with simple `and` gates which are `or`'ed on output to indicate a match or mismatch. This part of the identification algorithm can therefore be implemented with a small amount of logic. A very simplified implementation scheme can be seen in figure 10.

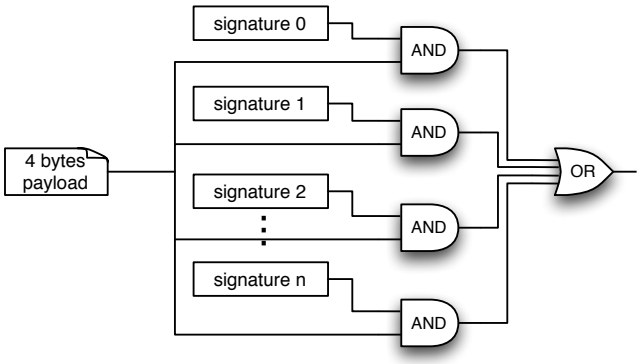

Fig. 10. Simplified PAT engine implementation.

DFI engine. First component needed are the registers to keep the hash values and the table entries as well as the logic to access the identification table. In addition the DFI engine needs to keep

state information about the two entries, which can be read and written during different stages of the identification. There are basically only two states that need to be kept, namely if the entry is updated (or created) or not. This can be modeled with a single flip-flop for the state `update`. The protocol header field must be compared against the constant values 6, 17 to select which path is executed. This can be implemented using less than 16 bit precision, since we are only interested in certain bits of the protocol field, namely those that must be set in case the one of the values are seen. So only those bits need to be `AND`'ed to get a match for either of the values. The output is fed into a multiplexor that selects which bits of the internal registers (age and certain value) are used to determine the flow type. The identification table needs only be updated if at least one of the entries is updated. The flow table only needs to be updated if the flow could be identified as either `P2P` or `NONP2P`, else not. This DFI engine also has small logic requirements. See figure 11 for a simplified and abstract implementation sketch.

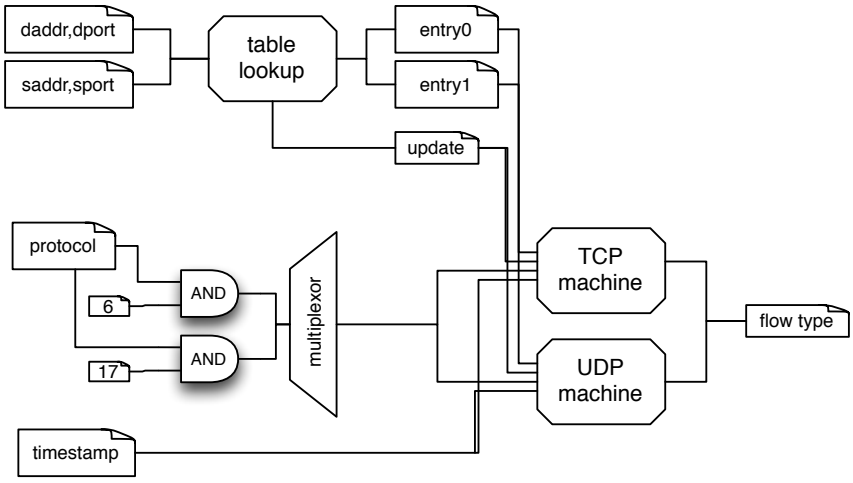

Fig. 11. Simplified DFI engine implementation.

6 Software Simulator

A prototype of the identification algorithm has been implemented in software to simulate and verify behavior and results of the developed scheme. The software simulation addresses the following issues.

- simulate identification behavior
  - using live data
  - using captured data
- verify identification results
  - computer aided
  - automatically

The simulator is written in c/c++ and uses libpcap[8] to process packet data either by live capture or using a pcap file. The simulator can also process verification files used to verify the identification results.

6.1 Classes and Data structures

Figure 12 shows the class diagram of the classes used by the picDFI simulator. It is pretty straight forward.

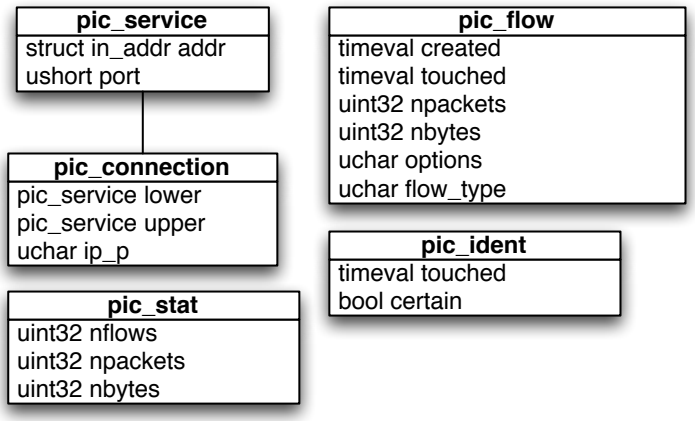

Fig. 12. Class diagram.

- `pic_service`: A single service or host, defined as the pair `addr, port`. Used both for connections as well as key to the identification table.
- `pic_connection`: A pair of two connected services and the connection protocol. It serves as the key to the flow table.
- `pic_flow`: The actual flow record. Keeps track of the times of creation and last touched as well as the number of packets, bytes, some options (which are unused) and the flow_type, which is initialized to `UNDEF`. The value of the flow table. The pair {`pic_connection`, `pic_flow`} are defined as the entries for the flow table.

[8] http://www.tcpdump.org

- `pic_ident`: Values for the identification table. The pair {`pic_service`, `pic_ident`} are entries to the identification table.
- `pic_stat`: Used to keep and compute statistical information.

The simulator uses a number of tables (c++ `maps`) to store certain information and associations.

Flow Table. Keeps track of all flows. {key, value} pair is {`pic_connection`, `pic_flow`}. For each arriving IP packet the corresponding flow is retrieved from the table and updated, or created if not available. The flow ages at certain time intervals, that is, all flows that are older than the last aging cycle are removed from the table. The default is 30 seconds, but the value can be defined by the user. Note that there is only one global aging timeout that also applies to the identification table.

Identification Table. The table used to keep the identification records needed to identify a flow. {key, value} pair is {`pic_service`, `pic_ident`}. The identification table also ages at the same time as the flow table.

Verification Table. Only used if a verification file is provided. Then it is the first table that is built. The verification table is practically a static second flow table. It maps connections to flow records. However, there are no updates or aging to the table during runtime. The flow type of the flow records in the verification table are considered to be the accurate or real flow type. Thus every flow that is identified is compared to the corresponding flow in the verification table. Thus it can be determined if the flow has bee identified correctly or not.

Statistics Table. Used to keep and compute statistics of flow types and their corresponding number of flows, packets and bytes.

6.2 Simulation Flow

The simulator uses `libpcap` to capture packets from the physical interfaces and to read/write pcap capture files. It analyses the packet headers and takes appropriate actions depending on the layer protocols. These actions are performed by so called `handlers` that operate on different layers and stages. The following steps are performed.

1. Parse command line options.
2. If a verification file is provided, parse the file and build verification table.
3. Initialize tables.
4. Get next packet from interface or pcap file.
5. If number of passed seconds is greater than aging timeout age the flow table then the identification table.
6. Update the flow table with packet information.
7. If needed run the identification algorithm.
8. Goto 4.

Every time a flow is removed from the flow table, either through aging or some other means, it triggers certain actions defined by `handlers`. The statistics table is updated using the flow's informations. If a verification table is available the flow is verified. The program runs as long as there are still packets in the file, or in case of capturing directly from the interface, as long as no interrupt signal is received. Then it terminates the packet capturing and analyzation, flushes the flow table and prints the final results on `stdout`.

6.3 Verification

The problem with verification is, that it can hardly be done automatically. To verify the correctness of the identification algorithm there must be a way of reliably know the real type of every flow. But the lack of existence of a reliable identification scheme, which is able to accurately identify every flow prevents automatic verification. The only way to verify the identification algorithm is using a controlled environment where global knowledge about every communication is absolute. This is possible in a lab environment but impractical on real world data. Thus, the only way of 'verification' is actually to compare the results to those of other identification mechanisms. An example is 'OpenDPI' from Ipoque. It is an open source DPI engine that can do protocol identification based on packet payloads. However, since it is prone to error in presence of encrypted and obfuscated protocols. As such 'OpenDPI' is not a reliable source of verification. It can be shown that it falsely classifies a large number of flows under certain circumstances. In fact, the picDFI algorithms has a better True Positive rate than OpenDPI.

Verification is done using a 'verification file'. It records plain text comma separated serializations of flow records and has the following fields:

`la,lp,ua,up,protocol,created_ts,touched_ts,npackets,nbytes,type\n`

Flow records are written in plain text and terminated with a newline. The implemented verification process actually only uses the flow ID and type for verification and neglects time stamp information. As a result, flows that have the same ID but appear over different time spans and also have different times are overlooked. More specifically, only the latest flow (as appears in the file, not by time) is considered. However, the file format is design to support also time based verifications.

Verification computes the following statistics.

- `na_positive`, could not verify, identified p2p
- `na_negative`, could not verify, identified nonp2p
- `true_positive`, identified and verified p2p
- `false_positive`, identified p2p, verified nonp2p
- `true_negative`, identified and verified nonp2p
- `false_negative`, identified nonp2p, verified p2p

6.4 Evaluation

Preliminary simulations show solid identification behavior of over 90 % accuracy and better quality than 'OpenDPI' especially in the presence of encryption. However, this is based on small captured data within a controlled environment. Though this is a good means of reliably verify the identification quality it is not representable for large-scale router traffic. Further tests and evaluation are performed in a real corporate setting after final adjustments and are subject to future work.

7 Future Work

Traffic identification methods used today rely on local information and mostly run independently on individual routers. They try to identify the protocol using DPI and behavioral algorithms as well as some statistical approaches. Though this has been proven to suffice for a lot of protocols (especially plain text based and well known protocols) it is prone to obfuscation, encryption and hiding. The locality-based approaches lead to high resource requirements on edge routers and a waste of resources. They also do not gain global knowledge about network usage like the creation/ existence of overlay networks.

Another aspect to consider is the granularity of traffic identification and the actual identification goal. Contemporary classification systems usually try to identify the exact application layer protocol. However, in many applications, this might not be the best or even desired choice. The application layer protocol is not necessarily an indicator for the actual type of conversation. As an example, HTTP can be used for ordinary web browsing as well as large scale file downloading. This identification goal can also easily be exploited for hiding the actual communication intend.

Instead of relying on local information that is based on well known signatures and fingerprints future traffic classification should gather and aggregate network wide information about global application behavior that is unlikely to change. Applications have specific behavior patterns and requirements of network resources. VoIP and other streaming protocols for example always need constant bit rates to provide a reasonable user experience. File sharing applications will always need an overlay network with lots of connections to multiple nodes. In general a set of behaviors and requirements can be identified that are unique to specific application domains.

This meta information about application behaviors and their network resource requirements can be used to devise a general *taxonomy of internet traffic* which can then be used to classify any network traffic into specific application domains that are independent of protocol and application implementation details.

Bibliography

[1] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *ICS '02: Proceedings of the 16th international conference on Supercomputing*. New York, NY, USA: ACM, 2002, pp. 84–95.

[2] M. Castro, M. Costa, and A. Rowstron, "Debunking some myths about structured and unstructured overlays," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 85–98.

[3] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy, "Transport layer identification of P2P traffic," in *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2004, pp. 121–134. [Online]. Available: http://portal.acm.org/citation.cfm?id=1028788.1028804

[4] D. Oneil, H. J. Kang, J. Kim, and D. Kwon, "Transport layer identification of P2P super nodes," 2004, available from http://www-users.cs.umn.edu/~jinohkim/docs/supernode.pdf.

[5] S. Sen, O. Spatscheck, and D. Wang, "Accurate, scalable in-network identification of P2P traffic using application signatures," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM, 2004, pp. 512–521.

[6] T. Karagiannis, A. Broido, N. Brownlee, K. C. Claffy, and M. Faloutsos, "Is p2p dying or just hiding?" in *Proceedings of the GLOBECOM 2004 Conference*. Dallas, Texas: IEEE Computer Society Press, November 2004.

[7] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos, "File-sharing in the internet: A characterization of p2p traffic in the backbone," *University of California, Riverside, USA, Tech. Rep*, 2003.

[8] W. Ngiwlay, C. Intanagonwiwat, and Y. Teng-amnuay, "Bittorrent peer identification based on behaviors of a choke algorithm," in *AINTEC '08: Proceedings of the 4th Asian Conference on Internet Engineering*. New York, NY, USA: ACM, 2008, pp. 65–74.

[9] B. Cohen, "Incentives build robustness in BitTorrent," 2003.

[10] J. Liang, R. Kumar, and K. Ross, "Understanding kazaa," 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.6301

[11] P. Biondi and F. Desclaux, "Silver needle in the Skype," Presentation at *BlackHat Europe*, Mar. 2006. [Online]. Available: http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf

[12] F. Desclaux and K. Kortchinsky, "Vanilla Skype," 06 2006.

[13] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli, "Revealing skype traffic: when randomness plays with you," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 37–48, 2007.

[14] D. Schmucki, "Unkooperative VoIP-Flowerkennung und Gegenmassnahmen (Uncooperative VoIP flow detection and countermeasures)," Master's thesis, University of Konstanz, Sep. 2008.

[15] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 53–65.

[16] T. Zink, "Packet forwarding using improved bloom filters," Master's thesis, University of Konstanz, 2009. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-131409