International Journal of Cooperative Information Systems

IDENTIFICATION AND MANAGEMENT OF SESSIONS GENERATED BY INSTANT MESSAGING AND PEER-TO-PEER SYSTEMS

ZHONGQIANG CHEN

Yahoo! Inc., Santa Clara,CA 95054, E-mail: zqchen@yahoo-inc.com

ALEX DELIS*

University of Athens, Athens, Greece 15771, E-mail: ad@di.uoa.gr

PETER WEI

Fortinet Inc., Sunnyvale, CA 94085, E-mail: pwei@fortinet.com

Received (June 19th, 2006)
Revised (April 17th, 2007)

Sessions generated by Instant Messaging and Peer-to-Peer systems (IM/P2Ps) not only consume considerable bandwidth and computing resources but also dramatically change the characteristics of data flows affecting both operation and performance of networks. Most IM/P2Ps have known security loopholes and vulnerabilities making them an ideal platform for dissemination of viruses, worms, and other malware. The lack of access control and weak authentication on shared resources further exacerbates the situation. Should IM/P2Ps be deployed in production environments, performance of conventional applications may significantly deteriorate and enterprise data may be contaminated. It is therefore imperative to identify, monitor and finally manage IM/P2P traffic. Unfortunately, this task cannot be easily attained as IM/P2Ps resort to advanced techniques to hide their traces including multiple channels to deliver services, port hopping, message encapsulation and encryption.

In this paper, we propose an extensible framework that not only helps identify and classify IM/P2P -generated sessions in real time but also assists in the manipulation of such traffic. Consisting of four modules namely, *session manager, traffic assembler*, IM/P2P *dissector*, and *traffic arbitrator*, our proposed framework uses multiple techniques to improve its traffic classification accuracy and performance. Through fine-tuned splay and interval trees that help organize IM/P2P sessions and packets in data streams, we accomplish stateful inspection, traffic re-assembly, data stream correlation, and application layer analysis that combined boost the framework's identification precision. More importantly, we introduce IM/P2Ps "plug-and-play" protocol analyzers that inspect data streams according to their syntax and semantics; these analyzers render our framework easily extensible. Identified IM/P2P sessions can be shaped, blocked, or disconnected, and corresponding traffic can be stored for forensic analysis and threat evaluation. Experiments with our prototype show high IM/P2Ps detection accuracy rates under diverse

*Partial support was provided by a European Social Funds and National Resources Pythagoras grant with No.7410 and the Univ. of Athens Research Foundation.

settings and excellent overall performance in both controlled and real-world environments.

Keywords: Instant Messaging; Peer-to-peer Overlay Networks; Analyzer-based Session Identification; Traffic Arbitration; Classification Accuracy.

1. Introduction

Steady improvements on processing units, storage options, and network bandwidth in conjunction with the need to deliver "rich" data have paved the way for the emergence of Instant Messaging (IM) services and Peer-to-Peer systems (P2Ps) [17,16,66,47,44]. Such IM/P2Ps not only facilitate instant communications, data exchange, and resource sharing, but also help reverse the "asymmetric" nature of the conventional web services established on the client/server paradigm [65]. Currently, IM/P2Ps constitute the dominant source of Internet traffic and consume a large fraction of available network bandwidth [65,31,45]. More than 100 million users from *AOL, MSN, Yahoo!* and *ICQ* generated 900 million messages every day in 2003 [14]. By the end of 2006, the IM population was expected to exceed 250 million users with 60% of real-time communications involving voice, text, and video [27]. On the other hand, *KaZaA*, a key P2P player, enjoys a strong following with 3 million online users on average (up to 5 million on peak) and is downloaded approximately 2 million times a week worldwide [41]. Similarly, *eDonkey* and *LimeWire* P2Ps have about 1 and 0.3 million online-users respectively [19,46]. As a P2P -based voice over IP (VoIP) application, *Skype* attracted 21.3 million users in 2006 and it is estimated that another 12 million will join in 2007 [4].

Network sessions generated by IM/P2Ps play a significant role in today's Internet as a major bandwidth consumer. Measurements in a backbone network showed that P2Ps created up to 50% of the traffic with an additional 18% of unidentified packets, possibly having the same origin [24]. Apparently, Internet traffic has shifted from "pure" text/image *WWW*-documents to instant messaging and resource-sharing dominated by audio, video, and media streams [65,68,31]. In a typical IM/P2P session, two peers reciprocate in terms of traffic generation and help maintain the continuity of system operations, thereby consuming about the same bandwidth in both directions; this is in contrast to the asymmetric bandwidth use of traditional Web services. In addition, IM/P2P sessions may require upto 90 times more bandwidth and many more concurrent connections than simple *HTTP* requests [31,67].

Users have often considered IM/P2Ps harmless and use them to share private or even sensitive data [56]. However, it is established by now that a large number of IM/P2P implementations suffer from deficient handling of input validation process, boundary conditions, access authorization, and race conditions [36,42]. All these security holes essentially transform IM/P2Ps to ideal channels for the rapid spread of viruses, worms, and greyware [36,71]. Furthermore, some IM/P2Ps are even bundled with adware, spyware, and keyloggers. For instance, analysis of *LimeWire* traffic for a period of 45 days revealed 95 distinct types of malware [36]. Similarly, it was reported that about 12% *KaZaA* clients are infected by various viruses [71] and approximately 50% of executable files downloaded through *KaZaA* contain malicious code and greyware such as *Gator*, *Cydoor*, and *SaveNow* [63].

Evidently, IM/P2Ps may reduce productivity by affecting regular network operation and it becomes imperative that organizations be able to detect, restrict, or even block such traffic [56,52]. To avoid detection by security systems, IM/P2Ps often try to "hide" their traffic with sophisticated techniques including port hopping, message encapsulation, and strong data encryption [10,25]. For instance, more than 38% of sessions in *KaZaA* use dynamically generated ports instead of its registered standard TCP port 1214 rendering port-based session identification a poor choice [45]. *MSN* and *Yahoo!* IMs can "camouflage" their traffic within Web data underlying the need for application-layer protocol dissection to improve traffic classification accuracy [49]. As IM/P2Ps mainly deliver their services on the stream-based TCP transport mechanism, packet-based traffic detection systems become entirely ineffective. A number of recent P2Ps releases including *Skype* are specifically designed to evade traffic filtering, prevent eavesdropping, and ultimately bypass all security control using strong cryptographic techniques [11,56]. As new-breed IM/P2P protocols are continually introduced and variations of existing ones often appear, a good fraction of traffic may go undetected should conventional fixed-port or packet-based traffic identification and detection methods be used [48].

In this paper, we propose an extensible framework that identifies IM/P2P sessions in real-time fashion so that we can improve traffic control and enhance IM/P2P stream manipulation. In direct contrast to existing intrusion detection systems (IDSs) that function off-line, we design our framework to operate "inline". In doing so, the framework intercepts, inspects, and classifies network traffic in real-time. To detect message encapsulation, port hopping, and other evasive techniques, our framework resorts to a combination of techniques including stateful inspection, traffic re-assembly, data stream correlation, layer-7 or application-level analysis, and session-based pattern matching. A unique feature of our approach is the use of "plug-and-play" analyzers for specific IM/P2P streams; they help analyze and detect unique stream characteristics and their use in the context of the framework is extensible. As new versions and types of IM/P2Ps appear, our framework is extended accordingly once corresponding analyzers become available often through reverse engineering. Manipulation operations on identified IM/P2Ps traffic include alert generation, traffic shaping, stream blocking, and/or termination of connections. A logging mechanism is also featured to stage-in-disk IM/P2P sessions for auditing and forensic analysis purposes if desired. We have carried out detailed stress tests using synthetic data streams in controlled environments and experimented with live traffic in real-world settings. Our results show that the proposed framework demonstrates excellent performance in detecting IM/P2P sessions under diverse workloads without raising false positives/negatives; at the same time, it imposes minimal overhead to examined application streams.

The rest of the paper is organized as follows: Section 2 outlines key characteristics of IM/P2Ps . Section 3 presents the key features of our framework and a number of IM/P2P analyzers are discussed in Section 4. Section 5 discusses our experimental effort and presents our main findings. Section 6 presents related work and concluding remarks can be found in Section 7.

2. Unique Characteristics of IM/P2P Systems

Instant Messaging systems (IMs) offer exchange of information and track status of active users [56]; using interconnections among IM-servers, they also provide real-time voice/text conversation, file transfers, and on-line gaming. Existing systems including the *AOL Instant Messenger (AIM)*, *Yahoo!* and *MSN* messengers use proprietary protocols making impossible for users to simultaneously access multiple IM-services through a single interface. We expect this trend to continue despite of various efforts on IM standardization [61,29,60,28,35]. On the other hand, P2P systems now offer a wealth of multimedia services with their nodes acting as either producers or consumers of data/resources and often organized in hierarchies according to their CPU capabilities, bandwidth, and availability; *ultra-peers* help balance load, stabilize networks, and improve scalability [42,37,23]. IM and P2P systems do have overlapping features and by integrating those, *Skype* has clearly benefited and has emerged as a very popular option in the field. In this section, we outline the unique features of IM/P2Ps and point out the challenges needed to overcome in order to identify pertinent network flows.

2.1. *Diverse Behavior of* IM/P2P*s Services*

Services that used to be offered in isolation such as voice chat, video communication, sharing of diverse type data-objects, and mail messaging are now provided by IM/P2Ps in an integrated fashion. IM/P2Ps can also demonstrate polymorphism in realizing a single service. For instance, file transfers can be conducted using pipelining, batching, or multi-source swarmed downloading. To accommodate this diverse set of services, IM/P2Ps often specify their proprietary formats for message exchanges; such formats may not be honored by the underlying transport services as a single IM/P2P message may stride over multiple TCP/UDP packets or multiple messages may be packed into a single transport packet. For instance, the *AIM/Oscar* protocol specification states that a number of *AIM* commands can be shipped as the payload of a single transport packet [76]. The field `payload-length` carried by every *Gnutella* message helps the restoration of application message boundaries [12,42]. Should routing devices support different maximum segment sizes (MSS), such devices may also yield inconsistencies between IM/P2P messages and transport packets.

As portions of IM/P2Ps messages are often generated dynamically and pushed into underlying protocol stacks on-the-fly, mapping discrepancies between application messages and corresponding transport packets are also formed. For instance, when a *Yahoo!* client resides behind a firewall, it encapsulates its traffic in *HTTP* streams. The *HTTP* header consists of a series of "key/value" pairs while the body of the message carries content significant only to application. When such an *HTTP* message is generated by a *Yahoo!* client, its header fields have fixed values and are quickly created. On the other hand, the body of the message contains session-related information and is dynamically generated by users. The latter implies that time delays in the delivery between header and body to the transport service may generate a different network packet sequence than its application counterpart. Table 1 presents two different *Yahoo!* client login sessions via TCP port 80. The *Yahoo!* clients in both sessions are configured to have *firewall with no proxy* type of connection [a] and use version 8.1.0.209. In the first session, the *HTTP* message head

pkt	*dir*	*message*	*description*
version: 8.1.0.209; protocol: TCP; server (S): 216.155.194.191:80; client (C): 192.168.5.36:1229;			
1	C→S	POST /notify/ HTTP/1.1 Content-Length: 47 YMSG|00 0B 00 00 00 1B 00 57 00 00 00 00 00 00 00 00 31 C0 80 73 ...|	standard HTTP method: "POST"; size of "data" section Yahoo! Messenger: login request;
2	S→C	HTTP/1.0 200 OK Content-Type: text/plain |01 00 00 00| YMSG |00 00 00 00 00 60 00 57 00 00 00 01 7A 60 ...|	reply from *Yahoo!* Messenger server data type Yahoo! Messenger: server reply;
version: 8.1.0.209; protocol: TCP; client (C): 192.168.5.40:3839; server (S): 216.155.194.191:80;			
1	C→S	POST /notify/ HTTP/1.1 Cookie: Y=v=1&n=ann72 ...	header of "POST" request no data section in this message
2	C→S	YMSG|00 0B 00 00 00 24 00 57 00 00 00 00 7A 60 00 00 31 C0 80 73 ...|	data are included in this message
3	S→C	HTTP/1.0 200 OK Content-Type: text/plain |01 00 00 00| YMSG |00 00 00 00 00 5C 00 57 00 00 00 01 7A 60 00 00 31 C0 80 ...|	header of HTTP reply embedded Yahoo! message Yahoo! client is authenticated

Table 1. *Yahoo!* IM traffic embedded in *HTTP*-streams where boundary inconsistencies may occur

and body are packed within a single TCP packet; this turns out to be the norm in our traffic analyses. However, we sometimes observe sessions whose header and body are placed into two TCP packets; this is the case with the second session of Table 1. In rare occasions, we encounter sessions that have the *HTTP* body spread over multiple TCP packets. The inconsistency in boundaries between application messages and transport packets leads to the conjecture that packet-based traffic identification methods inevitably generate false negatives.

IM/P2Ps may also demonstrate diverse behavior due to their configuration and the network environment they operate in. For example when the *Skype* "*automatic login*" option is not set, a specific user login-session based on TCP transport service is established generating unique patterns in traffic. The latter can be exploited to identify the session [5,20]. On the other hand when "*automatic login*" is enabled, the authentication is performed by supernodes (SNs) following an entirely different

[a]the option is under menu item *messenger/preferences/connection*.

approach. Similarly, the *Yahoo!* IM embeds its traffic in *HTTP* data sections when firewalls are present. Without firewalls in place, *Yahoo!* follows its native protocol even when its server listens to TCP port 80. Thus, traffic streams have to be checked against both IM/P2Ps native protocols and alternative hosting protocols such as *HTTP* and *HTTPS* to avoid false negatives.

2.2. *Multiple Protocols in* IM/P2P*s Service Realization*

To improve their reliability, IM/P2Ps frequently implement services with multiple transport protocols. In this regard, the *MSN*-messenger uses TCP connections for login and authentication while for file transfers and audio/video-conferencing uses TCP(port 6891) and UDP(ports 13324/13325) respectively. Even the same service can be delivered in multiple transport options. For instance, *Skype* determines the presence and type of Network Address Translation (NAT) devices using UDP when making a phone call. If firewalls block all UDP traffic, clients behind security devices are still functional as *Skype* provides its services over TCP as well. Also in most P2Ps , hosts utilize multiple mechanisms to access networks and manage services. For example, to join a *FastTrack* network [b], a host first probes the network by dispatching UDP-requests to a subset of cached super-nodes and may ultimately resort to TCP if no UDP-reply is received. Table 2 shows excerpts of traffic generated by a peer attempting to access a *KaZaA* network. At IP 192.168.5.143, the peer initially UDP-*pings* a subset of supernodes (Table 2 only shows 4 of them, i.e., packets 1–4). It then tries to establish TCP connections with the same set of supernodes as packets 5–7 show. Among the supernodes in question, the one at 66.130.102.247:2713 accepts the request and the peer joins *KaZaA* successfully as packets 8–11 indicate. P2Ps systems such as *KaZaA* and *Overnet* perform their search, retrieval, load-balancing and signalling operations over either TCP or UDP.

To avoid a single point of failure, IM/P2Ps often replicate features either physically or functionally. *Skype* clients often use login servers (LSs) to get authenticated. If a client finds all its TCP/UDP connections to LSs blocked, it can still join the network by having the authentication procedure performed or relayed by another node. Similarly, IMs often deploy multiple servers so that a single service is supported by geographically dispersed nodes. In this regard, the *AIM* login and authentication are provided by multiple servers that follow the *OSCAR* application protocol [c], while functionalities regarding locations of buddies/users and message exchanges are realized through multiple servers that run the *BOS (Basic OSCAR Services)* protocol. The *MSN*'s IM follows a similar approach. Finally, a number of P2Ps employ both TCP and UDP for different stages of a single service. For example, *Overnet*'s protocol consists of location determination of content and download

[b] that is *KaZaA*, *Grokster*, or *iMesh*
[c] the Open System for Communication of AOL in Real-time.

#	*src. (IP:port)*	*dst. (IP:port)*	*proto*	*payload*	*description*
1	192.168.5.141:3037	66.41.187.3:2202	UDP	|27 00 00 00 A9 80 4B 61 5A 61 41 00|	UDP ping
2	192.168.5.141:3037	66.71.66.69:2289	UDP	|27 00 00 00 A9 80 4B 61 5A 61 41 00|	UDP ping
3	192.168.5.141:3037	65.33.247.155:2936	UDP	|27 00 00 00 A9 80 4B 61 5A 61 41 00|	UDP ping
4	192.168.5.141:3037	66.130.102.247:2713	UDP	|27 00 00 00 A9 80 4B 61 5A 61 41 00|	UDP ping
5	192.168.5.141:29280	66.130.102.247:2713	TCP	(SYN)	use TCP
6	192.168.5.141:29281	66.71.66.69:2289	TCP	(SYN)	use TCP
7	192.168.5.141:29282	65.33.247.155:2936	TCP	(SYN)	use TCP
8	66.130.102.247:2713	192.168.5.141:29280	TCP	(SYN|ACK)	response
9	192.168.5.141:29280	66.130.102.247:2713	TCP	(ACK)	confirm
10	192.168.5.141:29280	66.130.102.247:2713	TCP	|0D 82 F6 68 CE 79 CF 7E 95 13 D8 A9|	handshake
11	66.130.102.247:2713	192.168.5.141:29280	TCP	|B7 5E D8 B3 28 94 04 29 EC 60 ...|	response

Table 2. Traffic generated by a *KaZaA*-peer (v3.2.5) during the process of joining of the network

of requisite files; the former uses UDP while the latter TCP. It is thus evident that both TCP and UDP transport protocols frequently participate in multiple IM/P2Ps phases to realize services. Should we be able to manipulate the ensued IM/P2P traffic, both TCP and UDP types of packets have to be scrutinized to help identify and classify traffic.

2.3. *Port Hopping and Message Encapsulation*

AIM, MSN, Yahoo! IMs register their native ports at 5190, 1863, and 5050 respectively, while P2Ps including *KaZaA, Gnutella*, and *eDonkey* correspondingly operate at default ports 1214, 6346, and 4661. However, IM/P2Ps often resort to dynamic port assignment to provide flexibility, making user intervention and manual configuration unnecessary; in addition, dynamic ports can avoid traffic shaping and manipulation by security devices that deploy port-based filters. For instance in *KaZaA*, only 20% of super-nodes use the registered TCP port of 1214 [45]. Furthermore, IM/P2Ps also employ port sweeping techniques termed *port hopping* to help session establishment between entities. In port hopping, a host attempts to connect a remote node over a set of ports systematically until the connection is established successfully. Clearly, for the same service, the actual ports used by remote nodes in the resulting sessions may widely vary over different hosts and/or time. For instance, with the help of the locally maintained list of supernodes, a *Skype* client first tries to contact a supernode on the port specified in the list. As a fallback mechanism, the client also attempts to connect the supernode over ports 443 and 80 as well. Similarly, the *MSN* IM permits clients to use TCP-port 80, while *Yahoo!* IM allows for the "scanning" of ports 23, 80, 25, 119, and 20, should the default 5050-port for authentication fails. In a similar fashion, *AIM*-clients attempt to reach servers over ports 20, 21, 23, and 80 in turn, should their default 5190 becomes inaccessible.

Some firewalls restrict the port ranges even for connections initiated by hosts within protected zones. To this effect, IM/P2Ps use sweeping to determine the port ranges blocked by firewalls. More specifically, a *Skype*-client randomly chooses a TCP port in the range of 1026 and 1040 while attempting to establish a connection with a super-node (SN). If the connection fails, the client increments the number of its attempted port and the process is repeated until a connection is established. Although most connections in port hopping fail due to incomplete TCP three-way handshakes, we have to develop mechanisms to identify IM/P2P traffic going through successfully via ports selected with sweeping. Obviously, port hopping and sweeping strategies in IM/P2Ps help provide the same service over seemingly arbitrary ports. It is projected that most IM/P2Ps are expected to use port hopping [45,39,38], rendering the identification of pertinent connections a challenge.

The situation is further exacerbated when ports usurped by port hopping happen to be used by *HTTP* and *HTTPS*. Here, IM/P2Ps resort to message encapsulation techniques to embed their messages to *HTTP/HTTPS* messages instead of using their native protocols. In the case of *Yahoo!* IM client working behind a firewall, the security device may block all traffic except that which is destined to TCP port 80. Should we configure *Yahoo!* IM to use the *firewall with no proxy* type of connection, *Yahoo!* IM encapsulates its stream into *HTTP*-messages as Table 1 shows. Once a TCP connection is established between the *Yahoo!* client and the server, the client exercises a *POST*-method consisting of *HTTP*-header and pertinent data. All header keys, such as *Host* and *Content-Length* are *HTTP*-defined; the server reciprocates with a standard *HTTP OK* message (Table 1). The rationale here is to foul IDSs/IPSs and AVs so that the latter allow the traffic through as benign Web-streams. Unless *HTTP* data portions are inspected, IM/P2Ps sessions with message encapsulation will go undetected and false negatives are unavoidable. To overcome this limitation, layer-7 [d] data stream analysis has to take place. Moreover, as IM/P2Ps hosts use proxy services including *HTTP/HTTPS* proxies and *SOCKS* to successfully tunnel their message through security devices, the need for layer-7 inspection on IM/P2Ps traffic becomes pressing [56].

2.4. *Mechanisms to Penetrate Security Systems*

To mitigate the depletion of IP address space, NAT devices are ubiquitously deployed in the Internet. The NAT asymmetric addressing and connectivity does affect IM/P2P applications as the latter may involve responders lacking a consistent and permanent IP address. Similarly, many one-way filters deployed in firewalls block connections initiated by hosts outside protected networks, making it impossible for hosts behind firewalls to participate as recipients to sessions. Typical techniques employed by IM/P2Ps to "penetrate" both firewalls and NATs include rendezvous-relay, connection reversal, and UDP hole-punching. In the rendezvous relay service, any communication between clients is relayed via super nodes (SNs). In the con-

[d] also known as application-oriented or "deep" inspection.

nection reversal scheme, the recipient of a session requested by its counterpart ultimately becomes the initiator of the intended session. This occurs after the recipient is informed about the specifics of the session to be established through a super node. When a NAT device maintains a consistent mapping between "private IP/port" and "public IP/port", the hole-punching technique can be used to establish UDP sessions between two entities behind NATs (same or different). Both entities can obtain each other's publicly visible IP address with the help of a super node (SN) and then initiate the UDP connection simultaneously and directly between them. Among others, *Skype* as well as *Yahoo!* and *MSN* IMs employ such penetration techniques to provide services for sites found behind NAT devices.

To discover the presence and types of NATs and firewalls between a host and the public Internet, IM/P2P applications typically employ techniques similar to Simple Traversal of UDP over NAT (*STUN*) [62]. *STUN* allows a host to determine the presence and type of a NAT device via a coordinated message exchange with a *STUN* server. The latter responds with messages containing the source IP address/port of a request. As the *STUN* server can only observe the requestor's publicly visible address, the requestor can determine both NAT presence and type by comparing its local address with that in the reply. Table 3 outlines *Skype*'s UDP probing to determine the presence of NATs. In this setting, no firewall is installed but a NAT device is in place with *Skype* running version 2.5.0.130. Each *Skype* UDP message consists of a header containing a frame ID (2 bytes) and a function type (1 byte) fields as well as a body whose size varies and in most cases its content is obfuscated with the help of *RC4* encryption method.

#	*dir*	*len*	*payload*	*description*
	protocol: UDP; SC: 10.2.42.169:16803; SN1: 64.246.48.23:33033; SN2: 76.0.43.219:6800			
1	SC→SN1	20	|47 3E 02 D4 46 BA B3 76 B3 C3 5B ...|	frame ID: 0x473E; func. type: 0x02, obfuscation; init vector: 0xD446BAB3; CRC32: 0x76B3C35B;
2	SN1→SC	11	|47 3E 27 42 23 FE 40 D3 33 0C 9A|	frame ID: 0x473E; func. type: 0x27 & 0x0F = 0x07, NACK; src: 0x4223FE40 (66.35.254.64); tag: 0xD3330C9A (211.51.48.154);
3	SC→SN1	25	|47 3E 23 01 D3 33 0C 9A 40 F6 30 17 76 B3 C3 5B 7A ...|	frame ID: 0x473E; func. type: 0x23 & 0x0F = 3; retrans.; tag:0xD3330C9A SN:0x40F63017 (64.246.48.23); CRC32: 0x76B3C35B;
4	SN1→SC	53	|05 A4 02 72 9D A6 0D 72 1B DC 36 ...|	frame ID: 0x05A4; func. type: 0x02, obfuscation; length = 53 indicates redirection
5	SC→SN2	27	|47 40 02 A0 F0 9C 99 5E 39 54 E4 6F FB 57 3B 49 97 ...|	SC contacts another SN; frame ID: 0x4740; func. type: 0x02, encryption used; init vector: 0xA0F09C99; CRC32: 0x5E3954E4;
6	SN2→SC	18	|8F 6E 02 4A BF 25 79 BD 0A 4F 4B BE 3E 2B F5 A4 D6 1A|	frame ID: 0x8F6E; func. type: 0x02, encryption; length = 18 acts as confirmation of accepting SC; SC joins the network

Table 3. UDP probe procedure in *Skype* v.2.5.0.130

The heavy-weight probe sequence formed by packets 1 to 4 of Table 3 is used to determine the presence and type of NAT. The initiating UDP message from the client (SC) contains the checksum derived from source/destination IP addresses. Due to NAT, the client's private IP (i.e., 10.2.42.169) is invisible to the super-node

(SN), causing the SN-computed checksum with SC's public address to be different from that in packet 1. Therefore, SN returns a negative acknowledgment message (*NACK*) as packet 2 in Table 3 shows. The SC's publicly visible IP address contained in the SN-originated *NACK* message allows SC to realize the existence of a NAT device and the mapping between private address 10.2.42.169 and public address 66.35.254.64. Once packets 1 and 2 have been exchanged and SC derives both NAT presence and type, penetration techniques can be used to facilitate IM/P2P services.

NAT devices typically define the lifetime for their mapping between private and public addresses. For a TCP session, its lifetime is determined by its connection establishment and termination phases while for a UDP connection, the length of its active period designates its lifetime. To maintain the consistent mapping between private and public addresses, IM/P2P systems may periodically inject probes. Moreover, the churn effect caused by the frequent and unpredictable arrival and departure of IM/P2P nodes also force active hosts to probe networks regularly to obtain current network topology and meta-data [75]. For instance, a *Skype* client routinely send out UDP probes to various super nodes to determine their availability as shown by packets 5 and 6 of Table 3. Compared to the heavy-weight probe sequence of packets 1 to 4, its light-weight counterparts of packets 5 and 6 involve fewer message exchanges. The *Skype* traffic of Table 3 clearly demonstrates that no application protocol field assumes fixed values, defeating any signature-based detection method. However, by correlating message streams in both directions of a *Skype* session, we can observe that the 2-byte *frame ID* field of a heavy-weight probing session, randomly chosen by SC, is echoed back in SN's *NACK* message. Similarly, the 4-byte tag in SN's *NACK* message is also carried by the subsequent SC messages. Therefore, traffic correlation is feasible and effective to the identification of *Skype* probe sessions, which is actually the technique used by our *Skype* analyzer (in Algorithm 4.1 of Section 4.5).

2.5. *Encryption of Communication Messages*

IM/P2Ps also employ cryptographic techniques to protect their communications from eavesdropping, alteration, and replay. However, some IM/P2P abuse encryption techniques to evade security systems. For instance, *KaZaA* obfuscates its communication streams to defeat pattern-based traffic identification systems. Similarly, *Skype* scrambles its traffic with various cryptographic methods according to the communication ports involved. When a *Skype* client (SC) cannot establish a TCP connection to a SN on non-privileged port, it then attempts TCP ports 443 and 80 in this order. If port 443 is used, *Skype* does not follow strictly the Transport Layer Security (*TLS*) protocol typically used by *HTTPS*. If SC uses TCP port 80, *Skype* does not respect the regular *HTTP* standards at all. Instead, *Skype* resorts to its own proprietary protocol or changes the interpretation of standard specifications such as *HTTPS*. Such traffic deviations from standard specifications on TCP ports 443 and 80 make it possible to discern *Skype* traffic in spite of encryption.

Table 4 presents part of the communications between a *Skype* SC/SN pair over TCP port 443 when version 2.5.0.130 is used. Our traffic analysis indicates that the first message from a SC always has 72 bytes even though stream-based TCP connections are used. Such a peculiarity is due to the fact that *Skype* always uses "*PUSH*" to flush out every message it creates. As demonstrated in Table 4, packet 1 is client-hello message embedded in a record layer of *SSL* version 2. In all our traffic traces, packet 1 of all *Skype* sessions over TCP port 443 share the same payload excluding the 16-byte challenge field.

#	*dir*	*len*	*payload*	*description*
protocol: TCP; client (denote as C):192.168.1.67; login server (denote as S): 165.234.212.137:443				
1	C→S	72	|80 46 01 03 01 00 2D 00 00 00 10 00 00 05 00 00 04 00 00 0A 00 00 09 00 00 64 00 00 62 00 00 08 00 00 03 00 00 06 01 00 80 07 00 C0 03 00 80 06 00 40 02 00 80 04 00 80 FD 0A 73 88 59 B6 2F 14 75 22 AB 60 51 4E E7 6C|	SSLv2 record layer (0x80); len: 70 (0x46); handshake msg type: client hello (0x01); ver: TLS 1.0 (0x0301); cipher spec len: 45 (0x002D); session ID len: 0 (0x0000); challenge len: 16 (0x0010); cipher specs: TLS_RSA_WITH_RC4_128_SHA (0x0005) ...(total: 15); challenge (16 B): 0xFD0A...
2	S→C	134	|16 03 01 00 4A 02 00 00 46 03 01 40 1B E4 86 02 AD E0 29 E1 77 74 E5 44 B9 C9 9C B4 31 31 5E 02 DD 77 9D 15 4A 96 09 BA 5D A8 70 20 1C A0 E4 F6 4C 63 51 AE 2F 8E 4E E1 E6 76 6A 0A 88 D5 D8 C5 5C AE 98 C5 E4 81 F2 2A 69 BF 90 58 00 05 00 37 86 50 A3 1B ...|	TLS record layer (0x16 & 0x80 = 0); type: handshake (0x16); ver: TLS 1.0 (0x0301); handshake: len: 74 (0x004A); server hello (0x02); len: 70 (0x000046); gmt_unix_time: Jan 31, 2004 09:23:18... (0x401BE486); bytes (28 B): 0x02AD...; ID len: 32; ID (32 B): 0x1CA0...; cipher suite: TLS_RSA_WITH_RC4_128_SHA (0005); compression: null; data: 0x3786...;
3	C→S	38	|44 0E D5 88 09 5B CB E0 2F 4E 4E DA 21 26 26 01 E4 ...|	encrypted data;

Table 4. Operations on TCP port 443 in Skype (version 2.5.0.130)

The SN-reply of packet 2 is supposed to be a *ServerHello* message; however, it fails to follow the *TLS* constraints in a number of ways: firstly, instead of specifying the server's current date and time, the field *gmt_unix_time* of *Skype* sessions always assume the same and fixed value (i.e., 0x401BE486), clearly deviating from *TLS* specifications. Secondly, the 28-byte field *random_bytes*, which is supposed to be a sequence of randomly generated numbers required by *TLS*, takes constant values (i.e., |02 AD E0 29 ...| as shown in Table 4) for all *ServerHello* messages in *Skype*, which is also a *TLS* violation. Finally, the portion of the message starting from byte 80 and on, does not comply with *TLS*. In addition to the artifacts in protocol fields of *Skype* encrypted messages, message size can be a good indicator for *Skype* streams as well. For instance, the first SC-originated message has always 72 bytes and the second SC-originated message is always 14 bytes for *Skype* version 1.4, while it varies for version 2.0 and later. Obviously, the unique characteristics of such *Skype* traffic over TCP port 443, including constant values in fields *gmt_unix_time* and *random_bytes* as well as the fixed size of first SC-originated message can be used as telltales to identify such traffic.

Skype traffic on regular TCP ports -other than that to port 443- is more challenging to identify as strong cryptographic algorithms may be used. Even so, a number of *Skype* design features generate artifacts that can be successfully exploited. For instance, the obfuscated *Skype* TCP stream is created by applying bitwise exclusive-*OR* operation on the original plaintext and a *RC4*-generated stream. In versions earlier than 2.0, *Skype* applies the first 10-byte of the *RC4*-stream to both the first and next 10-byte plaintext. This generates an artifact in the ensued ciphertext that can be readily exploited. Another artifact that all *Skype* version share in their TCP-streams is that the TCP "*PUSH*" bit is set for all messages. This forces the TCP/IP stack to deliver each *Skype* message individually in a TCP packet as long as the message size is less than *MSS* (maximum segment size). This boundary coincidence between *Skype* messages and TCP packets can help detect *Skype* TCP sessions by analyzing sizes of packets and the correlation among exchanged messages in sessions.

3. A Framework for IM/P2P Traffic Identification

In this section, we outline our framework that identifies IM/P2P sessions in *in-line* fashion. In this regard, it can intercept and thoroughly inspect all incoming/outgoing packets before either forwarding or dropping the packets. We treat network traffic as sequences of application messages instead of TCP/UDP packets; this assists in identifying an IM/P2P session even if boundaries of application messages do not coincide those of underlying transport packets. We resort to stateful inspection on data streams to improve detection accuracy; in addition, by correlating data streams in both directions of each session, false positives/negatives can be reduced dramatically. Our framework dissects data streams at application-layer to uncover IM/P2P sessions using port hopping and encapsulation mechanisms. Such "deep" analysis is feasible via the extensible use of IM/P2P analyzers that can be integrated into our framework in a plug-and-play fashion. Actions on detected IM/P2P sessions include alert generation, packet logging, traffic blocking as well as traffic shaping in order to limit the network bandwidth and resource consumption.

3.1. *Architecture of* IM/P2P *Traffic Identifier*

Two IM/P2P peers can exchange information only after a TCP/UDP connection is established. By denoting the originating site as the "client" while the destination node as the "server", we can uniquely identify a connection with a five-tuple $<IP_c, PORT_c, IP_s, PORT_s, PROTO>$, where IP_c and $PORT_c$ are the IP address and port of the originator (or client), while IP_s and $PORT_s$ are their counterparts for the recipient (or server), and $PROTO$ represents the protocol of the session (TCP/UDP). Within each session, two data streams can be defined, one from client to server, while the other from server to client. We anticipate that a typical IM/P2P system consists of a very large number of concurrent users with each potentially establishing multiple connections to others. To keep track of all IM/P2Ps sessions, a core objective of our framework, it is vital to organize session-related information efficiently so that manipulation of pertinent traffic can be conducted without affecting the performance of the network. The session-related information includes connection status, progress of transmissions, and application types. Tracking session information also makes it possible to perform stateful inspection, session-based manipulation of traffic, and correlation of data streams in both directions of the same session.

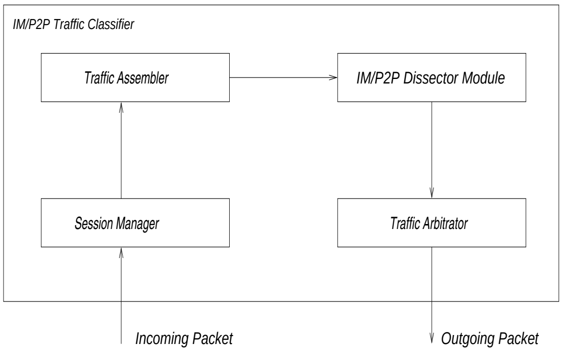

Fig. 1. Architecture of our IM/P2P traffic classifier

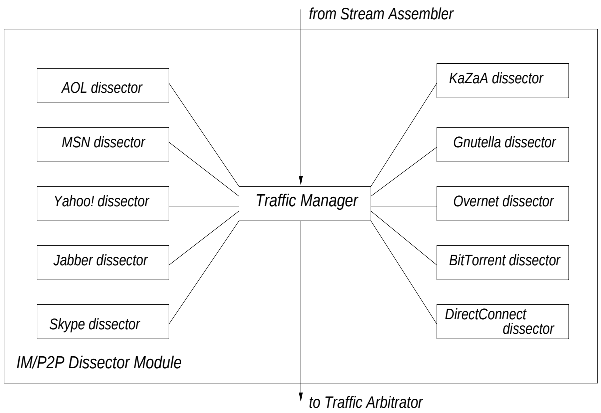

Fig. 2. Components in the IM/P2P–*dissector* module

To restore the boundaries of IM/P2P messages from a stream of transport packets, it is imperative that all constituent packets are stored and orderly re-assembled so that the resulting aggregations can be interpreted in accordance to respective IM/P2P specifications. Without such a re-assembly process, an IM/P2P session may go undetected if its messages happen to stride multiple transport packets, or several messages are packed inside a single transport packet. Figure 1 depicts the overall architecture of our system that consists of *Session Manager*, *Traffic Assembler*, *IM/P2P -Dissector*, and *Traffic Arbitrator*. Any time a packet P arrives, the *Session Manager* determines its session and creates a new one if no such session already exists. The role of the *Traffic Assembler* is to merge all observed packets from the same stream in the correct order. A re-assembled data stream is subsequently worked on by the *IM/P2P -Dissector* module whose objective is to determine the application's type (i.e., *MSN, Yahoo!, Gnutella, Skype* etc.) Finally, packet P is handed over to the *Traffic Arbitrator* module and may be dropped, shaped, forwarded according to designated system configuration, or stored along with other auxiliary information into the disk for forensic analysis.

3.2. *The Session Manager*

We maintain IM/P2P sessions with the help of the `session-table` structure shown in Table 5. Every bidirectional session is uniquely identified with the help of the first five fields in its `session-table` entry namely, IP_c, $PORT_c$, IP_s, $PORT_s$, and $PROTO$. The field `TYPE` provides the application type for the session if it has been identified by the *IMP2P–Dissector* module [e]; otherwise, it indicates that the traffic is non-IM/P2P . While the `CONFIRM` field is set once traffic correlation on both directions of the session has been performed and both streams follow the protocol specification identified by `TYPE`. Fields `start-time` and `last-access` respectively indicate the session creation time and the most recent instance in which traffic from this session was detected. To this effect, sessions without appropriate disconnection procedures can be identified and removed to reclaim resources (e.g., memory). Finally, the two data streams that make up the session are stored in `client_ptr` and `server_ptr` respectively[f].

field name	*size(bytes)*	*description*
IP_c	4	IP address of the originator (denoted as `client`) of the connection
$PORT_c$	2	port number of the originator (`client`) for the connection
IP_s	4	IP address of the recipient (denoted as `server`) for the connection
$PORT_s$	2	port number of the recipient (`server`) for the connection
$PROTO$	1	protocol utilized by the session (TCP/UDP)
`application`	4	application type, initially "unknown", eventually set to one of *non-IM/P2P, AOL, MSN, Yahoo!, KaZaA, ...*
`TYPE`	4	traffic type detected with flow from one direction, type can be one of *AOL, MSN, Yahoo!, KaZaA, ...*
`CONFIRM`	4	traffic type confirmed by flow from other direction of same session
`start-time`	4	creation time of the session
`last-access`	4	last active time of session in either direction (i.e., pkt transmission)
`serverptr`	4	pointer to server `stream` data structure
`clientptr`	4	pointer to client `stream` data structure

Table 5. The `session-table` structure

Our frameworks tracks sessions organized using splay-trees that achieve amortized access/update times within a constant multiple of the information theoretic

[e] *AIM, Yahoo!, MSN, KaZaA*, etc.
[f] `client_ptr` and `server_ptr` are pointers to another structure called `stream` that records packets discussed in Section 3.3.

lower bound; this is attained by moving accessed nodes "closer" to the root rendering future retrievals to frequently used nodes less expensive [72]. Every node in a splay-tree corresponds to a session under surveillance and essentially stores the information of `session-table` described in Table 5. Our session tree T can be searched with *key* constructed by the first five fields of the session table, and some tree operations work as follows: function *session-init(T)* initializes a splay-tree T and makes it ready for operations on sessions such as insertion, retrieval, or deletion of a packet P into T performed by functions *session-insert(T,P)*, *session-find(T,P)*, and *session-delete(T, P)*, respectively. Function *session-find(T,P)* locates the session for a packet P by initially constructing a tuple $<SIP, SP, DIP, DP, PROTO>$ from the source and destination IP/port pairs and the protocol $PROTO$ of P. The tuple is used as a key to search session tree T; if this yields no result, the "dual" tuple $<DIP, DP, SIP, SP, PROTO>$ is formed and is used to search T again. Searches without outcome in both attempts invoke *session-insert(T, P)* to create a new session for P with application type set to "unknown". Clearly, the complexity of the above functions is $O(\log n)$ and is incurred mainly due to splay-tree search.

3.3. *The Traffic Assembler*

The main objective of the *Traffic Assembler* module is to synthesize packets in a data stream according to their correct sequence numbers. For each one-way data stream within a session, we maintain the necessary information to facilitate the traffic re-assembly process. For a TCP stream, we mainly use the following fields of information: `state` tracks the connection status of its originator and its value can be *SYN-SENT*, *SYN-RCVD*, *ESTABLISHED*, or *CLOSE*. Field `ISN` stores the initial sequence number of the stream, while fields `total-size`, and `total-pkt` record the numbers of bytes and packets transmitted so far by the originator of the stream. We use an interval-tree constructed on red-black tree to organize all packets within a stream [13], and field `data` is a pointer to such a structure.

In a UDP stream, we use fields `total-size` and `total-pkt` very much like in TCP streams. Moreover, UDP streams use the fields: `data` points to a buffer that stores all data received for a stream so far, `data-size` indicates the size of buffered data, and `total-size` is the total bytes of data transmitted in the stream. In an interval tree representing a data stream, each node's key is the closed interval $[P_s, P_e]$ corresponding to the start- and end-sequence numbers of packet P while the node's value is the content of P. For a given packet P, P_s can be obtained directly from P's TCP header; while P_e can be derived from protocol fields *total length*, *IP header length*, and *TCP-header size* of P. In addition, we define function *stream-find*(S, P) as returning the stream I for which packet P is part of session S.

The relationship between any two packets P and Q can be determined based on their respective sequence intervals. Packets P and Q are duplicate if $P_s = Q_s$ and $P_e = Q_e$; P precedes Q if $P_e < Q_s$ and P follows Q if $P_s > Q_e$. If conditions $P_s > Q_s$ and $P_e < Q_e$ are satisfied, then P is contained by Q; on the contrary, P

contains Q if $Q_s > P_s$ and $Q_e < P_e$. Packets P and Q are overlapping if ($P_s < Q_s$ and $Q_s < P_e < Q_e$) or ($Q_s < P_s$ and $P_s < Q_e < P_e$). The above defined relationships between packets P and Q affect the behavior of operations on interval-trees. We define functions *interval-insert(I, P)* and *interval-delete(I, P)* to carry out the respective insertion and deletion operations on a given interval tree I for packet P. Obviously, these functions perform standard binary tree search, therefore have computational complexity of $O(\log n)$. Function *interval-retrieve(I, P)* finds all packets in I that have relationship of `overlap, duplicate`, or `contain` with P. Function *interval-build(I, low, high)* creates a new packet Q having start- and end-sequence numbers of [*low, high*]. If the end-sequence number $high = \infty$, then Q is built based on all packets received so far; similarly, if the start sequence number $low = -\infty$, Q is built from the initial sequence number (*ISN*) of the stream. Function *interval-traversal(I)* performs an in-order tree walk of I and lists all intervals in sorted order according to their low endpoints; this is particularly useful when it comes to packet logging. As functions *interval-retrieve(I, P)*, *interval-build(I, low, high)* and *interval-traversal(I)* potentially traverse the entire tree, their complexity is $O(n)$.

With the help of above packet relationships and operations, the stream re-assembly procedure works as follows: for an arriving packet P, its session S is located by function *session-find*(T, P); similarly, its stream I is fetched with the help of function *stream-find*(S, P); Then, function *interval-retrieve(I,P)* is invoked to obtain any packet Q that has relationship of `overlap, duplicate`, or `contain` with P. If Q does not exist, P is a fresh packet and function *interval-insert(I,P)* inserts P into I along with its arrival time. If P overlaps with Q, their overlapping parts are checked to determine whether they are identical. Should P and Q assume the same value in the overlapped part, P is inserted into I with its arrival time, otherwise P is malformed. If P and Q have the same sequence interval, their payloads are compared; if their content is identical, P is a retransmission of Q; otherwise, a TCP specification violation is found. Similarly, for cases where P contains Q or P is contained by Q, we compare the content on their common sequence number interval. If they are the same, then P is a forward overlapped packet when P contains Q and simply a retransmission of Q if P is contained by Q. In the latter case, P is inserted into I; otherwise, P is a suspect packet. Finally, a new sequence of bytes Q is built by calling function *interval-build(I, ISN, ∞)* to re-assemble all packets received by stream I so far.

3.4. *The* IM/P2P *Dissector Module*

As every IM/P2P system defines it own protocol that best suits its service needs, employing a monolithic organization for traffic identification would not be a viable option in the long run. As mentioned earlier, a number of IM/P2Ps such as *Yahoo!* and *MSN* can encapsulate their traffic within normal *HTTP* streams. The straightforward option would have been to develop a a single protocol dissector for HTTP traffic and within its dissector to have different segments for identifying every possible type of IM/P2Ps traffic. However, every time a new breed and/or variant of IM/P2Ps would emerge such monolithic mechanism should be reworked anew. Thus, we opt for a *Traffic Manager* that cooperates with different IM/P2P dissectors each of which analyzes a single protocol as Figure 2 shows. Our overall organization allows for the extensibility of the approach as analyzers for new variants and emerging systems can be readily incorporated into the framework. For example, as *Skype* generates unique traffic patterns when it uses *HTTP* port 80, a new IM/P2P protocol dissector can by developed and plugged into our framework instead of modifying any existing module.

Our *Traffic Manager* works as follows: as soon as a packet P arrives along with its session S, stream I, and re-assembled sequence Q, the *Traffic Manager* checks fields `TYPE` and `CONFIRM` of S to determine the application type of P. If S's type has been determined, P is transferred to module *Traffic Arbitrator* in conjunction with S and I. If the application type of S has not been determined, all IM/P2P analyzers are sequentially called until the type of packet is identified. Algorithm 3.1 outlines the procedure our *Traffic Manager* follows to determine the application type of an incoming packet and its session. As soon as the application type of a session has been determined, all the subsequent session traffic simply passes through the *Traffic Manager* and the entire *IM/P2P dissector* module without any further intervention. In addition, to improve the performance of Algorithm 3.1, we restrict the total sizes of data to inspect within a session in both directions to a user-defined threshold whose default value is 5 KB or 3 packets. If the session's application type cannot be identified after examining this maximum amount of data, the session is declared as non-IM/P2P.

Algorithm 3.1 *Traffic Manager* Operation

$P \leftarrow$ newly arrival packet; $L \leftarrow$ list of all registered IM/P2P analyzers in our framework; S and I are the session and stream of P;
if (field `application` is not "unknown") **then**
  P is part of an identified IM/P2P session; P is passed to module *Traffic Arbitrator* along with S and I and exit from this procedure
end if
while (still non-visited analyzer in analyzer pool L) **do**
  $A \leftarrow$ next analyzer in L; A is invoked with parameter P, S, I, and Q;
  if (field `application` is not "unknown") **then**
    application type of P has been determined; P is passed to module *Traffic Arbitrator* along with S and I and exit from this procedure
  end if
end while
if (total number of bytes of $S \geq MAX\_SIZE$) OR (total packets of $S \geq MAX\_PKT$) **then**
  fields `TYPE` and `CONFIRM` of S are set to be "non-IM/P2P " (default MAX_SIZE is 3KB and MAX_PKT is 5)
end if

To minimize the false positive/negative rates, our IM/P2P protocol analyzers correlate traffic in both directions of a session to ascertain that a protocol is indeed followed by both streams. More specifically for TCP traffic, each IM/P2P analyzer typically operates on the re-assembled data stream instead of the sequence of transport packets, and marks the field `TYPE` of a session based on one data stream of the session and ultimately sets the field `CONFIRM` according to information derived from the other stream. IM/P2P analyzers may dissect a data stream syntactically or semantically. Syntax analysis mostly deal with message structures and formats while the semantic part involve checks on validity of the values in various protocol fields and the message exchange process between the communication ends. As IM/P2P protocols share little commonality in their message structures, each analyzer maintains its own information pertinent to each session. As an example, Algorithm 3.2 demonstrates the functionality of our *Yahoo!* protocol analyzer that identifies IM traffic embedded in *HTTP* streams such as those presented in Table 1.

Algorithm 3.2 Operation of the *Yahoo!* IM Analyzer for *HTTP*-embedded traffic

Input: newly arrived packet P along with its session S, stream I, and re-assembled sequence of bytes Q;
if (Q's destination port is *HTTP* AND Q's method is *POST*) **then**
  data ← data section of the first *HTTP* message in Q;
  ID ← data[0, 3], where data[i, j] means the sequence of bytes from ith to jth in data; ver ← data[4, 5]; type ← data[10, 11];
  if (ID = "YMSG" AND ver is in [0, 9, 10, 11] AND type is in [0, 1, 2, 6, 4C, 57, ...]) **then**
    bit "Yahoo" of field `TYPE` is set;
  end if
else if (Q's source port is *HTTP*) **then**
  data ← data section of the first *HTTP* message in Q;
  tag ← data[0, 3]; ID ← data[4, 7]; ver ← data[8, 11]; type ← data[14, 15];
  if (tag = 0x01000000 AND ID = "YMSG" AND ver is in [0, 9, 10, 11] AND type is in [0, 1, 2, 6, 4C, 57, ...]) **then**
    bit "Yahoo" of field `CONFIRM` is set;
  end if
end if
if (bits "Yahoo" of fields `TYPE` `CONFIRM` are set) **then**
  field `application` of session S is marked as *Yahoo*
end if

Algorithm 3.2 uses the client-originated traffic to tentatively mark the type of S by setting the corresponding field `TYPE` (Table 5). It also resorts on the information derived from the stream of the opposite direction (i.e., from server) to ultimately confirm the type by setting field `CONFIRM`. To distinguish various IM/P2P types, fields `TYPE` and `CONFIRM` work as bit masks in which every IM/P2P type recognized by our framework is allocated one bit. By applying the procedure of Algorithm 3.2 to the traffic in Table 1, we can readily establish the session's type provided that application message boundaries are restored with the help of the preceding *Traffic Assembler* module.

3.5. *The Traffic Arbitrator*

Should the application type of a packet P as well as its session S have not been determined yet, the *Traffic Arbitrator* module simply forwards P to its destination; otherwise, the *Traffic Arbitrator* may impose a number of actions on P and any packets emanating from the same session S according to user-set system configuration. Such actions include dropping a single packet P or all subsequent packets in the session, forwarding traffic, shaping of traffic, and/or blocking connections from the same IP source. Regardless of the action, a copy of P is stored in main memory along with its session information in order to help subsequent re-assembly operation and the determination of its application type. In addition, all packets of an IM/P2P session (including subsequent packets) and information about the session (e.g., its application type, creation time, and transmission statistics) can be selectively flushed to permanent storage for future forensic analyses. In addition to the above choices, the *Arbitrator* module may pro-actively tear down a connection by sending out *TCP RESET* packets or *ICMP destination unreachable* messages to either or both ends of the communication channel. The ultimate handling or *shaping* of stream may depend on the traffic type detected; for instance, streams generated by the *Yahoo!* messenger may be passed but be subjected to traffic shaping to limit its bandwidth and resource consumption, while those of *KaZaA* sessions may be entirely blocked.

4. Protocol Analyzers for IM and P2P Systems

IMs predominantly use the client/server model with the client-part often installed in user machines. The IMs server-component manages and relays data among clients; servers are usually maintained by ISPs such as *AOL, Microsoft*, or *Yahoo!*. However, for media transmissions, IMs resort to peer-to-peer paradigm. In a P2P network, nodes come together to form a distributed platform for resource sharing where all peers are often considered *equal* and may establish direct communications. Contemporary systems such *KaZaA*, *Gnutella*, and *Skype* use hybrid architectures where peers act as either *supernodes* or *ordinary* nodes according on their capabilities. Supernodes host content-indices; regular nodes attach to supernodes and use them as relay outlets for all their operations. In this section, we discuss our analyzers for the *AIM* and *MSN* IMs as well as for *Gnutella*, *FastTrack/KaZaA*, and *Skype* P2Ps . In addition, our framework also supports a number of analyzers for other contemporary IM/P2Ps including the *Yahoo!* messenger, *Jabber*, *Trepia*, *eDonkey*, *BitTorrent*, and *DirectConnect* [8].

4.1. *AOL Instant Messenger (AIM) System*

Both *AOL Instant Messenger (AIM)* and *ICQ* messenger use the proprietary and binary-based protocol *OSCAR*[g] that is now understood with reverse engineering.

[g]Open System for Communication in Realtime

However, aspects of *OSCAR* continuously change so that new features are integrated and third-parties are prevented from connecting to *AIM* servers. *OSCAR* is considered a stream-based protocol as its services are provided through a series of commands in the *FLAP*-format, with each command spreading out over several TCP packets or multiple commands delivered within a single TCP packet depending on command types and transmission timing. Two types of servers exist in *AIM*, the *OSCAR server* and the *Basic OSCAR Service servers (BOS)*. The former is responsible for client authorization and the latter for the realization of instant messaging, information retrieval and account (e.g., buddy nicknames) management. *OSCAR* uses a single-login procedure for clients to join the system as users contact the authorization server (e.g., "login.oscar.aol.com"), provide their account information, and obtain back a cookie used to connect to other servers. By maintaining contact *buddy*-lists and online status information for users, *AIM* not only provides real-time text/voice communication services, but can also block abusive users, deliver status messages, create chatrooms, and support file sharing, group games, as well as audio and video conference.

Table 6 presents the syntax of the *FLAP* protocol. Each message in *FLAP* format contains five fields of which the first four are fixed and span 6 bytes. The field *command start* is the FLAP banner and is always character "*" or *0x2A* in ASCII code. The one-byte field *frame type* or *channel ID* specifies the message type of the current frame with *signon* and *data* values [h] very frequently used here. When the message type is *data*, the content of the data section follows protocol *SNAC* which is outlined in the second part of Table 6. An *AIM* session begins with a *sign-on* procedure as soon as a client connects to an *OSCAR server* via the default TCP-port 5190; the latter is configurable. Once signed-on, the client obtains a cookie from the *OSCAR server* and can subsequently connect to any *BOS server* without any further authentication. *AIM* has also the ability to work with proxy servers using protocols such as *SOCKS4* and *SOCKS5*.

In Table 7, we show a sample of an *AIM* session without the preceding TCP three-way-handshake. Although the client initiates the TCP connection, it is the server that begins the *sign-on* process by sending the client a *new connection* message along with *channel ID*, sequence number, and version number as shown in packet 1 of Table 7. The client responds with a *new connection* command through the sign-on channel (i.e., ID 0x01). Both ends then exchange information for login, authentication, and authorization (packets 3–5). Once the address of a *BOS server* is obtained shown in packet 6 of Table 7, the client closes its current connection and may establish a new session with the identified *BOS server* to finally request IM services.

To identify an *AIM* session, our *AIM* analyzer checks the traffic stream against *FLAP* and *SNAC*. For each incoming message Q, our analyzer first tests the size of Q for the current message, so that it satisfies the minimum length for a well-formed

[h] having the ASCII 01 and 02 values respectively.

field name	*size (B)*	*description*	*check*
		protocol: *FLAP*	
command start	1	command start identifier, always 0x2A ('*');	yes
frame type	1	channel identifier: Signon (0x1), Data (0x2), Error (0x3), Signoff (0x4)	yes
sequence number	2	random, increase in subsequent command, controlled independently in both directions	no
data length	2	size of FLAP data in data field	yes
data	n	data	yes
		protocol: *SNAC*	
family ID	2	identify service groups including Messaging (0x4), Invitation (0x6), Location Services (0x2)	yes
sub-type ID	2	a specific service in the family	yes
flags	2	optional and rarely used	no
request ID	4	identify non-atomic information	no
SNAC data	variable	parameters for the service	no

Table 6. *FLAP* and *SNAC* protocols

pkt	*dir*	*payload*	*description*
		protocol: TCP; server (S): 64.12.161.185:5190; client (C): 192.168.5.141:14431	
1	S→C	|2A 01 33 52 00 04 00 00 00 01|	connection in signon channel, seq. 0x3352
2	C→S	|2A 01 72 3C 00 04 00 00 00 01|	reply in signon channel, seq. 0x723C, ver. 1
3	C→S	|2A 02 72 3D 00 1F 00 17 00 06 00 00 00 00 00 00 00 01 00 09 7A 63 68 65 6E 70 6F 6C 79 00 4B 00|	SNAC family and sub-type: sign-on (0x0017), request (0x0006); username: zchenpoly other info.
4	S→C	|2A 02 33 53 00 15 00 17 00 07 00 00 00 00 00 00 00 09 34 37 ...|	family and sub-type: sign-on, reply (0x0007);
5	C→S	|2A 02 72 3E 00 97 00 17 00 02 00 00 00 00 00 00 00 01 00 09 ...|	subtype: signon, logon (0x0002); screen: zc...
6	S→C	|2A 02 33 54 01 AC 00 17 00 03 00 00 00 00 00 00 00 01 00 09 ...|	signon subtype: logon reply (0x03); screen: zc ...
7	C→S	|2A 04 72 3F 00 00|	close connection with channel ID 04

Table 7. Sample *AIM* packet stream

AIM message (i.e., 6 bytes) and then verifies the validity of values in fields of *FLAP* shown in Table 6 with "yes" in column "check". For messages with *channel ID* of 1, our analyzer further checks the first 4 bytes of its *data* field (i.e., field *version*) to ensure that it is the valid version number (i.e., 0x00000001). For messages in the *singon* channel, the analyzer interprets their *data* fields according to *SNAC* (Table 6). Finally, the analyzer tracks interactions between the client and the server of each *AIM* session to verify their conformance to protocol specifications. Such tracking information helps confirm the application type of *AIM* sessions.

4.2. *MSN Messenger*

Core services provided by the *MSN* IM include user login authentication, management of users contact lists, online state maintenance and notification, asynchronous communication mechanisms, and information access control. *MSN* IM services are provided by different types of servers each having multiple replicates for resilience. Dispatch servers (*DSs*) initially negotiate the version of the *MSN* protocol (*MSNP*) to be used for a specific client; this protocol can be either *MSNP9* or *MSNP10*. The majority of *MSN* services including client authentication, user property synchronization, and event notification delivery are provided by notification servers (*NSs*). Switchboard servers (*SSs*) offer lightweight communications among users and are predominantly used for messaging. Should a client A intend to communicate with B, A ships a request to an overseeing *NS* which in turn refers A to a *SS* server; B receives notification from its *NS* and finally, a connection is established to the same *SS*.

The *MSN* IM protocol is ASCII/line-based and uses TCP as its transport service. Each command begins with a case sensitive three-letter instruction followed by zero or more parameters, and terminated with carriage-return and line-feed (ASCII 0x0A/0x0D). Parameters are normally encoded with ASCII and are separated by whitespaces (%20) but UTF-8 encoding can be used as well. Requests are delivered asynchronously and so multiple requests can be concurrently submitted without waiting for responses from a server; the server should deliver either a response or error message for each request, but not necessarily in the same order as received. The parameter *transaction ID* in each command can be used to match request and response messages. As *MSN* IM functions over a network, any *NS* can provide authentication at TCP port 1863 by default. Clients can connect to servers via *SOCKS4*, *SOCKS5*, or *HTTP* proxy as well. With the exception of the authentication sequence in which passwords are *MD5*-encrypted, all messages are transferred in clear text, making it easy for our analyzer to dissect pertinent *MSN* IM sessions.

#	*dir*	*message*	*description*	*check*
protocol: TCP; server (S): 207.46.106.75:1863; client (C): 192.168.5.141:4370				
1	C→S	VER 4 MSNP10 MSNP9 CVR0	ver supported: NSNP9, MSNP10; transaction ID (i.e., TrID): 4	yes
2	S→C	VER 4 MSNP9 CVR0	reply to request with TrID 4, use ver. 9	yes
3	C→S	CVR 5 0x0409 winnt 5.1 i386	request with TrID 5, client's platform info.	yes
		MSNMSGR 6.1.0207 MSMSGS	client's MSN version	
		username@hotmail.com	user handler	
4	S→C	CVR 5 6.0.0602 5.0.0527 1.0.0000 http://download.microsoft.com	response to request with TrID 5, specify dispatch server	yes
5	C→S	USR 6 TWN I username@hotmail.com	user ID and security package	yes
6	S→C	USR 6 TWN S lc=1033,id=507, tw=40,fs=1, ...	information for authentication	yes

Table 8. A typical *MSN* IM session

Table 8 shows a typical *MSN* messenger session. The first client command *VER* in message 1 sets its *TrID* to be 4 and message 2 in its respective reply reciprocates with the same *TrID*. At first, client and server execute the version negotiation protocol to ensure that they both support the same version. The result of this negotiation in Table 8 is the use of *MSNP9* even though the client supports both *MSNP10* and *MSNP9* as message 1 depicts. The client also provides its *MSN* IM version, platform, and operating system through the *CVR* command. In response, the server manifests its own *MSN* version and provides a URL to download the latest *MSN* in messages 3 and 4. Command *USR* in messages 5 and 6 help exchange information between client and server for login and authentication. To identify an *MSN* IM session, our analyzer tracks every command in both data streams of the session, checks its format, the validity of its parameters, and interactions between requests and responses. To improve accuracy, all parameters are decoded if they are encoded with UTF-8 standard. In addition, for any session connecting to TCP port 80 of a host, our analyzer also inspects the data section of each *HTTP POST* message for encapsulated *MSN* messenger traffic.

4.3. *The Gnutella Network*

Although earlier versions of the *Gnutella* protocol used *flooding* method to locate files, its later versions feature a number of extensions including intelligent query routing, SHA hashing for file checksums, file compression, partial-file sharing, and parallel download (swarming) to improve performance. To avoid flooding storms, the network was organized around powerful *ultrapeers* that form an overlay network. The latter is responsible for the creation, organization, and maintenance of indices of resources found in ordinary peers under the jurisdiction of each *ultrapeer*. Flooding is prevented by having queries be communicated among *ultrapeers*. Typically, *Gnutella* peers acts as both client and server (termed servents). To join a *Gnutella* network, a peer has to latch to at least one active node; this involves querying a number of known bootstrap servers or a list of ultrapeers maintained locally by the client. Once online, a peer can communicate through messages *ping, pong, query, query-hit,* and *push*. *Ping* attempts to discover active hosts in the network and its response *pong* outlines a peer's address, port(s), and information about files available. *Query* tries to locate a requested resource; its receiver may respond with a *query hit* message if a match is found in the receiver's local data-set. Finally, *push* allows a servent behind firewalls to contribute data to the network.

Every *ultrapeer* is aware of its neighbors; this neighbor list is periodically updated by dispatching *ping*s with *TTL*s limiting their scope to 7 hops. A node's *pong*-response is routed in reverse over the same path traveled by the just-received *ping* and the response information is used to either create or update the initiator's neighbor list. Ordinary nodes maintain lists of active ultrapeers and update such lists any time they attach to a supernode. The descriptor *query* is passed from an ordinary node to one of its *overseeing* ultrapeers that returns a *query hit* if the request can be locally satisfied; otherwise, the query is delegated to neighboring ultrapeers. As soon as the requested resource has been located, a standard *HTTP* session is established directly between the initiator and provider to download the requested file making it hard to differentiate *Gnutella* from Web traffic.

A conversation between an ordinary node and an *ultrapeer* begins with a *Gnutella* three-way handshake procedure. In particular, Table 9 shows two sessions initiated by a peer: the first fails due to overload at the target ultrapeer as packet 1 and 2 depict. On the other hand, packets 3–5 show a successful join. The client's greeting

#	*dir*	*payload of message*	*description*
client (C): 192.168.5.141:33359; server-0 (S0): 68.104.240.174:6346; server (S): 137.99.153.234:6346			
1	C→S0	GNUTELLA CONNECT/0.6	request for joining *Gnutella*
2	S0→C	GNUTELLA/0.6 503 Full	*Gnutella* server is overloaded (code 503)
3	C→S	GNUTELLA CONNECT/0.6 Accept-Encoding: deflate X-Ultrapeer: False; X-Query-Routing: 0.1; Pong-Caching: 0.1; GGEP: 0.5; FP-1a: 128,h...	banner for Gnutella network can receive compressed data; this peer is not an ultrapeer; routing protocol and pong caching; Gnutella Generic Extension Protocol
4	S→C	GNUTELLA/0.6 200 OK Accept-Encoding: deflate; Content-Encoding: deflate; X-Try-Ultrapeers: 130.127.82.159:6346,...; X-Ultrapeer: True; X-Query-Routing: 0.1; Machine: 1,...	banner for Gnutella network can receive compressed data; may send compressed data; other ultrapeers (in "IP:port" pair); this peer is an ultrapeer; query routing protocol
5	C→S	GNUTELLA/0.6 200 OK Content-Encoding: deflate; FP-1c: j<c ...	handshake banner for Gnutella network compress data; encryption information

Table 9. Traffic generated by a *Gnutella* peer attempting to join the network

message contains the banner string *GNUTELLA CONNECT/0.6* and a number of *key:value* pairs, specifying its capabilities and functionalities. For instance, *Accept-Encoding* indicates the sender can decompress incoming messages, *X-Ultrapeer* specifies whether the sender is an ultrapeer, while *X-Query-Routing* and *GGEP* signal a client's support for query routing protocol and *Gnutella* Generic Extension Protocol (GGEP). Some clients even include information for authentication and encryption as shown by *FP-1a* and *FP-auth-Challenge* of packet 3. In its response, the server includes the banner *GNUTELLA/0.6 200 OK*, manifests its ultrapeer status with key *X-Ultrapeer* set to *True*, indicates acceptance of compressed messages with *Accept-Encoding* set to *deflate*, and signals that it can compress its transmitted messages as well by having *Content-Encoding* set to *deflate*. Authentication and encryption related information may be included in the key *FP-1b* of message 4. The third step of the handshake completes with the client sending back a message starting with banner *GNUTELLA/0.6 200 OK*; subsequently, the client node can look for files dispersed in the network. To identify a *Gnutella* session, our analyzer examines all banner strings (i.e., *GNUTELLA CONNECT/0.6* and *GNUTELLA/0.6*) appearing in messages and the interactions between the peers of the session (i.e., *GNUTELLA CONNECT/0.6* from client followed by *GNUTELLA/0.6* from server and finally *GNUTELLA/0.6* from client). In addition, it dissects some of the <*key:value*> pairs as information contained in such pairs helps in the analysis of subsequent messages. More specifically, information in key *X-Ultrapeer* is used to construct a list of potential *Gnutella* super nodes, which can be used to further verify the application types for sessions involving hosts in the list.

4.4. *The FastTrack Protocol and KaZaA Network*

The *FastTrack* protocol used by *KaZaA*, *Grokster*, and *iMesh* P2P systems, is a proprietary suite that integrates a number of advanced features including peer hierarchy, resumption of interrupted downloads, and simultaneous downloading of file segments from multiple sites. Joining peers are classified as either super or ordinary-nodes based on system characteristics. Supernodes act as directory facilities for sets of regular nodes. An ordinary node initially resorts to a hard-coded list of supernodes to join the network; each supernode is described by the template <*IP:port*>. Once attached, a peer may update its supernode list to better reflect its current environment. At the same time, the node uploads to its supernode a list of its files to-be-shared. A query is routed to the supernode of the requesting peer for identifying a site holding the sought file; eventually, the *HTTP* protocol helps transfer the file from the holding peer to the requesting peer. To prevent the development of open-source clones and defeat detection by security systems, *FastTrack* encrypts all messages among supernodes and most messages from ordinary nodes to supernodes, making it challenging to identify such traffic. Even so, reverse engineering is still possible for communications between ordinary nodes and supernodes since initialization data for encryption algorithms –not public key encryptions– are dispatched in clear text.

KaZaA uses both TCP/UDP layers and Table 10 shows the format for its *handshake request*, *handshake response*, *ping*, and *pong* messages. An ordinary node may try multiple methods to join the network. For instance, it may use flooding by sending UDP-*ping*s to all its supernodes, may attempt to establish TCP connections with multiple supernodes at the same time and expect a response from any one of them, or use UDP and TCP alternatively until a working supernode is located. When a TCP-connection gets established, the initiating peer sends a *handshake*

msg type	*field name*	*size*	*description*	*check*
handshake request (TCP)	rand	4	random number	no
	seed	4	cipher seed, used to encode "type" and derive encryption key	no
	type	4	encryption type ([0x29, 0xBF] before encoded with "seed")	yes
handshake	seed	4	cipher seed, used to encode "type" and derive encryption key	no
response (TCP)	type	4	encryption type ([0x29, 0xBF] before encoded with "seed")	yes
ping (UDP: peer to supernode)	message type	1	0x27 for "node *ping*" message	yes
	type	4	encryption type in [0x29, 0xBF]	yes
	unknown	1	always be 0x80	yes
	network name	n	zero-terminated network name ("KaZaA")	yes
pong (UDP: supernode to peer)	message type	1	0x28 for "node *pong*" message	yes
	type	4	encryption type in [0x29, 0xBF]	yes
	unknown(1)	1	always be 0x00	yes
	unknown(2)	5	purpose unknown	no
	network name	n	zero-terminated network name	yes

Table 10. *KaZaA*'s syntax for *handshake request*, *handshake response*, *ping*, and *pong* messages

request to the supernode containing a random number (4 bytes), a seed (4 bytes) for encoding and encryption purposes, and the *encryption type* (4 bytes), which is encoded with the help of the seed. The encoding algorithm in question has been reverse engineered [26,58]. Should the supernode accept the request, it ships back a handshake response containing its own encryption seed and type, while the latter is encoded with the same algorithm as that of the ordinary node. Based on the seeds provided by both ends of the session, the supernode and the ordinary node can compute encryption keys for their incoming and outgoing data streams, and all subsequent messages are encrypted using the resulting keys [26,58].

#	*dir*	*payload*	*field name*	*content*	*description*	*check*
protocol: UDP; client (C): 192.168.5.141:3037; server (S): 66.130.102.247:2713;						
1	C→S	|27 00 00 00	message-type	27	ping msg to join KaZaA	yes
		A9 80 4B 61	type	0x000000A9	encryption type (0xA9)	yes
		5A 61 41 00|	unknown	80	purpose unknown	yes
			network-name	KaZaA	network name	yes
2	S→C	|28 00 00 00	message-type	28	"pong" message (0x28)	yes
		A9 00 35 2C	type	0x000000A9	encryption type	yes
		5C 34 F3 4B	unknown(1)	00	purpose unknown	yes
		61 5A 61 41	unknown(2)	0x352C5C34F3	purpose unknown	no
		00|	network-name	KaZaA	network name (6 bytes)	yes

Table 11. A typical UDP-based portion of *KaZaA* session with *ping* and *pong* messages

When the underlying transport protocol is UDP, the encryption type is transferred in clear-text, making it straightforward to identify a UDP-based *KaZaA* session. We show a sample UDP-based *KaZaA* session with the supernode using UDP port 2713 instead of the default port 1214 in Table 11; the ordinary node sends a *ping* and the supernode answers with a *pong*. Both sides of the session agree to use encryption type 0xA9 among the 100 types of encryption available for their subsequent communications. To identify a UDP- or TCP-based *KaZaA* session, our analyzer scans messages for their syntax and semantics conformance to the *FastTrack* protocol. The analyzer examines values appearing in fields specified in column *check* of Tables 10 and 11 for legitimate values. Since all messages except the first two in a *KaZaA* session (TCP or UDP) are encrypted and cannot be dissected without being decrypted first, our protocol analyzer always dissects the first two messages of each session according to the *FastTrack* protocol for establishing potential *KaZaA* traffic.

4.5. *The Skype VoIP System*

As a voice over IP (*VoIP)* application based on P2P technology, *Skype* has gained in popularity after its initial launch [30]. Since 2003, more than twenty different *Skype* versions have been released. To facilitate inter-operations, many *VoIP* systems are constructed according to the standard specification of Session Initiation Protocol (SIP). In contrast, *Skype* resorts to proprietary protocols for signalling and media delivery. In addition, communications in *Skype* are obfuscated or end-to-end encrypted with strong cryptographic algorithms such as RSA, AES, and RC4. Furthermore, its ability to detect firewalls and NATs offers *Skype* the capability to penetrate and circumvent network security systems [5,20]. Finally, with the help of information on network environments and security restrictions, *Skype* automatically adjusts its behavior including transport services (TCP and/or UDP), communication ports, and message exchange procedures avoiding all together its manual user configuration.

In a way reminiscent to *KaZaA*, *Skype* uses an overlay P2P network with three main components: login server (*LS*), supernode (*SN*), and *Skype* client (*SC*). *LSs* authenticate *SCs* and grant the latter access rights to the network. *SNs* route *SC*-messages to appropriate destinations, relay login messages between *LSs* and *SCs*, and handle traffic between *SCs* should the latter are NAT/firewall-restricted. The *SCs* are essentially the user interface to *Skype* functionalities including call initiation, instant messaging and file transferring [7]. To join the network, an *SC* attempts to contact an *SN* whose information is locally stored and kept up-to-date. Information on several bootstrap-*SN*s is hard-coded into *Skype* binaries to help first-time *SCs* obtain updated lists of currently available *SNs*. The *SC* transport service can be either TCP or UDP [i] depending on the network environment. If resident behind security devices, a client attempts to initiate connections to an *SN* using TCP ports 443 (*HTTPS*) and 80 (*HTTP*). In general, *Skype* uses arbitrary ports when UDP is the transport option; the only exception to this rule is when bootstrap-*SNs* are contacted and in this case port 33033 is used.

Through analysis and correlation of *Skype* sessions generated by various versions, we have established that each *Skype* UDP message has a header consisting of a *frame ID* (2 bytes) and *function type* (1 bytes) and a message body whose size may vary and in most cases its content is obfuscated with the help of the RC4 encryption method. *Frame IDs* are used to distinguish different sessions while *function types* determine the constructions and interpretations of message bodies. In *Skype* signalling, UDP messages are mainly used for NAT detection and *SN* availability probing; the former is deemed as heavy-weight probing since it consists of multi-round information *SCs* and *SNs* exchanges while the latter is a light-weight procedure as it involves only a single messaging round. Although such UDP probes are very similar among different *Skype* versions, the size of their messages may vary. Table 12 depicts the communications between a SC/SN pair in *Skype* version 1.3.0.60 at the very beginning of an *SC* execution. For comparison, we use the same setting for the traffic in Tables 12 and 3.

Our traffic analysis confirms that the first UDP message by *SCs* is always 14 bytes for all versions upto 2.0 as shown in Table 12. However, this initial message shows varying sizes after version 2.0. Table 3 shows a first UDP message with size 20 bytes for example. A similar observation was made as far as the size of the second *SC*-originating message is concerned. The size was fixed to 23 bytes in versions prior to 2.0. Nevertheless, what all versions share in common is that the size of second *SC* message is always 5 bytes larger than the first one. The first *SN* message is 11 bytes long and the length of the second either 18 or 51 or 53 bytes. The *function type* of

[i]not mandatory.

#	*dir*	*len*	*payload*	*description*
protocol: UDP/TCP; SC: 192.168.1.66:3993/5422; SN: 71.207.146.44:12653				
1	SC→SN	18	|37 91 02 3C 79 79 FE 27 64 A3 EF B1 38 15 19 60 B2 AB|	frame ID: 0x3791; function type: 0x02, obfuscation msg; initialization vector: 0x3C7979FE; CRC32: 0x2764A3EF;
2	SN→SC	11	|37 91 17 47 87 44 F6 6D 3F 3B 99|	frame ID: 0x3791; func. type: 7 (0x17&0xF), NACK; src: 0x478744F6; tag: 0x6D3F3B99;
3	SC→SN	23	|37 91 03 01 6D 3F 3B 99 47 CF 92 2C 27 64 A3 EF F2 9C D5 13 C6 4A 5F|	frame ID: 0x3791; function type: 0x03, retransmission; tag:0x6D3F3B99; SN:0x47CF922C (71.207.146.44);
4	SN→SC	18	|FD 09 02 91 89 6E 9C 04 11 F5 8A E0 CE 41 62 D3 63 AD|	frame ID: 0xFD09; function type: 0x02, an obfuscation message; length = 18 indicates joining *Skype* successfully
5	SC→SN	0	(SYN)	TCP session to SN on same port (i.e., 12653)
6	SN→SC	0	(SYN|ACK)	
7	SC→SN	0	(ACK)	
8	SC→SN	14	|10 EA 08 5D AE 92 7B 97 1D 6C 10 E3 7B CF|	encrypted data

Table 12. Joining the *Skype* network by a *SC* v.1.3.0.60

packet 3 is 0x03, indicating a retransmitted message by *SC*; in addition to using the same *frame ID*, this message also echoes back the 4-byte tag in the NACK and includes the IP address of the *SN* (i.e., 71.207.146.44). The *SN*-dispatched packet 4 has a different *frame ID* from packet 3 and its message body is entirely obfuscated. However, as our analysis shows, the message with length of 18 bytes indicates that the particular *SN* can serve the requesting client. The *SC* establishes a TCP connection with this *SN* on port 12653 as packets 5 to 8 of Table 12 show. In contrast, packet 4 of Table 3 is 53 bytes long, implying that the just-contacted *SN* is busy and so the *SC* request is redirected to other *SN*s as packets 5 and 6 demonstrate.

Algorithm 4.1 Protocol Analyzer for *Skype* UDP traffic

```
P is the newly arrival packet, S and I are session and data stream that P belongs to
ID ← P[0, 1]; where P[i, j] means the sequence between ith and jth bytes of P starting from
  zero; type ← (P[2] & 0x0F); OBFUSCATE ← 0x02, NACK ← 0x07;
if (I is from client to server in session S) then
  if (P is the first packet of I in session S) and (type is OBFUSCATE) and (size of P >=
  18) then
    ID is store in I; TYPE of S is set to Skype
  end if
else
  if (type is NACK) and (length of P is 11) and (ID is the same as that stored by client)
  then
    CONFIRM of S is set and a Skype session is detected
  else if (type is OBFUSCATE) and (length of P is 18, 51, or 53) then
    CONFIRM of S is set and a Skype session is detected
  end if
end if
if (both TYPE and CONFIRM of S is set to Skype) then
  application of S is set to Skype
end if
P is handed over to Traffic Arbitrator (TA)
```

Skype services such as authentication, buddy search, and call initiation follow different protocols and consequently generate different sequences and/or templates of traffic. As these sequences demonstrate diverse characteristics and share few commonalities, we have no other choice but design various *mini*-analyzers that cover all aspects of *Skype* generated traffic. For brevity, we only present in Algorithm 4.1 the procedure followed by our framework to specifically identify *Skype* UDP probing discussed above. This *mini*-analyzer is based on traffic correlation to identify the probing procedure. The first client-message of a session is examined to ensure that its payload is longer than or equal to 18 bytes and its function type is "*OBFUSCATE*" (i.e., 0x02). A positive outcome tentatively marks the session as *Skype*. Then, server-dispatched messages are used to finally verify the session. For heavy-weight UDP probing, the first message from the *SN* always has function type of NACK and payload size of 11 bytes as mentioned earlier. In contrast, the *SN* typically uses function type of "*OBFUSCATE*" in light-weight UDP probing. Algorithm 4.1 handles both heavy and light-weight probing appropriately and confirms pertinent sessions accordingly.

5. Experimental Evaluation

We have implemented our IM/P2P framework in C and incorporated it into *FortiGate*'s IPS module. *FortiGate* is a stand-alone network device providing firewall, anti-virus, and IDS/IPS functionalities [34]. Its modularized architecture allows for the seamless coupling of new packages and forms the basis for its multi-modal operation. In particular, we have integrated our framework into *FortiGate*-800 that features 4 Gigabyte main memory and is prorated to handle upto 400 Mbps traffic and can maintain up-to one million connections per second. We have established the correct operation of the suggested framework in the controlled testbed environment of Figure 3 and evaluated its operation through deployment in real-world networks as well. Section 5.1 outlines our baseline experimentation while section 5.2 presents our scalability experimentation in our controlled test environment, and finally Section 5.3 discusses representative outcomes of the framework's deployment in real networks.

While working with the controlled testbed environment of Figure 3, we use a number of machines to either execute clients for various *IM* systems or act as peers –ordinary nodes or supernodes– for *P2P* networks. All test machines use Windows2000 or Linux and connect to the *FortiGate* via a 100/1000 Mbps switch. To verify the behavior of our framework, we used the traffic sniffer *Ethereal* [22] on a dedicated machine (shown as *Sniffer* in Figure 3). The sniffer captures data exchanged between test machines and our framework.

5.1. *Baseline Behavior of Our Framework*

To establish the baseline operation of our framework in the presence of port hopping and/or use of dynamic ports, we experiment with all our IM/P2Ps analyzers [8]. For

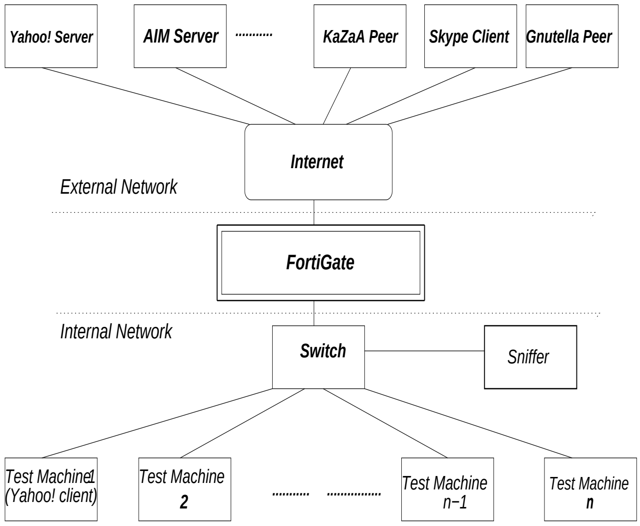

Fig. 3. The controlled environment used for testing our framework

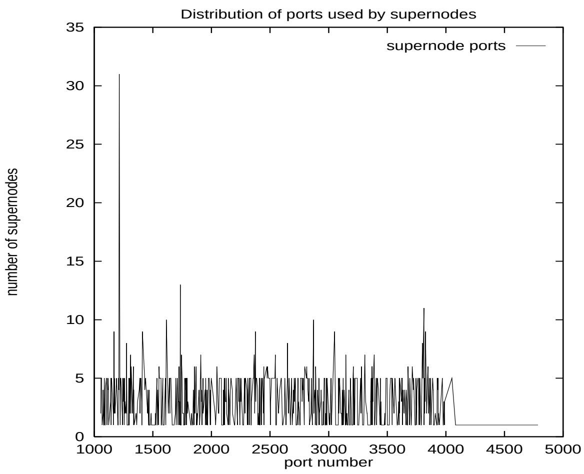

Fig. 4. Observed distribution of *KaZaA* supernode ports

brevity, we discuss our findings involving a *Yahoo!*-client login session as well as a session with a *KaZaA*-client.

We install a *Yahoo!*-client on test machine 1, and configure it with *no firewall* connection-type implying that its user believes there is no security device involved. In actuality though, we position *FortiGate* between the internal network where the *Yahoo!*-client resides and the external network.

- We initially set the framework to forward all identified *Yahoo!*-sessions and use the *Sniffer* to capture all communications. We show portion of the ensued traffic in Table 13. Using the normal TCP three-way handshake procedure, the client establishes a TCP connection with the *Yahoo!*-IM server at its default 5050 port. In its first subsequent message in packet 1, the client dispatches a "*service verify*" request (with type 0x4C) along with its supported version (0x0B) and status ("*available*"). Through its *Yahoo!* analyzer, our framework tentatively marks the connection as *Yahoo!* IM session. As soon as the server replies with packet 2 accepting the request and granting access rights, our framework correlates the information of packet 2 with what it has already "seen" in the other direction of the session and confirms the session type. As the configured counter-action is set to *forwarding*, the session "flows" through the framework with no obstruction.

#	*dir*	*payload*	*description*
protocol: TCP; client (C): 192.168.5.40; server (S): 216.155.193.145;			
counter-measure on identified *Yahoo!* sessions: forwarding			
1	C:4000→S:5050	YMSG|00 0B 00 00 00 00 00 4C 00 00 00 00 00 00 00 00 00 00|	client msg; ver: 0x0B; type: VERIFY (0x4C);
2	S:5050→C:4000	YMSG |00 00 00 00 00 5C 00 57 00 00 00 01 7A 60 ...|	server reply; ver: 0x00; type: AUTHENTICATE (0x5C);
counter-measure on identified *Yahoo!* sessions: blocking			
1	C:4058→S:5050	YMSG|00 0B 00 00 00 00 00 4C 00 00 00 00 00 00 00 00 00 00|	client message: ver: 0x0B; type: VERIFY (0x4C);
2	C:4058→S:5050	(FIN|ACK)	connection closed by client side
3	C:4063→S:23	YMSG|00 0B 00 00 00 00 00 4C 00 ...|	Telnet port
4	C:4064→S:80	YMSG|00 0B 00 00 00 00 00 4C 00 ..|	HTTP Web server
5	C:4065→S:21	(SYN)	FTP control port
6	C:4066→S:25	YMSG|00 0B 00 00 00 00 00 4C 00 ...|	SMTP
7	C:4067→S:119	YMSG|00 0B 00 00 00 00 00 4C 00 ...|	NNTP
8	C:4068→S:20	YMSG|00 0B 00 00 00 00 00 4C 00 ...|	FTP data port
9	C:4070→S:5050	YMSG|00 0B 00 00 00 00 00 4C 00 ...|	standard port

Table 13. Procedure for a *Yahoo!*-client to join the network at the presence of our framework

- We next configure *FortiGate* to block all identified *Yahoo!*-sessions. While repeating the above *Yahoo!*–IM login process, we generate the traffic shown in the second portion of Table 13. By correlating information from the client-initiated request and the server's reply –the latter not shown in Table 13,– our framework confirms the session as *Yahoo!*-IM and blocks it. The client issues packet 2 terminating the attempt to port 5050 after a specified time period elapses and subsequently tries to contact the server over ports 23, 80, 21, 25, 119 and 20. These ports are for the provision of `telnet`, *Web*-server, *FTP*-control, *SMTP*, *NNTP*, and *FTP*-data services respectively. Through correlation of streams, *FortiGate* identifies and blocks all attempted *Yahoo!* IM sessions regardless of the ports attempted.

It is worth pointing out that the *Yahoo!*–server does not listen to TCP-port 21 and so the connection attempted by packet 5 is not successful as the normal TCP three-way handshake procedure fails. In contrast, the server provides services over ports 23, 80, 25, 119, and 20; the *Yahoo!*-client can dispatch messages to the server once TCP connection has been established. Moreover, the payloads of packets 3, 4, 6, 7 and 8 start with *YMSG* which is the banner for *Yahoo!*–IM messages indicating that no message encapsulation occurs.

- By setting *FortiGate*'s counter-measure to *reset connection* for identified IM/P2P sessions, we observe that *Yahoo!* sessions similar to those shown in Table 13 are terminated with TCP-*Reset* packets dispatched by our module instead of the normal disconnection procedure. Similarly, our framework appropriately delivers all prescribed actions in all occasions; these actions also include *reset client, reset server, block source-IP,* and *block destination-IP* with marginal time overheads.

Next, we configure the *Yahoo!*-client on test machine 1 with *firewall with no proxy* connection-type to have *Yahoo!*-IM use its encapsulation technique to "penetrate" the firewall. By repeating the above testing procedure multiple times and with the help of the *Sniffer* we capture the resulting traffic, a portion of which is shown in Table 1. If we compare the second (TCP) session of Table 1 with that represented by packet 4 of Table 13, different patterns occur. Although in both cases the respective servers listen to TCP-port 80, the session shown in Table 1 has its data section encapsulated in *HTTP* messages; in addition, a single *Yahoo!*–IM message is split into multiple TCP packets transport as the first client-initiated message of Table 1 depicts. Our framework identifies correctly all such *Yahoo!*–IM sessions and delivers appropriate counter-action.

We derive similar findings when experimenting with the entire range of IM/P2P systems [8] as our framework is based on layer-7 analysis. The latter helps successfully deal with encapsulation to IM/P2P 2p protocols, spread of application messages into multiple transport packets, placement of multiple messages in a single packet as well as use of evasion techniques in the test-environment of Figure 3.

To assess the capability of our framework on the recognition of P2P sessions and the use of dynamic ports in the context of our test environment, we monitor a *KaZaA*-client connected to a respective overlay-network. Pertinent client-to-server connections are facilitated with the help of a list of approximately 200 supernodes –in the format of <*IP:port*>– found in the client's registry [j]. We initially configure *FortiGate* to forward all *KaZaA* traffic. While operating the client for a long time, we extract the content of its supernode list every 30 minutes; we then aggregate this data to obtain the statistics of Figure 4 regarding the use of ports versus nodes. The figure clearly shows that only a very small fraction of supernodes –the long peak that corresponds to only 1.6%– uses the default port 1214 for service. Most of the *KaZaA* supernodes use ports within the [1024, 4054] range; although some small peaks exist in Figure 4, the use of dynamic ports appears to be relatively uniform.

[j] the list is located at *HKLM/Software/KaZaA/ConnectionInfo/KazaaNet* and is updated as discussed earlier.

By changing *FortiGate*'s action to blocking and with the help of the *Sniffer*, we establish that our framework successfully intercepts and drops all *KaZaA* traffic despite the ever changing use of ports and supernodes by the client.

5.2. *Scalability and Performance Under Diverse Synthetic Workloads*

To ascertain the capabilities of our framework in handling very large numbers of concurrent IM/P2P sessions and successfully tracking the states of such sessions, we use the testbed of Figure 5. In this context, we recreate IM/P2P flows from the *Sniffer*-captured sessions of section 5.1 [9] and feed them into *FortiGate* with the help of an in-house developed IPS testing-system called *Tester*. The above flows are termed *foreground* traffic. We generate such foreground traffic by either using a single-type of IM/P2Ps flow or mixing a number of IM/P2P streams with different ratios. As our objective is to establish the behavior of our framework in light of diverse workloads, we also inject non-IM/P2P streams into the testbed of Figure 5. Non-IM/P2P streams make up what we call *background* traffic and are generated with the help of *CAW WebAvalanche* and *CAW WebReflector* devices, typically used for system and network equipment testing [74]. *CAW WebAvalanche* and *CAW WebReflector* help create *HTTP* requests and replies to emulate the behavior of Internet users and "ordinary" traffic characteristics in terms of intensity and duration of sessions. These *HTTP* requests and replies feature on average payloads of 200 and 1000 bytes respectively [53].

While performing stress-tests with the testbed of Figure 5, we vary the number of concurrent IM/P2P sessions from 1 to 400,000 and that of *HTTP* traffic from 10,000 to 750,000 to investigate the scalability of our framework. The selection of the above session ranges that we experiment with is bounded by the 4 Gigabytes memory used by *FortiGate-800*. In particular, if n_f, n_b represent the number of foreground and background sessions and S_f, S_b express in bytes the main-memory requirements for managing single foreground/background sessions in our framework, then the total memory required is $M{=}n_f S_f{+}n_b S_b$ bytes. Background sessions are permitted to flow through the device without further inspection from our framework after only a few messages have been analyzed. On the other hand, foreground sessions may take upto the maximum allocated buffer space if IM/P2P traffic is user-configured to be logged for forensic analysis [k]. In the worst case, we may have only IM/P2Ps traffic coming into the framework requiring $M^{'}{=}n_f S_f$ bytes; should n_f is 1,000,000 and in the average each session uses $S_f{=}4$ Kbytes, we are faced with buffer space depletion. In a different scenario where 50% of the concurrent connection are IM/P2Ps -originated, $n_f{=}500{,}000$, $n_s{=}500{,}000$, and on average $b_f{=}27$ Kbytes and $S_b{=}3$ Kbytes, we exceed the available buffer space. The above analysis underlines the fact that the memory of *FortiGate-800* may become the performance bottleneck

[k]Otherwise, only the first 3 KBytes of an IM/P2P session is recorded.

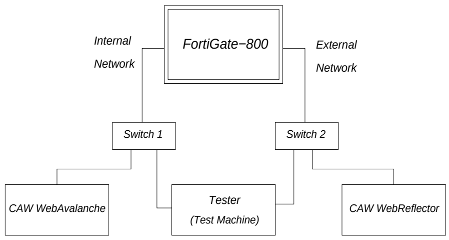

Fig. 5. Testing using *WebAvalanche*, *WebReflector* and *Tester*

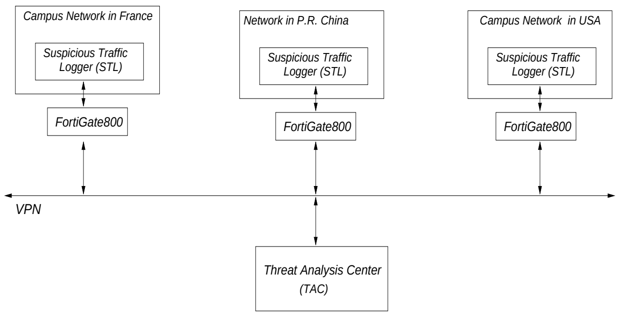

Fig. 6. Deployment of our framework in France, P.R. China and United States

once we reach one million concurrent sessions.

In the results we discuss here, we use the *Yahoo!*–IM session segment depicted in the second portion of Table 1 to create the foreground traffic. The packets of this segment are split into two parts: the first containing the normal TCP three-way handshake process (not shown in Table 1) and packets 1–2. The second part consists of all remaining packets as well as the normal disconnection procedure (also not shown in the Table 1). The *Tester* replays the trace-derived packets to *FortiGate* with source and destination IP/port information modified on-the-fly [1]. We configure our framework to generate an alert when the `TYPE` of a session is tentatively marked as IM/P2P and to yield a second alert when the `CONFIRM`-ation of a session finally occurs.

Table 14 presents the results for tests we have carried out in conjunction with the *Yahoo!* traffic. We obtained similar results while experimenting with different types of traffic including *MSN*, *AIM*, *KaZaA*, and *Skype* as well as combinations but we restrict our discussion here for brevity. In test case 1, the *Tester* replays the first half of the trace and pauses. Then, *WebAvalanche* opens varying number of *HTTP* sessions to *WebReflector* as shown in the columns of Table 14 (e.g., column "10,000" indicates ten thousand simultaneously open sessions of background *HTTP* traffic

[1]to comply with the actual features of the testbed.

are generated). *WebReflector* reciprocates for each request received and responds with a Web page without any delay. 500 ms after its pause, the *Tester* resumes its replay procedure by injecting the second half of the foreground traffic into *FortiGate*. We observe the behavior of *FortiGate* and compute its detection precision, which is defined as the ratio between *Yahoo!* sessions identified by *FortiGate* and total replayed *Yahoo!* sessions. The first row of Table 14 shows the outcome of our framework as the number of *HTTP* sessions gradually increases upto 750,000.

#	*Description*	*10,000*	*75,000*	*100,000*	*500,000*	*750,000*
1	1 *Yahoo!* session, no HTTP delay	100	100	100	100	100
2	1 *Yahoo!* session, HTTP delay	100	100	100	100	100
3	10,000 *Yahoo!* sessions, no HTTP delay	100	100	100	100	100
4	10,000 *Yahoo!* sessions, HTTP delay	100	100	100	100	100
5	100,000 *Yahoo!* sessions, no HTTP delay	100	100	100	100	100
6	100,000 *Yahoo!* sessions, HTTP delay	100	100	100	100	100
7	200,000 *Yahoo!* sessions, no HTTP delay	100	100	100	100	100
8	200,000 *Yahoo!* sessions, HTTP delay	100	100	100	100	100
9	300,000 *Yahoo!* sessions, no HTTP delay	100	100	100	100	100
10	300,000 *Yahoo!* sessions, HTTP delay	100	100	100	100	99.90
11	400,000 *Yahoo!* sessions, no HTTP delay	100	100	100	100	99.80
12	400,000 *Yahoo!* sessions, HTTP delay	100	100	100	100	99.00

Table 14. Session identification rates of the proposed framework under diverse workloads

Test case 2 is identical to 1 except that the *WebReflector* now delays its response to each *HTTP* request by half a second, attempting to lengthen the *HTTP* sessions and therefore forcing *FortiGate* to endure a much longer period with sustained concurrent sessions. In case 3, the *Tester* replays the first portion of *Yahoo!* traffic 10,000 times simultaneously. When the *Tester* pauses, the *WebAvalanche* creates concurrent *HTTP* sessions similarly to those of case 1 and the *WebReflector* responds for every received request without delay. Subsequently, the *Tester* resumes its replay process and feeds the second half of the trace into the testbed 10,000 times. Case 4 builds on 3 but there is a 0.5 second delay between every *HTTP* reply and request. Cases 5, 7, 9, and 11 are similar to 3 but the number of *Yahoo!* sessions ranges from 100,000 to 400,000. Likewise, cases 6, 8, 10 and 12 are similar to 4 with the number of IM/P2P sessions varied from 100,000 to 400,000.

The results of Table 14 indicate that our proposed framework correctly identifies *Yahoo!* streams as long as active sessions are under one million regardless of the types and mixtures of the IM/P2P and non-IM/P2P sessions. When the workloads feature in excess of one million sessions and significant time delays are introduced between *HTTP* requests and replies, which effectively lengthens the period of sustained concurrent sessions, *FortiGate* fails to properly identify a limited number of *Yahoo!* sessions (cases 10-12 with 750,000 background sessions). It is worth mentioning however that while examining *FortiGate*'s event-log, we have established that even missed *Yahoo!* sessions are still tentatively tagged as such. By simply using an alternate session eviction policy rather than the default LRU policy and staging-out sessions with application type non-IM/P2P first, we have attained 100%

accuracy detection in all above cases.

#	*Description*	*attempted sess.*	*min. t*	*max. t*	*avg. t*
1	no IM/P2P traffic without framework	2,278,563	109	135	115
2	no IM/P2P traffic with framework	2,278,500	109	135	115
3	10 Mbps IM/P2P traffic with framework	2,286,445	109	192	120
4	20 Mbps IM/P2P traffic with framework	2,281,437	109	377	121
5	30 Mbps IM/P2P traffic with framework	2,279,526	109	1,405	124
6	40 Mbps IM/P2P traffic with framework	2,280,431	109	2,112	125

Table 15. Framework under constant 200 Mbps *HTTP*-traffic and varying intensity IM/P2Ps traffic

Our next goal is to quantify the impact of our framework on the response time of normal applications as well as its overhead on non-IM/P2P sessions. We first remove our IM/P2Ps module from the *FortiGate* device of Figure 5 so that we can establish the baseline performance of *HTTP* traffic created by *WebAvalanche* and *WebReflector*. The *HTTP* traffic generated by both *WebAvalanche* and *WebReflector* is sustained at 200 Mbps with an average Web page size of 1,000 bytes. The foreground traffic is generated with the help of the *Tester*, which repeatedly replays, with the specified rate in the range [0, 40] Mbps, the IM/P2P *Sniffer*–captured traces of Section 5.1 and shown at Tables 1, 2, 7, and 9. In addition, our framework is configured to forward all identified IM/P2P sessions. Then, we integrate our module into *FortiGate* and repeat the experiments. In both settings, we measure the time elapsed between the launch of an *HTTP* request by *WebAvalanche* and the time the corresponding *WebReflector*-originating-reply arrives back. Table 15 shows the number of *HTTP*-sessions attempted as well as the minimum, maximum and average response times per request in microseconds. Case 1 of Table 15 shows the response time when only *HTTP* traffic is involved and our framework is not present in the testbed. The results for cases 2–6 are compiled while varying the intensity of IM/P2P traffic from 0 to 40 Mbps in the presence of our framework. In all our experiments here, there are no failed *HTTP*-sessions, our framework poses only minor increases in the average rates compared with the *avg. time* of case 1, and the minimum response times appear constant. However, the framework forces the response time to be occasionally as long as 2,112 microseconds when 40 Mbps of IM/P2P traffic is injected into the testbed. This deviation occurs only when the IM/P2P traffic accounts for the 17% of the total traffic (i.e., 40/240) and the combined traffic is 60% of the prorated bandwidth of the device(i.e., 240/400).

5.3. *The Effectiveness of Suggested Framework in the Real World*

We have evaluated our IM/P2P identification framework in real-world settings and in this section we discuss representative findings from the deployment of IM/P2P -identification-enabled *FortiGate-800*s at the edge of the network of three higher education institutions in France, P.R. of China and the U.S.A. In collecting network-traffic data, we used the layout of Figure 6 to capture, store and forward both confirmed and suspicious IM/P2P sessions to a *Threat Analysis Center (TAC)* for further verification and signature crafting purposes. To help discover new types of attacks and better ascertain the false-negative rate of *FortiGate-800*, we use a *Suspicious Traffic Logger (STL)* module to log streams and/or sessions that are potential yet not resolved/known IM/P2Ps . Such suspicious IM/P2P traffic is identified through various criteria including flows to default ports of already known IM/P2P applications, streams marked tentatively by our framework but lacking confirmation information, and sessions having obtained with only partial signature matching. Regional devices periodically transfer data of both detected IM/P2P and suspicious connections to *TAC* where the actual sampling and analyses of the traffic take place.

Table 16 shows session statistics derived with the help of the deployed devices for traffic ultimately forwarded to *TAC* during the period of August 1st to 5th, 2006. The table presents both confirmed and suspected instances of top-ranked IM/P2P

	Day 1	Day 2	Day 3	Day 4	Day 5
IM/P2P	*cfm./susp.*	*cfm./susp.*	*cfm./susp.*	*cfm./susp.*	*cfm./susp.*
eDonkey	67981/339908	57280/265785	60470/348007	72013/269200	71259/296284
BT	37604/188022	30572/170400	40642/181011	40185/150980	39461/152523
Gnutella	22884/114420	17774/115476	23320/88193	18500/94026	19032/105366
Skype	7442/37214	6464/35332	7557/32800	7997/34273	7890/28881
KaZaA	4905/24528	4756/18517	3745/22410	4971/26647	4667/24101
Yahoo	2050/10250	1751/9480	1838/9689	1662/8859	1565/11081
IRC	1777/8884	1596/8925	1655/8875	1496/8801	1681/7410
MSN	1727/8635	1845/7356	1750/6840	1699/9114	1320/7408
D-Connect	648/3244	635/2918	661/3181	662/3108	567/2864
AIM	214/1070	204/1086	215/881	193/869	184/1089

Table 16. IM/P2P detected by our framework operating in networks in France, P.R. China and the U.S.A.

sessions for each observation day in columns *cfm.* and *susp.* respectively. The top-three IM/P2P types of identified sessions are due to *eDonkey*, *BitTorrent (BT)*, and *Gnutella* whose cumulative number of sessions is by far larger than all remaining systems including *KaZaA* and *Skype*. Moreover, streams generated by P2Ps are exceedingly more voluminous than their IMs counterparts highlighting their ever increasing popularity and wide-spread use. Table 16 shows that the number of confirmed sessions for any specific IM/P2P type does not change significantly during the observation period implying that IM/P2P users have consistent behavior. For instance, the *eDonkey* sessions appear with a mean of 65,800 sessions per day and standard deviation 5,901 sessions. Similar observations are drawn for other IM/P2P types as well as suspicious sessions. It is worth pointing out that the volume of IM/P2P traffic in the above environments remains relatively stable on a daily basis and typically makes up about 10% of the total traffic.

Forwarding captured sessions to *TAC* offers the opportunity to examine in detail and verify the nature of sessions. In addition, it helps with the identification of new strains/versions of IM/P2P systems. As it is clearly infeasible to manually inspect every session, we resort to sampling and select 4,000 from those confirmed IM/P2P sessions. The sample maintains the same ratio of confirmed IM/P2Ps types in the forwarded traffic; within each type, sessions are selected randomly. For instance, we select 1,063 sessions from those confirmed *BitTorrent* flows as overall *BitTorrent* has a 27% session-presence (i.e., 188,464 *BitTorrent*-confirmed over the total number of 708,966 IM/P2P -confirmed sessions). Through manual examination by domain experts at *TAC*, we establish all IM/P2P sessions marked by the framework are indeed generated by IM/P2P systems.

From the captured sessions in the same period, we also randomly choose 4,000 suspect, yet not identified sessions. We carry out manual inspection and we draw the following observations:

(1) Sessions not tentatively marked by our framework are verified that they are unlikely to be generated by known IM/P2P systems.
(2) Among those tentatively marked sessions, 15% use TCP as their transport service, and most of the TCP sessions use well-known ports such as 80 and 443. Our exhaustive manual evaluation reveals no true IM/P2P steams in these sessions.
(3) The UDP sessions are approximately five times more common than their TCP-based counterparts. By and large, these sessions are created by P2P applications using UDP messages to probe for the availability of servers and/or supernodes. UDP-probing to inactive nodes, also known as churn effect [75], produces no information for our framework to conduct traffic correlation and so to successfully confirm the application type.

In summary, our *TAC*-based analyses showed that the proposed framework generates no false positives/negatives for IM/P2P sessions. In its default mode, the module that implements the framework does not report failed attempts by IM/P2P applications to establish sessions. However, this can be addressed by configuring the module to produce alerts for tentatively marked UDP IM/P2P sessions in addition to all confirmed ones.

5.4. *Using Other Open-Source Tools for Detecting* IM/P2P *Sessions*

A few open-source projects can be used to detect IM/P2P sessions including *Snort*, *Bro* and *IPP2P* [59,55,21,57]. *Snort* and *Bro* have been designed as intrusion detection/prevention systems (IDSs/IPSs) and base their operations on pattern matching methods. Through specially-crafted signatures, they can also detect IM/P2P traffic. *Snort*'s steadily increasing user-group has created a wealth of pattern signatures. From such signatures currently available in the "out-of-the-box" configuration, only 2% can help in the detection of IM/P2P activity generated by *AIM*, *Yahoo!*, and *BitTorrent*. Table 17 in Appendix 8 presents a number of such *Snort* signatures. In its official build, *Bro* can only detect *Gnutella* and *IRC* and its capability of identifying IM/P2P traffic heavily relies on signatures imported from *Snort*; Table 18 shows pertinent signatures. On the other hand, the main objective of *IPP2P* has been to exclusively detect P2P systems exploiting telltale patterns that appear in ensued traffic. A few rules that *IPP2P* uses to carry out its session identification appear in Table 19. While experimenting with the signatures of Tables 17, 18 and 19 in both synthetic workloads and traces, we observe the following regarding *Snort, Bro*, and *IPP2P*:

- their capabilities mostly focus on the transport packet level rather than the application layer messages. Despite the fact that both *Snort* and *Bro* have built-in TCP reassembly features, their IM/P2P -related signatures do not exploit such features inevitably affecting their identification accuracy.
- their IM/P2P –related signatures are by and large designed to detect traffic on specific ports so that false positives are avoided. For example, source/destination ports are required in *Snort* signatures *1991, 2450*, and *1382*; the same applies for *Bro* signatures as well. Clearly, this is disadvantageous.
- they are ineffective when it comes to detecting IM/P2Ps system that offer different services via multiple transport mechanisms; the latter create diverse traffic characteristics any time different services are invoked. This is the case with *KaZaA* that produces entirely different traffic patterns in the login/authentication phase and the file downloading operations. In this regard, none of the *Snort, Bro*, and *IPP2P* can entirely block all *KaZaA* communications.
- they are "blind" to IM/P2Ps that use encryption to obfuscate their streams. In its official signature database, *Snort* provides no signature for TCP-based signalling traffic generated by *KaZaA* and *Skype*; similarly, *IPP2P* has no attempt to identify such traffic, either.
- they offer weak traffic correlation capabilities. Traffic correlation can be used to enhance IM/P2P identification accuracy, especially when uni-directional traffic signatures prove ineffective. Only, *Snort* provides limited traffic correlation functionality demonstrated by signatures *6000* and *6001* of Table 17.

It is worth pointing out that packet-based signatures also demonstrate different degree of effectiveness across the tools. For example, *Bro* signature *s2b-1631-8* [m] offers improved detection accuracy over the *Snort* signature *1631* as it searches for a longer pattern. Compared to *Snort* signature *1383*, *IPP2P* may yield better accuracy for *KaZaA* downloading traffic; *Snort* rule *1383* can only identify *KaZaA* downloading traffic connecting to TCP port 1214. Moreover, rule *1383* only checks the first four bytes of the packet to be "*GET* ". In contrast, *IPP2P* detects *KaZaA* media transfer traffic based on content instead of using fixed ports; furthermore, *IPP2P* not only ensures that the packet should begin with string "*GET* " but also contain either "*X-Kazaa-Username:*" or "*User-Agent: PeerEnabler*" reducing false positive/negative rates. In comparison to the above options, our framework avoids the use of fixed-ports, does not exclusively focus on packet-based signature

[m] that is derived from *Snort* signature *1631*

detection and exploits traffic correlation in a concerted effort to eliminate false positives/negatives. Last but not least, its extensibility through the use of plug-and-play analyzers enables the comprehensive treatment of both IM and P2P sessions in a unified manner.

6. Related Work

IM/P2Ps demonstrate traffic patterns that are very different from those created by traditional WWW-applications [23,2,78]. Although IM/P2Ps still heavily rely on the client/server paradigm for delivering services including authentication and authorization, they mainly follow the peer-to-peer paradigm when it comes to functionalities such as file sharing and signalling for the management of overlay networks [76,31,42,1]. Due to their "symmetric" communications, long data transfer times, and geographically dispersed resources, nodes in IM/P2P systems show significant consumption on both computational resources and network bandwidth [65,45,31]. The popularity of IM/P2Ps has also made them prime target for attacks including *DOS* attacks, disclosure of sensitive data, loss of data integrity, and host compromise or crash [6]. P2P systems including *KaZaA*, *Grokster*, and *Morpheus* suffer from spoofing identity attacks in their file request handling [6] and are vulnerable to *DOS* attacks, should malicious hosts repeatedly send large numbers of messages, and demonstrate buffer overflow problems in super-node packet handlers [6]. P2Ps appear to have a close relationship with spywares such as adwares, browser hijackers, keyloggers, and spybots [63]. Piggybacked on P2Ps and silently installed on clients, spywares can modify browser settings, track client's activities, display targeted advertisements, and even record passwords [33].

As computing infrastructures are undoubtedly affected by the resource-intensive and security-vulnerable IM/P2P applications, organizations have opted for restricting and/or blocking such traffic [56,52] and a number of approaches have been proposed. Fixed-port IM/P2P classification methods, widely used by firewalls and traffic filters, base their operations on the assumption that IM/P2P applications always use their default ports [51,69,64,25]. Heuristics have been also used to extract P2Ps packet patterns based on associations between source and destination IP-addresses/ports [39], However, the rapid IM/P2P development along with the adoption of port hopping and message encapsulation render fixed-port approaches ineffective [70,48]. Signature-based IM/P2P identification methods have been proposed to classify traffic according to unique patterns appeared in various data streams [18,70]. Streams are pronounced to be P2P as long as their traffic contains respective telltales. Unfortunately, such methods can only recognize IM/P2P traffic as far as their file downloading activity is concerned and fail to identify encrypted streams and signalling communications often used for maintaining overlay networks [18,70]. In addition, the packet-based pattern searching used in [70] is rather ineffective when a pattern spreads over multiple packets. The traffic monitor proposed in [38] examines sequences of bits to identify P2P signalling traffic from *eDonkey*, *Gnutella*, and *Bit-*

Torrent. However, packet-based pattern matching and inspection on first 44 bytes of each packet only make the proposed method vulnerable to false positives and negatives. In [32], an attempt to automatically generate signatures is discussed and aims at reducing costs often necessitated by the use of domain experts.

Statistical methods have been also proposed to identify IM/P2P traffic based on the aggregate behavior of data streams instead of the content of their flows. To this effect, both statistical and structural aspects of messages are used in conjunction with Markov process models and common substring graphs to characterize application streams in [48]. In [18], a classification system for the identification of Internet relay chat (IRC) systems is discussed based on packet-size statistics in addition to fixed-port and telltale pattern matching. P2P statistical characteristics such as the percentage of failed connections as well as the ratio of initiated over received connections are used to identify P2Ps as well [3]. Furthermore, distributions of the packet inter-arrival time in flows are exploited by IM/P2P classifiers [39,79]. Machine learning techniques such as Bayesian analysis have been also applied to IM/P2P traffic classifications [50]. Although statistical and machine learning techniques may assist in identifying traffic with encryption and encapsulation, the level of granularity such approaches operate in is coarse and often such method are unable to distinguish among various IM/P2Ps that take place simultaneously [48]. In addition, significant amounts of data in flows should be observed in order to ensure the validity of aggregated properties and subsequently their classification. The latter implies that the use of statistics and machine-learning-based approaches might not be an effective choice, should near-real-time manipulation of networking sessions is required [40,79,15].

Among other functions, intrusion detection/prevention systems (IDSs/IPSs) attempt to classify traffic of IM/P2P applications and subsequently manage it [59,71,40]. In order to deliver counter-actions on identified IM/P2P flows, IPSs typically employ hybrid approaches using port- and signature-based detection methods. For instance, with specially-crafted signatures, *Snort*, an open-source IPS, may identify IM/P2Ps including *AIM* and *Gnutella*, and its counter-measures on detected sessions include packet-dropping, session blocking, and/or connection termination [59,73]. A number of products also provide IM/P2P traffic identification and manipulation functionalities including *RealSecure*, *UnityOne* and *WatchDog* [77,56]. *RealSecure* may identify and block some IM/P2P traffic by predominantly using fixed ports and pattern matching [56]. The *Peer-to-Peer Piracy Prevention* module of *UnityOne* can restrict P2P traffic by setting policies and quotas for users based on fixed IP addresses and/or application types [77]. When integrated with firewalls and/or IPSs, *WatchDog* can detect, shape, and block P2P sessions [52]. In summary, most IM/P2P traffic classification methods identify IM/P2Ps control/data streams based on fixed port-numbers and packet-based inspection, and may generate false positives/negatives. In addition, many techniques can only identify IM/P2P media traffic but fail to recognize critical IM/P2P signalling messages. More importantly, they are not designed to be flexible and extensible to account with ease for new flavors of IM/P2P traffic without major rework in their internals.

7. Conclusions and Future Work

Sessions generated by Instant Messaging and Peer-to-Peer (IM/P2Ps) systems now constitute a significant portion of Internet traffic consuming network bandwidth and computing resources. Although IM/P2Ps present readily exploited vulnerabilities, users often consider them harmless and use them to share private information and sensitive data. Thus, it becomes imperative for organizations be able to identify, monitor, and manipulate IM/P2P traffic. The unique features of such traffic make detection increasingly difficult as new-breed IM/P2Ps systems resort to traffic hiding, security penetration techniques, port hopping, message encapsulation, and information encryption. In this paper, we propose a comprehensive framework to identify and control IM/P2P traffic in real-time. We resort to traffic re-assembly, stateful inspection, data stream correlation, application layer analysis as well as session-based pattern matching to classify traffic.

Our framework consists of four core-modules that operate synergistically, namely: *Session Manager*, *Traffic Assembler*, IM/P2P *Dissector*, and *Traffic Arbitrator*. The *Session Manager* organizes TCP/UDP connections so that sessions information can be efficiently managed. The *Traffic Assembler* re-constructs data within a stream so that they can be effectively handled as a sequence of application messages instead of a set of independent transport packets. We use splay/interval trees to organize the voluminous IM/P2P sessions and their associated streams with each stream likely consisting of numerous packets. The operation of the IM/P2P *Dissector* module is based on specific protocol analyzers that can be deployed in an plug-and-play fashion. The analyzers interpret each application message according to protocol specifications defined by individual IM/P2P services. This analysis goes beyond conventional syntactic inspection as it exploits semantics including order and relationships of messages in different directions of the flows. We have designed analyzers for a wide range of IM systems such as *AIM, MSN, Yahoo!*, and *Jabber*, as well as for P2P systems including *Gnutella, KaZaA, eDonkey, BitTorrent, DirectConnect*, and *Skype*. As soon as our framework identifies an IM/P2P session, the *Traffic Arbitrator* module helps control and/or manipulate the corresponding streams with counter-measures including packet dropping and connection termination in a configurable manner. We have implemented our framework and tested it in a wide range of settings to demonstrate its capabilities. Our experiments in both controlled settings and real networks show that our prototype raises no false positives or false negatives, identifies IM/P2P traffic correctly when traffic is encapsulated or encrypted. Finally, our framework does not affect system performance noticeably in terms of throughput and response time.

In the future, we intend to enhance our framework with analyzers for other IM/P2P systems such as *Winny, WinMX*, and *EarthStation*. In addition, we also continuously update our framework to identify new-breed IM/P2P systems and their variances, especially the upcoming versions of the very popular *Skype* system. We anticipate that a very large number of coexisting analyzers may put significant strain on our framework implementation that has to be addressed. We also intend to thoroughly evaluate the framework's requirements on resources, and its behavior under conditions of unusual traffic load.

8. Appendices

8.1. *Snort-rules for Identifying* IM/P2P *Sessions*

Signature-based traffic classification methods are widely used in IM/P2P session detection, which mostly entails searching for telltale patterns in data flows, specific message exchange styles, and/or abnormal values in various protocol fields. In this regard, *Snort*, an open-source IDS/IPS, is a good example [73,59]. Pattern matching is the predominant IM/P2P detection methods employed by *Snort* and some of such IM/P2P -specific signatures are presented in Table 17, here, each signature is assigned an identifier presented in column "*sid*". Signature *1631* identifies an *AIM* login session by matching the packet's destination IP address with one of those *AIM* servers provided in variable *$AIM_SERVERS* (e.g., 64.12.24.0/24, 64.12.25.0/24, where /24 is the network mask). Qualified packets are further inspected to ensure that their TCP-payload start with character sequence of "*|01|". This rule fails to detect an *AIM* session connecting to servers not included in *$AIM_SERVERS*; in addition, *AIM* sessions escape *Snort* detection when message boundaries are not honored by the underlying transport services. False positives may also occur as the rule only checks the first two bytes of the packet payload. The intend of signatures *1991* and *2450* is to trap *MSN* and *Yahoo!* IM sessions, respectively. However, by inspecting only TCP-sessions with destination ports 1863 or source port 5050, these rules miss all sessions with source/destination ports latching to ports rather than their default ones. For instance, both *MSN* and *Yahoo!* provide port hopping mechanism to locate servers, the former sweeps ports 1863 and 80, while the latter scans ports 23, 80, 25, 119, and 20 in addition to its default port 5050. Furthermore, Signatures 1631, 1991, and 2450 may generate false alarms as they check only traffic in one-direction.

If string *GNUTELLA CONNECT* is found within the first 40 bytes of a TCP-packet sent to a server, signature *556* marks the session involved as *Gnutella*. *KaZaA* sessions used for file downloads can be detected with signature *1383* only if such sessions happen to use the default TCP port 1214, such traffic is rare in nowadays *KaZaA* networks. In contrast to the above signatures that function individually and independently, signatures *6000* and *6001* are crafted to work together in order to identify *Skype* login sessions. By monitoring traffic from client to server, signature *6000* verifies that the packet starts with pattern "|16 03 01 00 CD|", if so, the session will be marked as *Skype*, but no alert is generated yet at this stage. The session is declared as true *Skype* only after conditions specified in signature *6001* are also satisfied by the traffic from server to client of the same session; the conditions in signatures *6001* state that the packet must start with pattern "|16 03 01 00 CD|" and the session in question has been marked as *Skype* by signature *6000*. In essence, signatures *6000* and *6001* attempt to correlate the packets exchanged between client and server, but in a coarse granularity as they operate on packet level instead of IM/P2P

sid	*rule*	*explanation*
1631	tcp $HOME_NET any − > $AIM_SERVERS any (msg:"CHAT AIM login"; flow:to_server,established; content:"*|01|"; depth:2;)	In established TCP sessions, find pkts to AIM servers (e.g., 64.12.24.0/24), start with "*|01|"
1991	tcp $HOME_NET any − > $EXTERNAL_NET 1863 (msg:"CHAT MSN login attempt"; flow:to_server, established; content:"USR "; depth:4; nocase; content:" TWN "; distance:1; nocase;)	In established TCP sessions, find pkts to servers at port 1863, containing cmd "USR" and "TWN" (security package)
2450	tcp $EXTERNAL_NET 5050 − > $HOME_NET any (msg:"CHAT Yahoo! IM successful logon"; flow: from_server,established; content:"YMSG"; depth:4; nocase; content:"|00 01|"; depth:2; offset:10;)	In established TCP sessions, spot packets from servers at port 5050, starting with "YMSG" and having service type 1 (i.e., "LOGON")
556	tcp $HOME_NET any − > $EXTERNAL_NET any (msg:"P2P Outbound GNUTella client request"; flow:to_server, established; content: "GNUTELLA CONNECT"; depth:40;)	Check TCP pkts to any port of external network connecting to "server", having "GNUTELLA CONNECT" at first 40 bytes
1383	tcp $EXTERNAL_NET any − > $HOME_NET 1214 (msg:"FastTrack (kazaa) GET request"; flow:to_server,established; content:"GET "; depth:4;)	in established TCP session, check pkts from client to server on port 1214 and contain "GET "
6000	tcp $HOME_NET any − > $EXTERNAL_NET any (msg:"P2P Skype client login startup"; flow:to_server, established; content:"|16 03 01 00 CD|"; depth:5; flowbits:set,skype.alternate.login; flowbits:noalert;)	traffic from client to server; starts with specified pattern; mark session as *Skype*;
6001	tcp $EXTERNAL_NET any − > $HOME_NET any (msg:"P2P Skype client login"; flow:to_client, established; flowbits:isset,skype.alternate.login; content:"|17 03 01 00 D9|"; depth:5;)	traffic from server to client; *Skype* is marked already by 6000; starts with specified pattern;

Table 17. Rules used in *Snort* to detect IM/P2P sessions

message level. However, compared to signatures that only inspect uni-directional data stream of sessions, such stream correlation techniques definitely improve IM/P2P traffic classification accuracy. Overall, state-of-the-art devices identify IM/P2Ps traffic based on packet boundaries in the transport layer instead of using divisions at the application level. Signatures based on standard and fixed ports are likely to miss all IM/P2Ps sessions due to use of port hopping and message encapsulation.

8.2. *Detecting* IM/P2P *Sessions using Bro*

A Unix-based IDS, *Bro* passively monitors network traffic and may detect suspicious activities by parsing/extracting application-level telltale patterns [55,54,43]. *Bro* has been developed primarily as a research platform and thus, it is used either for experimental purposes or to help verify results of other IDS systems. Developed around the libpcap library, *Bro* carries out filtering of packet streams based on *policies*, then its *event-engine* classifies filtered streams into events which are finally processed by its *script interpreter*. Suspicious activities designated by user-authored policy scripts are ultimately recorded by the *script interpreter*. The IDS counter-actions include generation of e-mail/paging messages, termination of connections and shaping of traffic with the help of external programs. *Bro* features only a few signatures and policy scripts as far as IM/P2P traffic is concerned. In its most recent version (1.1d), only portions of *IRC* and *Gnutella* traffic can be detected in *Bro*'s default configuration. Due to the esoteric nature of *Bro* scripting, the most convenient way to use the tool is through importation of *Snort* rules.

Table 18 shows a few signatures imported from the *Snort* rule set version 2.4 depicted in *Bro*'s format. Signature *s2b-1631-8* is used to detect traffic generated by the *AIM* login

signature	*rule*	*explanation*
s2b-1631-8	ip-proto == tcp; tcp-state established,originator; event "CHAT AIM login"; payload \x17\x00\x06/; /\x02.{4}.{0,4}\x00	TCP pkt begins with "*|02|"; then "|00 17 00 06|;
s2b-2452-4	ip-proto == tcp; dst-port == 5050; tcp-state established,originator; event "CHAT Yahoo IM ping"; payload /[yY][mM][sS][gG]/; payload /.{9}\x00\x12/;	TCP pkt to port 5050; with two given strings;
s2b-2586-2	ip-proto == tcp; dst-port == 4242; tcp-state established,originator; event "P2P eDonkey transfer"; payload /\xE3/;	TCP pkt to port 4242; pkt starts with 0xE3;
s2b-1699-7	ip-proto == tcp; tcp-state established,originator; event "P2P FastTrack kazaa/morpheus traffic"; payload /GET/; payload /.*UserAgent\x3A KazaaClient/;	established TCP session's pkt begins with "GET"; contains string "User...";
s2b-2180-2	ip-proto == tcp; tcp-state established,originator; event "P2P BitTorrent announce request"; payload /.{0,1}GET.{1}.*\/announce/; payload /.{3}.* info_hash=/; payload /.{3}.*event=started/;	TCP pkt to server; contains three strings: "GET", "info...", and "event=..."
s2b-2181-2	ip-proto == tcp; dst-port >= 6881; dst-port <= 6889; tcp-state established,originator; event "P2P BitTorrent transfer"; payload /\x13BitTorrent protocol/;	TCP pkt to [6881, 6889]; contains given string: "BitTorrent ..."

Table 18. Rules used in *Bro* to identify IM/P2P traffic

procedure; it monitors the TCP stream to the server that starts with pattern "*|02|" followed by string "|00 17 00 06|". Compared to *Snort* signature with *sid 1631* of Table 17, we can observe that both signatures try to identify the same *AIM* traffic, however, they inspect different telltale patterns. The reason is that signature *s2b-1631-8* is derived from *Snort* rule with revision 8, while *sid 1631* of Table 17 is with revision 6. Clearly, *Snort* revises its rules progressively in order to improve its detection accuracy. Similarly, *Snort*-rule *sid 2450* and *Bro* signature *s2b-2452-4* identify *Yahoo!* IM traffic. However these two signatures search for different *Yahoo!* IM service types; *s2b-2452-4* looks for the *ping* service, while *2450* inspects the login service. Both signatures *s2b-2180-2* and *s2b-2181-2* are designed to identify *BitTorrent*'s tracker and peer communications; the former searches for patterns "*GET*", "*/announce*", "*info_hash=*", and "*event=started*", while the latter looks for pattern "*|13|BitTorrent protocol*". As *Bro* mainly relies on signatures imported from *Snort*, its capabilities can only be as powerful as those of *Snort*. Moreover, *Bro* is further limited as it does not support *Snort*' flow correlation and byte-wise operations.

8.3. *IPP2P-signatures for Detecting* P2P *Sessions*

IPP2P builds its functionality around the combined use of iptables and netfilter and searches for telltales unique to P2P flows [57]. The tool can impose traffic logging, packet dropping, traffic shaping to limit both bandwidth and system resource consumption. A noteworthy limitation of *IPP2P* is that it cannot deal with P2Ps that may create diverse stream types such as *Skype*. Also, *IPP2P* only recognizes some of the UDP-signalling of *KaZaA* and is completely "blind" to all *KaZaA* TCP-based signalling messages. Version 0.8.2 of *IPP2P* can recognize twelve P2P systems when it comes to their TCP traffic (including *Gnutella, eDonkey*, and *BitTorrent*) and five UDP-traffic generating packages (including *Direct Connect*, *BitTorrent*, and *KaZaA*).

Table 19 outlines a few rules used by *IPP2P* for pattern matching at packet level. Should a *DirectConnect* command be delivered in multiple TCP packets as it happens

traffic type	*traffic characteristics inspected*
TCP-based P2P traffic	
eDonkey	protocol tag (1 byte); message length (2 bytes); message type (1 byte) and should be "Hello" or "Hello-Answer"
DirectConnect	packet starts with "$" and ends with "\|"; command after "$" should be either "Lock" or "MyNick";
Gnutella	packet ends with characters "carriage return (0x0D)" and "new line (0x0A)"; pkt starts with "GNUTELLA CONNECT/", "GNUTELLA/", or "Get /get/"; for the latter, packet should further contain "X-Gnutella-" or "X-Queue:";
KaZaA	packet ends with characters "carriage return (0x0D)" and "new line (0x0A)"; packet starts with "GIVE " or "GET /"; for the latter, packet should further contain "X-Kazaa-Username: " or "User-Agent: PeerEnabler/";
BitTorrent	packet starts with "|13|BitTorrent protocol" or "GET /"; for the latter, packet should further contain "scrape?info_hash=" or "announce?info_hash=";
UDP-based P2P traffic	
eDonkey	checked fields: protocol tag (1 byte); message type (1 byte); packet length; not detect eMule and Overnet
DirectConnect	pkt starts with "$" and ends with "\|"; command after "$" should be "SR " or "Ping ";
Gnutella	packet starts with "GND" or "GNUTELLA ";
KaZaA	packet ends with "KaZaA|00|";
BitTorrent	mainly check packet length and some fields with constant values; for instance, when payload is 16 bytes, first 8 bytes should be |00 00 04 17 27 10 19 80|;

Table 19. Rules used in *IPP2P* to identify TCP/UDP P2P traffic

frequently, then the command start and end delimiters $ and | may appear in different packets. The latter will evade the check carried out by *IPP2P* and will ultimately lead to false negative. Another known limitation of *IPP2P* is that it identifies only a limited set of message types for any specific P2P system. For example, among the *eDonkey*'s more than 100 message types, only two can be successfully dealt by *IPP2P*. Furthermore, *IPP2P* only attempts to identify *KaZaA*'s downloading data flows, while for its signalling traffic, *IPP2P* detects UDP-based signalling messages, letting *KaZaA* TCP-based signalling traffic go completely undetected. Finally, due to lack of any traffic correlation, *IPP2P* may generate false positives. To go beyond the packet-based telltale examination, *IPP2P* have to jointly work with other packages such as *CLASSIFY* and *CONNMARK* to process P2P streams at the session level.

9. Acknowledgments

We are very grateful to the three reviewers for their meticulous comments that helped us significantly improve our work. We are indebted to Ping Wu, Qinghong Yi, and Gary Duan of *Fortinet* for helping with the development of the framework prototype; we also thank Hong Huang and Joe Zhu for their valuable help in the manual testing phase of our framework.

References

1. K. Aberer, M. Hauswirth, and M. Punceva. Self-Organizing Construction of Distributed Access Structures: A comparative Evaluation of P-Grid And Freenet. In

Proceedings of the 5th Workshop on Distributed Data and Structures (WDAS'2003), 2003.

2. P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the ACM SIGMETRICS'98*, Madison, WI, June 1998.
3. G. Bartlett, J. Heidemann, and C. Papadopoulos. Inherent Behaviors for On-line Detection of Peer-to-Peer File Sharing. Technical report, December 2006. ISI-TR-627, University of southern California.
4. H. Barton. Skype Growth: Analysis and Forcast for 2007. *http://homepage.mac.com/hhbv/blog/skypegrowth/skypegrowth.html*, December 2006.
5. S. A. Baset and H. G. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. In *Proceedings of IEEE INFOCOM'06*, Barcelona, Spain, April 2006. IEEE.
6. BugTraq. *BugTraq Vulnerability Database*, 2006. http://www.securityfocus.com/.
7. K. Chen, C. Huang, P. Huang, and C. Lei. Quantifying Skype User Satistication. In *Proceedings of the SIGCOMM 2006*, Pisa, Italy, September 2006. ACM.
8. Z. Chen, A. Delis, and P. Wei. Analyzers for the Identification of Instant Messagers and Peer-to-Peer Systems Sessions. Technical report, February 2007. Deprt. of Informatics and Telecommunications, University of Athens, Athens, Greece, www.di.uoa.gr/∼ad/imp2p-analyzers.pdf.
9. Z. Chen, P. Wei, and A. Delis. A Pragmatic Methodology for Testing Intrusion Prevention Systems (IPSs). Technical report, December 2005. Deprt. of Informatics and Telecommunications, University of Athens, Athens, Greece.
10. W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, Reading, MA, 1994.
11. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009, pages 46–66. Springer-Verlag, 2001.
12. Clip2. The Gnutella Protocol Specification v0.4. *http://www.clip2.com/GnutellaProtocol04.pdf*, March 2001.
13. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1997.
14. International Data Corporation. http://www.idc.com/. 2004.
15. M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Traffic Classification through simple Statistical Fingerprinting. *ACM SIGCOMM Computer Communication Review*, 37(1):7–16, January 2007.
16. M. Day, S. Aggarwal, G. Mohr, and J. Vincent. Instant Messaging / Presence Protocol Requirements. *Internet Engineering Task Force*, February 2000.
17. M. Day, J. Rosenberg, and H. Sugano. A Model for Presence and Instant Messaging. *Internet Engineering Task Force*, February 2000.
18. C. Dewes, A. Wichmann, and A. Feldmann. An Analysis of Internet Chat Systems. In *Internet Measurement Conference 2003*, Miami Beach, Florida, USA, October 2003.
19. Edonkey2000. Home Page. *http://www.edonkey2000.com*, July 2005.
20. S. Ehlert, S. Petgang, T. Magedanz, and D. Sisalem. Analysis and Signature of Skype VoIP Session Traffic. In *Proceedings of Communications, Internet, and Information Technology (CIIT 2006)*, St. Thomas, US Virgin Islands, November 29 - December 1 2006.

21. D. Ennis, D. Anchan, and M. Pegah. The Front Line Battle Against P2P. In *SIGUCCS'04*, pages 101–106, Baltimore, Maryland, USA, October 2004. ACM.
22. Ethereal. Ethereal: Powerful Multi-Platform Analysis. *http://www.ethereal.com*.
23. P. Felber, E. Biersack, L. G. Erce, K. W. Ross, and G. U. Keller. Data Indexing and Querying in P2P DHT Networks. In *ICDCS 2004*. Tokyo, Japan, 2004.
24. C. Fraleigh. Packet-Level Traffic Measurements From the Sprint IP Backbone. *IEEE Network*, 2003.
25. A. Gerber, J. Houle, H. Nguyen, M. Roughan, and S. Sen. P2P: The Gorilla in the Cable. In *National Cable and Telecommunications Association (NCTA) 2003 National Show*, 2003.
26. gift-fasttrack development project. http://developer.berlios.de/projects/gift-fasttrack/.
27. Gartner Group. http://www.gartner.com. 2004.
28. Prim Working Group. Presence and Instant Messaging Protocol (Prim). *http://www.ietf.org/html.charters/prim-charter.html*, July 2001.
29. SIMPLE Working Group. SIP for Instant Messaging and Presence Leverage (SIMPLE). *http://www.ietf.org/html.charters/simple-charter.html*, June 2004.
30. S. Guha, N. Daswani, and R. Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS) 2006*, Santa Barbara, AC, February 2006. Microsoft Research.
31. K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP-19)*, Lake George, NY, October 2003.
32. P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated construction of Application Signatures. In *Proceedings of the 2005 Workshop on Mining Network Data*, pages 197–202, 2005.
33. E. L. Howes. Comments on the Problems of Spyware. In *Proceedings of Advance of the Federal Trade Commission (FTC) Spyware Workshop*, Washington, DC, April 19th 2004.
34. Fortinet Inc. Intrusion Prevention System. *www.fortinet.com*, 2006.
35. Jabber. Home Page. *http://www.jabber.org*, 2005.
36. A. Kalafut, A. Acharya, and M. Gupta. A Study of Malware in Peer-to-Peer Networks. In *Internet Measurement Conference 2006*, Rio de Janeiro, Brazil, October 2006. ACM.
37. V. Kalogeraki, A. Delis, and D. Gunopoulos. Peer-to-Peer Architectures for Scalable, Efficient and Reliable Media Services. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003.
38. T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos. Is P2P Dying or Just Hiding? In *IEEE GLOBECOM Conference*, Dallas, TX, November 2004.
39. T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy. Transport Layer Identification of P2P Traffic. In *Internet Measurement Conference (IMC)*, Taormina, Sicily, Italy, October 2004.
40. T. Karagiannis, D. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 229–240, 2005.
41. KaZaA. Home Page. *http://www.kazaa.com*, May 2004.
42. T. Klingberg and R. Manfredi. Gnutella 0.6 RFC. *http://rfc-gnutella.sourceforge.net/draft.txt*, June 2002.

43. C. Kreibich and J. Crowcroft. Efficient Sequence Alignment of Network Traffic. In *Internet Measurement Conference 2006*, Rio de Janeiro, Brazil, October 2006. ACM.
44. J. Ledlie, J. Taylor, L. Serban, and M. Seltzer. Self-Organization in Peer-to-Peer Systems. In *Proceedings of the 2002 SIGOPS European Workshop*, St. Emilion, France, September 2002.
45. N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the Kazaa Network. In *3rd IEEE Workshop on Internet Applications (WIaPP'03)*, Santa Clara, CA, 2003.
46. LimeWire. Home Page. *http://www.limewire.com*, May 2004.
47. Q. Lv, S. Ratsnasamy, and S. Shenker. Can Heterogeneity Make Gnutella Scalable? In *International Workshop on P2P Systems, 2002*, Osaka, Japan, 2002.
48. J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. In *Internet Measurement Conference 2006*, Rio de Janeiro, Brazil, October 2006. ACM.
49. Yahoo! Messenger. Home Page. *http://messenger.yahoo.com*, 2005.
50. A. W. Moore and D. Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *SIGMETRICS'05*, Banff, Alberta, Canada, June 2005. ACM.
51. D. Moore, K. Keys, R. Koga, E. Lagache, and K. Claffy. Coralreef Software Suite as a Tool for System and Network Administrators. In *Usenix LISA*, San Diego, CA, 2001.
52. P2PWatchDog. Home Page. *http://www.p2pwatchdog.com*, May 2004.
53. R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A First Look at Modern Enterprise Traffic. In *Internet Measurement Conference 2005*, pages 15–28, Berkeley, CA, USA, October 2005.
54. R. Pang, V. Paxson, R. Sommer, and L. Peterson. Binpac: A yacc for Writing Application Protocol Parsers. In *Proceedings of Internet Measurement Conference 2006*, Rio de Janeiro, Brazil, October 2006. ACM.
55. V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Compute Networks*, 31(23-24):2435–2463, December 1999.
56. P. Piccard. Risk Exposure: Instant Messaging and Peer-to-Peer Networks v2.0. *White paper of Internet Security Systems*, 2004.
57. IPP2P project. The Home Page of IPP2P Project. *http://www.ipp2p.org*, 2007.
58. MLdonkey Project. MLdonkey, a Multi-Networks File-Sharing Client. http://savannah.nongnu.org/projects/mldonkey/, 2005.
59. M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *USENIX 13-th Systems Administration Conference – LISA'99*, Seattle, Washingto, 1999.
60. M. Rose, G. Klyne, and D. Crocker. The Application Exchange Core. *Internet Engineering Task Force*, July 2002.
61. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. *Internet Engineering Task Force*, June 2002.
62. J. Rosenberg, J. Weinberger, C. Huitema, and R.Mahy. STUN – Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). *Internet Engineering Task Force*, March 2003.
63. S. Saroiu, S. D. Gribbe, and H. M. Levy. Measurement and Analysis of Spyware in a University Environment. In *Proceedings of the first Symposium on Operating Systems Design and Implementation (OSDI 2004)*, Francisco, CA, March 2004.
64. S. Saroiu, P. Gummadi, R. J. Dunn, and S. D. Gribbe. An Analysis of Internet Content Delivery Systems. In *Proceedings of the fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
65. S. Saroiu, P. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*

2002, January 2002.

66. R. Schollmeier. A Definition of Peer-to-Peer Netwoking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings of the IEEE 2001 International Conference on Peer-to-Peer Computing (P2P2001)*, Linkoping, Sweden, August 2001.
67. R. Schollmeier and F. Hermann. Peer-to-Peer Traffic Characteristics. In *Proceedings of 9th EUNICE Open European Summer School (EUNICE'03)*, Budapest-Balatonfured, Hungary, September 2003.
68. R. Schollmeier and G. Schollmeier. Why Peer-to-Peer (P2P) Does Scale: An Analysis of P2P Traffic Patterns. In *2nd International Conference on Peer-to-Peer Computing (P2P'02)*, 2002.
69. S. Sen, , and J. Wang. Analyzing Peer-to-Peer Traffic Across Large Networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, Marseilles, France, November 2002. ACM.
70. S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *World Wide Web Conference 2004*, pages 512–521, New York, New York, USA, May 2004. ACM.
71. S. Shin, J. Jung, and H. Balakrishnan. Malware Prevalence in the KaZaA File-Sharing Network. In *Internet Measurement Conference 2006*, Rio de Janeiro, Brazil, October 2006. ACM.
72. D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.
73. Inc. Sourcefire. Snort 2.0: Detection Revisited. *http://www.snort.org*, February 2003.
74. Spirent. Spirent Avalanche: the Best Choice for System and Network Equipment Testing. *http://www.spirentcom.com*, 2007.
75. D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Internet Measurement Conference 2006*, Rio de Janeiro, Brazil, October 2006. ACM.
76. PyxisSystems Technologies. AIM/Oscar Protocol Specification. *http://aimdoc.sourceforge.net/OSCARdoc/*, May 2003.
77. TippingPoint. The Fundamentals of Intrusion Prevention System Testing. *White Paper*, 2003.
78. A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. The Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Charleston, SC, December 1999.
79. S. Zander, T. Nguyen, and G. Armitage. Self-Learning IP Traffic Classification Based on Statistical Flow Characteristics. In *Proceedings of the Passive and Active Network Measurement Workshop*, March 2005.