SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Catching Remote Administration Trojans (*RATs*)

Zhongqiang Chen [1] Peter Wei [2] and Alex Delis [3]

[1] *Yahoo! Inc., Santa Clara, CA 95054, E-mail: zqchen@yahoo-inc.com*
[2] *Fortinet Inc., Sunnyvale, CA 94085, E-mail: pwei@fortinet.com*
[3] *University of Athens, Athens 15771, Greece, E-mail: ad@di.uoa.gr*

SUMMARY

A Remote Administration Trojan (*RAT*) allows an attacker to remotely control a computing system and typically consists of a server invisibly running and listening to specific *TCP/UDP* ports on a victim machine as well as a client acting as the interface between the server and the attacker. The accuracy of host and/or network-based methods often employed to identify *RATs* highly depends on the quality of Trojan signatures derived from static patterns appearing in *RAT* programs and/or their communications. Attackers may also obfuscate such patterns by having *RATs* use dynamic ports, encrypted messages, and even changing Trojan banners. In this paper, we propose a comprehensive framework termed *RAT Catcher*, that reliably detects and ultimately blocks *RAT* malicious activities even when Trojans use multiple evasion techniques. Employing network-based methods and functioning in *inline* mode to inspect passing packets in real time, our *RAT Catcher* collects and maintains status information for every connection and conducts session correlation to greatly improve detection accuracy. The *RAT Catcher* re-assembles packets in each data stream and dissects the resulting aggregation according to known Trojan communication protocols, further enhancing its traffic classification. By scanning not only protocol headers but also payloads, *RAT Catcher* is a truly application layer inspector that performs a range of corrective actions on identified traffic including alerting, packet dropping, and connection termination. We show the effectiveness and efficiency of *RAT Catcher* with experimentation in both laboratory and real-world settings.

Indexing Terms: **Remote Administration Trojans, Trojan detection accuracy, session and event correlation, application layer inspection.**

1. Introduction

Remote Administration Trojans (*RATs*) are malicious pieces of code often embedded in legitimate programs through *RAT*-ification procedures [31, 55, 43]. They are stealthily planted and help gain access of victim machines, through patches, games, E-mail attachments, or even in legitimate-looking binaries [31, 6]. Once installed, *RATs* perform their unexpected or even unauthorized operations [57] and use an array of techniques to hide their traces to remain invisible and stay on victim systems for the long haul. For instance, *RAT*-ified versions of programs *Unix* ps and *Windows* taskmgr.exe keep *RATs* from appearing in the list of active processes; moreover, by modifying system configurations including the boot-scripts and the *Registry* database, *RAT*-binaries often survive system reboots or crashes. A typical *RAT* consists of a server component running

[*] Correspondence to: Zhongqiang Chen

Contract/grant sponsor: European Social Funds and National Resources Pythagoras Grant & Univ. of Athens Research Foundation; contract/grant number: 04-7410

on a victim machine and a client program acting as the interface between the server and the attacker [28]. The client establishes communications with its corresponding server as soon as the *IP* address and port of the latter become available through feedback channels such as Email, Instant Messaging and/or Web access [43]. While interacting with a *RAT* server, an attacker can record keystrokes, intercept passwords, manipulate file systems, and usurp resources of victim systems [43].

By continually changing their name, location, size, and behavior, or employing information encryption, port hopping and message tunneling for its communications, *RATs* may elude the detection of security protection systems such as firewalls, anti-virus systems (AVs), and intrusion detection/prevention systems (IDSs/IPSs) [42, 16]. Once bound to legitimate programs, *RATs* in execution inherit a victim's privileges and raise havoc; moreover, they launch attacks against other systems purporting themselves to be superusers [53, 43]. *RATs* provide the ideal mechanism for propagating malware including viruses, worms, backdoors, and spyware [13, 31, 21, 54]. The number of *RATs* has been steadily increasing from 1,359 in 2002 to 20,355 in 2004 and their update rates are also impressive; just SubSeven delivered 12 versions in 2002 alone [43]. The number of *RAT*-infected machines is staggering: in 2000, 35% of security incidents in Korea were Trojan-inflicted mostly by Back Orifice (BO) [3] and in 2001, 10% of intrusions in Israel were due to NetBus and BO [41]. *PestPatrol* reports that roughly 2% of all incidents are attributed to *RATs* [43]. Compromised machines are often used as spring-boards for distributed denial of service attacks, further exacerbating the problem [28].

The best option for avoiding *RATs* is to verify every piece of software before installation using a-priori known program signatures [19, 7]. This, however, becomes impractical as a comprehensive database of known program signatures is unavailable and *RATs* are frequently delivered via multiple channels such as patches, attachments, file sharing, or simply Web-site accessing. The polymorphic nature and parasitic mechanisms of *RATs* render their identification a challenge even if we seek specific and known types of Trojans [11, 12, 2, 43]. Host- and network-based techniques have been widely employed by firewalls, AVs and IDSs/IPSs to detect and block *RATs* [55, 7]. Static fingerprinting is the predominant method in host-based *RAT* detection where unique facets of Trojans are extracted to establish a *Trojan Database*, which entails file names, sizes, locations, checksums, and special patterns in *RATs* [35, 5, 14, 38]. By periodically scanning every file in a system and matching fingerprints against those in the established database, *RATs* can be revealed. In addition, monitoring the access of files in the startup folder, registries, auto-start files, and configuration scripts of a system is another popular host-based technique that helps identify suspicious activities [40, 38]. Network-based methods follow a different philosophy as they examine both the status and activity on *TCP/UDP* ports to determine any deviation from expected network usage [33, 49]. Abnormal behavior and/or malformed network messages can be detected by monitoring port access patterns and/or analyzing protocol headers of packets exchanged among systems [51]. Similar to host-based methods, unique *RAT*-manifested telltale patterns in network communications are exploited as signatures to distinguish malicious traffic [44, 51]. Clearly, the *RAT* detection accuracy of host- and network-based methods depends on the quality of the Trojan database and signatures used; the latter can be easily obfuscated by attackers using an array of evasion techniques.

In this paper, we propose a comprehensive framework for detecting and dealing with known *RATs* which employs network-based detection methods and operates in *inline* mode to inspect and manipulate every passing packet in real-time. Our objective is to enhance the reliability and accuracy of the detection process in comparison with existing anti-Trojan options. To track suspicious *RAT* activities, our framework monitors network sessions established by both potential Trojans and normal applications, records and maintains state information for their entire lifetime; furthermore, this information is archived even after a session has terminated in order to conduct stateful inspection, intra-session data fusion, and inter-session correlation. By performing packet re-assembly and interpreting the resulting aggregations against protocols followed by Trojans, our *RAT Catcher* morphs data streams into sequences of Trojan messages, facilitating the application-layer inspection and classification of malicious traffic. A number of options are available to manipulate identified *RAT* sessions ranging from simple alert and log generation to packet dropping and pro-active session blocking/termination. Experimentation with the *RAT Catcher* shows its effectiveness as well as its efficiency in a range of laboratory and real-world application settings. We organize our paper as follows: Section 2 outlines related work and Section 3 discusses the working mechanisms of *RATs*. Section 4 presents our proposed framework and Section 5 outlines the findings of our prototyping effort and experimental evaluation. Concluding remarks and future work can be found in Section 6.

2. Related Work

Upon activation, a *RAT* inherits privileges of its program-carrier and complies with the program's expected behavior most of the times, making it challenging to distinguish legitimate activities from malicious ones solely based on program ownership and user profiling [31, 27]. Verifying that a program is virus-infected is known to be computationally impossible [12]; even searching an executable for known *RATs* is challenging. The "least privilege principle" is considered an effective way to limit the potential access-scope of Trojanized programs in mulit-user systems [45, 24, 22] even though it is routinely violated [24, 26]. Techniques such as "partitioned protection domains" and "multi-level security models" are also used as means for protection against Trojans [4]; in the former, system partitions provide discretionary access control and in the latter each system entity is statically assigned a security classification [37, 10]. In this context, a Trojan may only obtain information either within the same partition or from entities tagged with *lower* security classification, thereby limiting its potential damage. Preventing users in different protection domains from sharing programs can work together with integrity models [†] that compute and store checksums for all files in the systems, and periodically re-calculate these statistics to detect possible file modifications [35, 58]. Such security procedures often affect the flexibility of a system and are deemed as burden to users.

Fault-tolerance methods have been used to detect unexpected behavior of program segments by treating such deviations as errors [34]. Changes in size, frequency of modification, and ignition rates of programs in conjunction with user profiles [20, 39, 56] have been used by AVs to detect viruses and *RATs* [18, 14]. To survive system reboots or crashes, *RATs* modify system files such as *win.ini*, *system.ini*, and/or *registry* entries in *Windows* and boot-scripts in *Unix* systems. To help detect and prevent such system-level modifications, a number of host-based security systems allow users to directly enable/disable startup items [38]. Unfortunately, this helps little as it is difficult for users to distinguish legitimate from illegal items and *RATs* often resort to renaming themselves. Parasitic *RATs* inject their malicious codes into running processes on-the-fly, effectively shielding themselves from detection [43]. For instance, Trojan Beast inserts itself into active processes such as *winlogon* and *Explorer* and becomes a background thread in these programs [47]. Most security systems fail to deal with parasitic *RATs* as in their effort to curb suspicious activities, they kill legitimate processes as well [47]. A group of *RATs* can disable firewalls and AVs by killing processes or removing files needed by such security systems [47]. Clearly, the above host-based detection methods may be ineffective as contemporary *RATs* can readily defy their access control, integrity checking, and behavior profiling.

[†] such as *Tripwire*

RAT servers typically listen on specific ports waiting for instructions from attackers [15]. Utilities such as *netstat*, *Fport* and *TCPView Pro* are designed to monitor active ports for suspicious network activities [15, 33, 49]. A number of firewalls also detect Trojans by searching for applications inducing unauthorized communications. This general approach however may yield false negatives as *RATs* may bind to legitimate programs and use standard ports [38]. Overall, the effectiveness of techniques based on "static" Trojan characteristics is questioned as soon as *RATs* commence using non-default ports, hijack ports from other applications, and/or occasionally change communication ports.

By inspecting network traffic and searching for possible Trojan patterns, IDSs/IPSs can establish the intention and/or behavior of data streams [44, 51]. The telltale patterns of Trojans are typically obtained via reverse engineering and data mining techniques [52]. Most IDSs/IPSs heavily base their Trojan-detections on fixed ports and/or simple pattern matching mechanisms, inevitably generating significant false positive or negative rates. Also, this pattern matching is typically conducted only within individual transport-layer *TCP/UDP* packets rendering IDSs/IPSs vulnerable to evasion attacks [50]. To mitigate evasive attacks, some IDSs/IPSs including *Snort* and protocol analyzer *Ethereal* offer traffic stream-reassembly functionality. Unfortunately, this re-assembly feature is only available for pre-specified ports reducing the defense capabilities of such systems significantly [44, 23]. *Ethereal* is mainly designed as a protocol analyzer that passively collects network traffic without generating any alert for ongoing traffic making it impossible to deliver counter measures in real time [23]. Furthermore, most network-based anti-Trojan systems identify only *RAT* control channels and do not deal with the content of data channels producing elevated false negative rates [7]. In summary, conventional *RATs* detection techniques demonstrate limitations and may fail to identify Trojans that resort to a range of advanced evasion techniques.

3. Characteristics of *RATs*

As *RATs* can essentially capture every screen and keystroke, intruders may obtain account information, passwords, and sensitive computing system data. *RATs* can also spawn arbitrary numbers of processes on specific *TCP/UDP* ports, impersonate victims, redirect traffic for specific services to other systems, and launch distributed denial of service (*DDoS*) attacks. In this section, we examine the salient features of *RATs* and briefly analyze their capabilities.

3.1. Frequently Observed Functionalities of *RATs*

RATs typically provide attackers with comprehensive command repertoires for file management, process scheduling, and system configuration manipulation. File management features include potentially destructive operations such as *delete/move* a file or directory on victim systems. The process scheduling component in a *RAT* permits intruders to create, view, and/or terminate running processes at will. The configuration manipulation element allows *RATs* to alter the behavior of the victim system by for instance disabling its security features after modifying the *Windows Registry*. *RATs* can often operate as device controllers being able to open/close CD-ROMs, disable the mouse and network cards, intercept keystrokes and/or screen snapshots, flip the victim's screen or change its resolution, monitor password dialog boxes and clipboards, capture audio/video of the victim's environment, and finally, crash the victim [43]. The re-direct feature of *RATs* allows an attacker to chain various services together and ultimately forward the results to a specified destination, making it trivial for intruders to hijack network connections, intercept private data, and inject fake messages. By functioning as packet sniffers, *RATs* can also monitor a victim's network activities and determine its topology. Furthermore, by scanning the entire system of the victim machine, including its garbage bin, a number of

RATs can collect personal information such as user accounts, passwords, credit cards, and Email addresses.

Most *RATs* integrate all the above functionalities and therefore act as a swiss army knife for intruders. In this spirit, Back Orifice (BO), SubSeven, and DeepThroat provide around 60, 100, and 120 commands, respectively. Table I depicts a few commands available in SubSeven and Back Orifice (BO). Here, the SubSeven command *IRG* can be used to add, remove, or modify system configurations from Registry, command *FFN* retrieves files from the victim system, and *COM* can help execute a specific program. Commands *GMI*, *OCD*, and *PWD* are pertinent to information collection and *TKS* logs all keystrokes. Screen snapshots can be captured with *CSS* and audio/video can be recorded with the help of *RSF* and *IN7*. Similar actions are achieved through the repertoire of Back Orifice (BO) as well.

ID	*command*	*description*	*ID*	*command*	*description*
Some commands provided by SubSeven					
1	*GMI*	get remote machine info	2	*PWD*	get server password
3	*RAS*	retrieve RAS passwords	4	*GIP*	get ICQ passwords
5	*RSH*	file manager	6	*NTF*	download file
7	*FFN*	find files (e.g., wildcard) in given directory	8	*TKS*	key logger
9	*IN7*	open web cam	10	*OCD*	open CD-ROM
11	*RWN*	shutdown/restart Windows	12	*FTP*	start or stop FTP server
13	*IRG*	registry editor	14	*COM*	execute a command
Some commands provided by Back Orifice					
01	*PING*	ping the current host	02	*REBOOT*	reboot the remote host
04	*PASSES*	display remote cached passwords	06	*INFO*	Display remote system info
07	*KEYLOG*	log keystrokes to file	09	*DIALOG*	display a dialog box
0E	*APPADD*	spawn a console application on a TCP port	14	*HTTPON*	enable the HTTP server
19	*PLUGINEXEC*	execute a plugin	20	*PROCKILL*	kill process given by *PROCLIST*
23	*REGMAKEKEY*	create a key in the registry	28	*CAPAVI*	obtain video stream from device
2A	*SOUND*	play a WAV file	3D	MD	make a directory

Table I. A few commands provided by SubSeven and Back Orifice (BO)

A number of *RATs* offer the proxy functionality that turns a victim machine into a server for services including *Telnet*, *FTP*, *HTTP*, *ICQ*, and *IRC*, offering free storage and complete anonymity for attackers. The Trojan Eclypse for instance can be instructed to act as a *FTP* server; this is depicted by the sample traffic of Table II, which establishes separate channels on different *TCP* ports for its control and data transmissions much in the spirit of *FTP*. The apparent protocol similarities in both syntax and semantics between Eclypse and *FTP* make it difficult to distinguish normal *FTP*-flows from Trojan-generated traffic. The traffic of Table II shows that the data channel is constructed dynamically (in row 9) once the client submits command *NLST* to obtain a list of files from the compromised machine (row 8). Clearly, the data channel is server-initiated although the request is client-originated and the data port of the client is specified with command *PORT* dynamically (row 6). Similarly, SubSeven can be configured to act as a proxy server with the help of commands *FTP*, *URL* and *COM* (Table I). *RATs* can be also used as *DDoS* attack tools provided that enough victims are harvested and instructed to simultaneously flood specific machines. Finally, some *RATs* can disable, mis-configure, or even kill firewalls, AVs, and IPSs/IDSs incapacitating surveillance.

#	*dir*	*payload*	*description*
protocol: TCP; attacker (denoted as A): 192.168.5.143; victim (denoted as V): 192.168.5.141			
1	V:3791⟶A:1074	220 EclYpse 's FTP Server is happy to ...	banner in FTP format; telltale: "EclYpse"
2	A:1074⟶V:3791	*USER* (none)	intruder logins with name "(none)"
3	V:3791⟶A:1074	331 Password required for (none).	require password for login
4	A:1074⟶V:3791	PASS xxxxxx	password for account name
5	V:3791⟶A:1074	230 User (none) logged in.	attacker logins successfully
6	A:1074⟶V:3791	*PORT* 192,168,5,143,4,51	specify port for data channel (i.e., 4*256 + 51 = 1075)
7	V:3791⟶A:1074	200 Port command successful.	server stores data port number
8	A:1074⟶V:3791	*NLST*	request "send host info"
9	V:1030⟶A:1075	(*SYN*)	establish data channel with port 1075
10	V:3791⟶A:1074	150 Opening data connection for ...	data channel has been established
11	A:1074⟶V:3791	*QUIT*	disconnect from the server
12	V:3791⟶A:1074	221 Goodbye.	server closes the session

Table II. Traffic generated by the Eclypse (v.1.0) Trojan

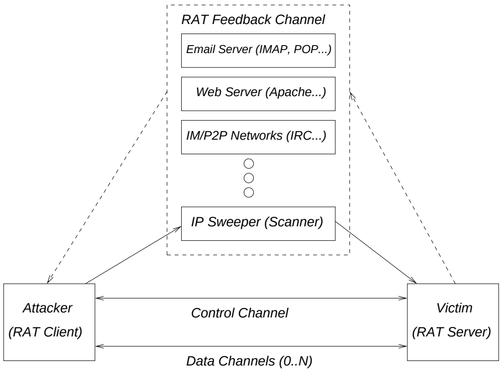

Figure 1. Working mechanisms of a *RAT*

3.2. Working Mechanisms of *RATs*

Before their installation, *RAT*-servers can be customized via *RAT*-provided configuration packages termed *binders*. This customization includes the setting of the default *TCP/UDP* ports utilized by *RAT* servers, definition of auto-start methods, encryption algorithms and designation of initial login passwords. For instance, EditServer and bo2kcfg are the *binders* for SubSeven v2.2 and Back Orifice 2000 (BO2K) respectively [43, 42]. Prior to being delivered, *RAT*-servers may be named as software patches or games with the corresponding *binders*, tricking users into downloading, un-bundling, and finally, executing such malicious programs. Once the servers are configured, they are shipped to victims via a number of delivery channels as described in Figures 1 and 2. During their installation, *RAT*-servers may piggyback themselves to other legitimate programs, termed *hosts*, so that they are executed every time their hosts are invoked. In this way, a BO2K-server can install itself as a thread to the host program IEXPLORE.EXE [42]. *RAT*-servers typically run in the background and listen on designated network ports waiting for attacker-issued instructions, leaving victims unaware of their damaging activity.

There is a multitude of avenues to spread Trojans to victim machines as Figure 2 depicts; the most notable for the time being are *Instant Messengers (IM)* and *peer-to-peer (P2P)* systems. With the help of either MSN-messenger or KaZaA, an attacker may freely visit chat-rooms, scan buddy-lists, or even randomly select candidate victims among encountered active users, and subsequently deliver *RATs* to victims. Additional delivery options include *HTTP* servers especially created to disseminate Trojans along with regular web-pages, opening *Email* attachments, execution of malware and distributions for software patches, freewares, and/or games. Hence, anti-Trojan systems are easily defeated if their *RAT*-detection methods cover only a small portion of such propagation channels.

The *IP* addresses, *TCP/UDP* ports, access passwords, and other information of *RAT*-servers can be obtained by intruders through feedback channels shown in Figure 1. *IM/P2P* systems, *Email* services, and shared folders can even provide auto-notifications between *RAT*-servers and clients. In Guptachar [43] for example, an attacker may set up an *IRC*-server via its *IRCBOT* [‡] function by providing a login account *nickname*; every time a compromised system is activated, it connects to the above *IRC*-server using *nickname* to upload the victim's *IP*-address and port number. Furthermore, most *RATs* resort to multiple methods to outlive system crashes or reboots and evade AVs/IPSs/IDSs [43]. By editing

[‡] Internet Relay Chat Robot

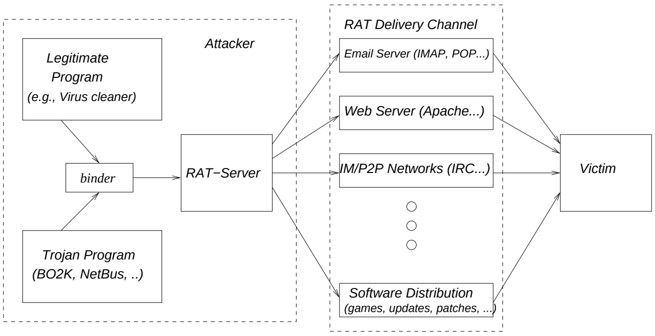
Attacker
Legitimate Program (e.g., Virus cleaner)
binder
Trojan Program (BO2K, NetBus, ..)
RAT−Server
RAT Delivery Channel
Email Server (IMAP, POP...)
Web Server (Apache...)
IM/P2P Networks (IRC...)
Software Distribution (games, updates, patches, ...)
Victim

Figure 2. *RAT* delivery procedure

Registry entries, modifying system files such as *win.ini*, *system.ini* and *autoexec.bat* as well as inserting items on the startup folders, *RATs* can easily "hide" and be transparently triggered on every reboot. In this regard, host-based detection methods are inferior to their network-based counterparts as far as *RAT* detection ic concerned.

3.3. Encrypted *RAT* Traffic

RATs such as Back Orifice (BO), BO2K and NetBus v2.0 encrypt their traffic flows to defeat security systems that base their *RAT* detection on pattern matching techniques. The first half of Table III shows portion of encrypted traffic generated by Back Orifice (BO). The scrambled content of encrypted *RAT* communications forces many anti-Trojan systems to predominantly use fixed network ports (e.g., the *UDP*-port for BO is configurable with default 31337), inevitably leading to false positives/negatives. In BO, the encrypted traffic is generated by applying an *XOR* operation on the original data and a random stream created with the help of a four-byte encryption *key* derived from the intruder-specified password [§]. At the receiver side, the *seed* (i.e., *key*) can be guessed via an exhaustive search due to its short range; this seed can then be used to recover the original traffic [42]. Obviously, the *XOR* operation neither changes the length of the original message nor destroys string patterns; this fact may help identify the BO even if its encrypted traffic is only available.

#	*dir*	*size*	*payload*	*description*
protocol: UDP; attacker (denoted as A): 192.168.5.143:1034; victim (denoted as V): 192.168.5.141:31337				
Encrypted messages by Back Orifice				
1	A⟶V	19	|9E F4 C2 EB 87 89 A2 04 4E 42 E8 69 3B B9 98 55 C3 80 66|	encrypted msg
2	V⟶A	37	|9E F4 C2 EB 87 89 A2 04 78 42 E8 69 3A B9 98 55 C3 A0 46...|	encrypted msg sharing pattern with msg 1
3	A⟶V	19	|9E F4 C2 EB 87 89 A2 04 4E 42 E8 69 38 B9 98 55 E2 80 66|	encrypted msg very similar to msg 1
4	V⟶A	41	|9E F4 C2 EB 87 89 A2 04 74 42 E8 69 3B B9 98 55 62 A0 46...|	encrypted msg very similar to msg 2
5	A⟶V	19	|9E F4 C2 EB 87 89 A2 04 4E 42 E8 69 39 B9 98 55 C4 80 66|	encrypted msg very similar to msg 1
6	V⟶A	49	|9E F4 C2 EB 87 89 A2 04 6C 42 E8 69 37 B9 98 55 44 D3 1F...|	encrypted msg very similar to msg 2
Decrypted Back Orifice traffic				
1	A⟶V	19	|2A 21 2A 51 57 54 59 3F 13 00 00 00 01 00 00 00 01 00 00|	msg PING starts with string "*!*QWTY?"
2	V⟶A	37	|2A 21 2A 51 57 54 59 3F 25 00 00 00 00 00 00 00 01 20 20 21 50...|	reply to "PING" (i.e., "PONG");
3	A⟶V	19	|2A 21 2A 51 57 54 59 3F 13 00 00 00 02 00 00 00 20 00 00|	cmd PROCESSLIST (0x20) from client
4	V⟶A	41	|2A 21 2A 51 57 54 59 3F 29 00 00 00 01 00 00 00 A0 20 20...|	reply to "PROCESSLIST" (partial)
5	A⟶V	19	|2A 21 2A 51 57 54 59 3F 13 00 00 00 03 00 00 00 06 00 00|	cmd INFO (0x06) from client
6	V⟶A	49	|2A 21 2A 51 57 54 59 3F 31 00 00 00 0D 00 00 00 86 53 79...|	reply to "INFO"

Table III. Back Orifice generated traffic in both encrypted and decrypted formats

[§] its default value is 31337, the same as its default *UDP* port.

Through exhaustive search, we can recover the seed *2160* used in the BO-server/client traffic of Table III. The decrypted traffic demonstrates that each BO message commences with the 8-byte magic-string of **!*QWTY?* followed by the 4-byte *packet length*, 4-byte *packet ID*, 1-byte *message type*, the variable length *message data* and the 2-byte *checksum* field. The *packet length* specifies the size of the entire message; for example, the first client message (row 1) is 19-bytes in length. Initially, the BO-client probes the server with command *PING* (0x01 in the 1-byte *message type* at index 16) to check the availability of the server. The active server responds with command *PONG* along with its version number in row 2. In rows 3 and 5, the client requests information from the victim (i.e., the BO server) with commands *PROCESSLIST* (0x20) and *INFO* (0x06). Table III shows that decrypted BO traffic possesses strong message structure and clear semantics. This structure in conjunction with the commands of Table I help attain improved traffic identification based on decrypted content, which is the main techniques used in our *RAT Catcher* to detect *BO*; a similar approach was also employed by *Snort*.

The *XOR*-based encryption algorithm used by the international version of BO2K can be also reverse-engineered and the original traffic can be recovered, making BO2K detection feasible. Nevertheless, the use of plugin modules in BO2K complicates matters. To this end, the plugins enc_aes, enc_cast and enc_idea encrypt BO2K traffic with the *AES*, *CAST*, and *IDEA* encryption algorithms respectively [46]. It has been shown that such algorithms are resistant to sophisticated differential and linear cryptanalysis techniques [1, 30]. Hence, it is impossible to recover original form from encrypted traffic without the requisite encryption keys. For this type of Trojans, the only pragmatic approach for detection has to be based on the *RAT* external behavior which involves monitoring of message sizes, handshake procedure, and traffic correlation between the two traffic streams within each *RAT* session.

#	*dir*	*payload*	*description*
protocol: TCP; attacker (denoted as A): 192.168.5.143:1026; victim (denoted as V): 192.168.5.141:20034			
1	A⟶V	*BN* |20 00 02 00 08 08 05 00 41 0C 69 1F 5D 28 5B BC ...|	msg starts with *BN*; size: 0x20; ver: 0x02; cmd: 0x05;
2	V⟶A	*BN* |10 00 02 00 08 08 05 00 41 0C 6B 1F 5D 28|	reply msg; size: 0x10; ver: 0x02; cmd: 0x05;
3	A⟶V	*BN* |0B 00 02 00 DC 33 30 00 41|	client msg; size: 0x0B; cmd: 0x30
4	V⟶A	*BN* |0C 00 02 00 DC 33 30 00 41 0C|	server msg; size: 0x0C; cmd: 0x30
5	V⟶A	*BN* |34 00 02 00 F0 AB 30 00 41 0D 3E F4 56 DC ...| *BN* |18 00 02 00 6B 29 30 00 41 0D ...| *BN* |2F 00 02 00 8B 79 30 00 41 0D ...|	multiple messages packed into one TCP packet; msg 1 is cmd 0x30 with size 0x34; msg 2 is 0x30 with size 0x18 ...
6	A⟶V	*BN* |1E 00 02 00 F0 AB 50 00 41 C9 57 ...|	client msg; size: 0x1E; cmd: 0x50
7	V⟶A	*BN* |15 00 02 00 8C 05 50 00 41 0C 35 ...|	msg size: 0x15; ver: 0x02; cmd: 0x50;

Table IV. Portion of NetBus Pro (v.2.0)-generated traffic

NetBus Pro [29] is a noteworthy Trojan in that it uses proprietary encryption algorithms but still offers opportunities for detection based on traffic correlation; it listens to default but reconfigurable *TCP*-port 20034 and supports plugins enabling the integration of new functionalities [36]. Each NetBus Pro message has a fixed-length header (i.e., 10 bytes) consisting of string "*BN*" followed by four 2-byte fields namely, *message-size*, *version-number* (typical value 0x0002), *unknown* (often a random number), and *command-code* fields. The variable-sized data section follows the header and its size is specified in the *message-size* field of the header. Table IV shows a portion of NetBus Pro(v2.0)-generated traffic. Apparently, the beginning of every message is readily determined by string "*BN*"; correspondingly, the end of the message can be resolved from the *message-size* field of its header. Multiple messages may be packed in a single *TCP* packet as is the case with row 5, each of which can be identified through payload content inspection and message structure analysis. Stream-based inspection can also correctly interpret NetBus Pro messages spanning multiple *TCP* packets. Also, inspection on both streams of a session to ensure their conformance with NetBus Pro protocol specification further improves the detection accuracy. Despite the fact that decryption of *RAT* traffic is not always feasible, techniques derived from dissecting protocol syntax of *RATs* combined with analysis of patterns found in message exchanges can be exploited to detect Trojans. The NetBus Pro protocol analyzer outlined by Algorithm 4 of Section 4 functions on this premise.

3.4. Diversified Use of Protocols by *RATs*

By and large, the proliferation of *RATs* can be attributed to the fact that existing *RATs* serve diverse constituencies and deliver substantially differentiated services using a multitude of transport protocols. For instance, NetBus, Socket de Troie and SubSeven use *TCP* while Back Orifice, DeepThroat and DeltaSource are *UDP*-based. As *RATs* evolve they also use different protocols. In this regard, BO2K uses both *TCP* and *UDP* even though its ancestor BO was exclusively *UDP*-based. The syntax and semantics of client-server messages also demonstrate diverse characteristics. *RATs* including Back Orifice, SubSeven and BO2K maintain well-formed binary message structures; on the other hand, Trojans such as Dolly and Frenzy follow text-based message formats. Last, some *RATs* including Eclypse, WanRemote and Drat uses syntax and semantics similar, if not identical, to the standard *FTP*, *HTTP*, and *Telnet* protocols.

In Table II, we show portion of the traffic generated by Eclypse, an *FTP*-based Trojan and Table V presents traffic generated by WanRemote [43] that clearly follows the *HTTP* specification in both directions [25]. For brevity, we do not show the *HTTP* headers of all messages from the server but only the first one in row 2. Server responses are embedded in the data sections of *HTTP* messages. Table V indicates that the client supports *HTTP(v.1.1)* and the server uses *HTTP(v.1.0)* forcing each command to be transported using a separate *TCP* connection. For example, the packet of row 1 has *TCP*-source port 1071 while the corresponding source ports for messages in rows 3 and 5 are 1089 and 1105, respectively. By placing different commands in *HTTP* requests, the client can perform various operations on the victim's machine via the Trojan server. To this effect, the attacker can traverse directories in victim's file system with request "*cd=C:/*" (row 3) and obtain designated files with command "*get=C:/*" (row 5). Through "*kill=*" of row 9, the attacker calls for the termination of a process at the server that reciprocates with status information in row 10. The apparent pattern "*WANRemote 3.0*" definitely helps detect individual WanRemote(v.3.0) sessions; moreover, the correlation between different sessions (i.e., inter-session correlation) can further improve the detection accuracy.

#	*dir*	*payload*	*description*
protocol: TCP; attacker (denoted as A): 192.168.5.143; victim (denoted as V): 192.168.5.141			
1	A:1071⟶V:80	GET / HTTP/1.1	standard HTTP msg, implying login
2	V:80⟶A:1071	HTTP/1.0 200 OK Content-Type:text/html <title>WANRemote 3.0 - Main Menu</title>	response with "main menu" to attacker
3	A:1089⟶V:80	GET /fm?cd=C:/ HTTP/1.1	change directory
4	V:80⟶A:1089	<title>WANRemote 3.0 - File Manager</title>	enter "file manager"
5	A:1105⟶V:80	GET /fm?get=C:/autoexec.bat HTTP/1.1	get file "autoexec.bat"
6	V:80⟶A:1105	WANRemote 3.0 - File Manager: c:/autoexec.bat	file transferred successfully
7	A:1142⟶V:80	GET /process HTTP/1.1	get process list
8	V:80⟶A:1105	<title>WANRemote 3.0 - Task List</title>	return process list
9	A:1164⟶V:80	GET /process?kill=800 HTTP/1.1	kill specified process
10	V:80⟶A:1164	<title>WANRemote 3.0 - Task List</title>	processed killed
11	A:1175⟶V:80	GET /x-logout HTTP/1.1	logout from server
12	V:80⟶A:1175	<title>WANRemote 3.0 - Log Out</title>	session end

Table V. WanRemote(v.3.0)-generated traffic

Drat is a representative of the *Telnet*-based Trojans in which the server echoes back any attacker input and respective pieces of output as Table VI shows. Acting effectively as a command interpreter, the Drat-server displays the prompt *D:/temp>* waiting for instructions from the attacker as row 3 indicates. Rows 5, 7 and 9 depict the echoes of the three characters in command *dir* entered by the attacker in rows 4, 6 and 8; the character-by-character transmission and echo-back mechanism are typical of the *Telnet* protocol and together provide a reliable way to identify such traffic. We combine together all packets that convey the client's command "*run c:/windows/notepad.exe*" and show them in the row marked 13-41; the 42-70 line depicts the respective echo activity. Clearly, the attacker mistakenly types *ran* for the desired *run* command and later uses "backspace" (0x08) to correct the mistake. By making available editing functionalities, Drat provides a true interactive environment. However, it poses a challenge for security devices that detect

RATs based on patterns as the latter can be readily evaded by inserting an arbitrary number of edition keys such as *backspace*, *delete*, and/or empty spaces. An anti-Trojan system has to simulate shell functionality and act as a command interpreter, should it successfully identify *Telnet*-based *RATs*.

<table>
<tr><th><i>#</i></th><th><i>dir</i></th><th><i>payload</i></th><th><i>description</i></th></tr>
<tr><td colspan="4">protocol: TCP; attacker (denoted as A): 192.168.5.143; victim (denoted as V): 192.168.5.141</td></tr>
<tr><td>1</td><td>A:1085⟶V:48</td><td>|0D 0A|</td><td>attacker just enter “RETURN”.</td></tr>
<tr><td>2</td><td>V:48⟶A:1085</td><td>Welcome to DaRat’z Telnet Rat.</td><td>server responses with banner</td></tr>
<tr><td>3</td><td>V:48⟶A:1085</td><td>DRat Version 1.3.0 ... DRat : D:/temp&gt;</td><td>server displays prompt</td></tr>
<tr><td>4</td><td>A:1085⟶V:48</td><td>d</td><td>attacker enters cmd (one char at a time)</td></tr>
<tr><td>5</td><td>V:48⟶A:1085</td><td>d</td><td>server echoes back input</td></tr>
<tr><td>6</td><td>A:1085⟶V:48</td><td>i</td><td>attacker enters next char</td></tr>
<tr><td>7</td><td>V:48⟶A:1085</td><td>i</td><td>server echoes back</td></tr>
<tr><td>8</td><td>A:1085⟶V:48</td><td>r</td><td>attacker enters next char</td></tr>
<tr><td>9</td><td>V:48⟶A:1085</td><td>r</td><td>server echoes back</td></tr>
<tr><td>10</td><td>A:1085⟶V:48</td><td>|0D 0A|</td><td>with “RETURN”, cmd “dir” is formed</td></tr>
<tr><td>11</td><td>V:48⟶A:1085</td><td>|0D 0A|</td><td>server echoes back</td></tr>
<tr><td>12</td><td>V:48⟶A:1085</td><td>DRat The Worlds Ultimate Virtual SPY. doc = 46592 DRAT.exe ...</td><td>cmd executed, return file list</td></tr>
<tr><td>13-41</td><td>A:1085⟶V:48</td><td>ra|08|un c:/windows/notepad.exe|0D 0A|</td><td>attacker runs an app; “08” is backspace</td></tr>
<tr><td>42-70</td><td>V:48⟶A:1085</td><td>ra|08|un c:/windows/notepad.exe|0D 0A|</td><td>server echoes back all inputs</td></tr>
<tr><td>71</td><td>V:48⟶A:1085</td><td>ShellExecute (2 Being a Error) Returned</td><td>app is executed; server replies</td></tr>
</table>

Table VI. Drat(v.1.0)-generated traffic

3.5. Presence of Multiple Evasion Techniques in *RATs*

A straightforward approach for *RATs* to evade detection is to continually change both message structures and banner information as such artifacts are exploited as signatures by firewalls, AVs, and IDSs/IPSs. In this regard, DeepThroat(v1.0) flushes the name of the victim machine (e.g., *SHEEP*) at the very end of its banner “*–Ahhhhhhhhhh My Mouth Is Open SHEEP*” while in other versions, the name of the victim machine appears first as demonstrated by the banner ‘*SHEEP - Ahhhhh My Mouth Is Open (v2)*” of DeepThroat(v2.0). Telltale patterns used in DeepThroat change slightly in different versions as well. For instance the pattern *Ahhhhhhhhhh* in v.1.0 has been shortened to *Ahhhhh* in versions v.2.0 and v.3.0 and ultimately became *Ahhhh* in v.3.1. Such banner adaptations make it more difficult for security systems to detect with a single fixed signature the traffic generated by various versions of DeepThroat operating simultaneously.

Sending decoy messages is another popular evasion technique. By doing so, Trojans such as Doly attempt to induce security systems to generate large numbers of false positives [43], forcing the security officers to spend considerable periods of time examining logs. The latter produces a good chance for the intruders to go undetected. A Doly client attempts to establish a *TCP* connection with a victim by trying ports 3456, 4567, 5678, 6789 [¶], 7890, 9182, 8374, 2345, 7654, and 27559 in sequence four times regardless if a *RAT* is present. This activity resembles to port scanning which is typically graded as a low-severity surveillance activity by AVs/IDSs/IPSs and is frequently ignored by security officers. Although *RATs* often operate on their default ports, they can be configured to use ports in either the privileged or non-privileged range. While experimenting with real traffic, we observed that *RATs* mostly employ port hopping techniques and servers use arbitrary ports selected on the fly readily defeating port-based detection approaches. Hence, a content-based approach would be by far more fruitful in detecting Trojans using decoy messages, dynamic port hopping, and other evasion techniques.

[¶] Doly’s server default port.

4. A Framework for Apprehending *RATs*

Conventional security systems use add-on modules and/or specially-crafted signatures to identify malicious *RAT* activities and are unaware of *RATs* unique characteristics. To address these shortcomings, we propose an extensible framework named *RAT Catcher* that employs network-based detection methods, operates *in-line*, and manipulates *RAT*-traffic in real-time. We base our design on the following constraints: i) *RAT*-servers are implanted on victim machines through the channels discussed in Section 3.2. Through its real-time operation, our framework puts emphasis on detecting *RAT* communications as opposed to conventional AVs and host-based security systems that mainly focus on the Trojan installation process [14]; and ii) features of sought *RATs* including format of messages, handshake procedures, and functionalities are available typically via reverse engineering, behavior analysis, and data mining techniques [53, 2]. By tracking the progress of all established connections initiated by either normal applications or Trojans, our framework conducts data correlation between different sessions or traffic streams, performs stateful inspection, and identifies abnormal and/or deviating behavior in real-time. Our *RAT Catcher* stores packets in every data stream, re-assembles them together and the resulting aggregations are subjected to protocol dissection according to standard *TCP/IP* specifications as well as the syntax and semantics of individual Trojans. In this way, our framework performs layer-7 or application-level inspection and can effectively combat evasive mechanisms used by *RATs*. As soon as a session is verified as Trojan, our framework can immediately take corrective steps by logging, blocking, terminating the connection or simply taking over the session.

4.1. Design Rationale and Architecture for the *RAT* Catcher

To remotely control a computing system, an attacker should first set up either a *TCP* or *UDP* channel with the *RAT* server implanted on the victim machine. A *TCP* session is defined by its distinct connection and disconnection processes [15]. The former is a three-way handshake procedure where the initiator (or client) starts a connection with a *TCP SYN* packet and the recipient (or server) replies with a *TCP SYN-ACK* packet which in turn incurs an *ACK* packet from the initiator. The disconnection procedure typically involves a four-message exchange with each side dispatching a *TCP FIN* packet and corresponding acknowledgments from the two ends. Evidently, a specific *TCP* connection can be uniquely identified by the tuple <*client-IP, client-port, server-IP, server-port*>. Similarly, a specific *UDP* session is also identified by a similar four-element tuple with its termination often designated with a time-out mechanism. Any change in the above tuple implies a new *TCP/UDP* session. Overall, a connection can be represented with the extended tuple <*client-IP, client-port, server-IP, server-port, protocol*> where *protocol* can be either *TCP* or *UDP*. Within each session, two data streams exist, one from originator (or client) to recipient (or server) and the second in the opposite direction. Each data stream within a session can be identified with a four element tuple $< IP_s, PORT_s, IP_d, PORT_d >$, where IP_s and $PORT_s$ are the IP address and network port of the source and IP_d and $PORT_d$ are their destination counterparts.

Messages exchanged between a Trojan server and client are generated in accordance to the *RAT*'s own syntax rules and semantics and are shipped following the constraints of *TCP/UDP* transport protocols. The latter may not respect the Trojan's message boundaries, therefore, inconsistencies between Trojan message borders and transport packet demarcations are unavoidable. The transport layer may also deliver packets in an arbitrary order and the original data stream can be only recovered via reassembly by its recipient. For instance, the *TCP* packet of row 5 in Table IV consists of three NetBus Pro messages. Similarly, in the SubSeven-generated traffic of Table VII, multiple Trojan commands are contained in a single *TCP* packet as row 12 shows. In this, commands *IN7* and *CL7* open the Webcam of the victim, *IN2* and *CL2* attempt to open the screen preview and *PSS* obtains cached passwords. On the other hand, command *dir<CF><LF>* entered by the Drat attacker in Table VI is split into four *TCP* packets; the payload of each packet contains only one or two characters.

#	*timestamp*	*dir*	*payload*	*description*
protocol: TCP; attacker (denoted as A): 192.168.5.143; victim (denoted as V): 192.168.5.141				
1	0.002077	V:55555⟶A:3689	connected. 21:17 ..., ver: Legends 2.1	banner from server, access time and version
2	20.819971	A:3689⟶V:55555	URLhttp://www.fortinet.com/	instruct victim to goto given “URL”
3	20.820263	A:3388⟶F:80	(SYN)	connect to server F (i.e., www.fortinet.com)
4	21.675816	V:55555⟶A:3689	web browser has been opened.	server successfully launches browser
5	72.771562	A:3689⟶V:55555	FFNF05*.exeC:/	try to find files “*.exe” under “C:/”
6	72.912758	V:55555⟶A:3689	C:/EXPLORER.EXE C:/putty.exe ...	a list of specified files is returned
7	108.402655	A:3689⟶V:55555	FTPenable!@@@21:::1$$$C:/	enable FTP server by cmd “FTP”
8	118.066511	A:3689⟶V:55555	FTPdisable	try to disable FTP server
9	118.068258	V:55555⟶A:3689	FTP server disabled	remote FTP server is disabled
10	129.070129	A:3689⟶V:55555	IRG	access registry editor
11	129.131769	V:55555⟶A:3689	Console Control Panel Environment ...	menu of registry editor is returned
12	139.577822	A:3689⟶V:55555	IN7CL7IN2CL2PSS	open Webcam (IN7CL7), get passwords (PSS)
13	139.579805	V:55555⟶A:3689	PSS cached passwords: [www.fort...]	return network addr, login name, password
14	208.272307	A:3689⟶V:55555	CLG	close registry editor

Table VII. SubSeven (v.2.1)-generated traffic

There are often multiple concurrent sessions between a *RAT* client and its server coexisting with sessions created by other applications. For instance in the SubSeven traffic of Table VII, two sessions coexist after the packet in row 3. The first is identified by tuple *<A,3689,V,55555,TCP>* where *A* stands for the attacker located at *IP* address 192.168.5.143 and *V* is the victim with *IP* address 192.168.5.141. The second session is identified as *<A, 3388, F, 80, TCP>* where *F* is the *IP* address of the Web server *www.fortinet.com*. In a similar fashion, the DeepThroat traffic of Table VIII creates two sessions, the control session identified as *<A,60000,V,2140,UDP>* and the data session commencing at row 4 and identified as *<A,60000,V,3150,UDP>*. Most anti-Trojan systems detect *RAT* control-sessions only and they are completely blind to respective data-sessions simply because the latter lack unique telltale patterns.

#	*dir*	*payload*	*description*
protocol: UDP; attacker (denoted as A): 192.168.5.143; victim (denoted as V): 192.168.5.141			
1	A:60000⟶V:2140	00	a “ping” message
2	V:2140⟶A:60000	host - Ahhhh My Mouth Is Open (v3.1)	a “pong” message
3	A:60000⟶V:2140	39	a request to “create directory”
4	A:60000⟶V:3150	c:/temp/host	parameter to “create directory” command
5	V:2140⟶A:60000	Directory Created	reply to command “create directory” (39)
6	A:60000⟶V:2140	35	command “freeze mouse”
7	V:2140⟶A:60000	Mouse frozen	reply to command “freeze mouse” (i.e., 35)
8	A:60000⟶V:2140	12	command “send host info”
9	A:60000⟶V:3150	http://www.fortinet.com	parameter to “send host info” command
10	V:2140⟶A:60000	Host Sent To URL	reply to command “send host info” (12)

Table VIII. DeepThroat(v.3.1)-generated traffic

For all established connections originated by either normal applications and/or suspected-Trojans, *RAT Catcher* maintains session records, including information of connection initiator and recipient, messages exchanged, and application type of the session. Such session information is not only maintained during the lifetime of a session but also remains accessible beyond its lifetime as part of a history repository. The capability of tracking the state of each active session facilitates stateful inspection, intra-session data fusion, and inter-session correlation for Trojan-generated traffic. With the help of stored information, we can also determine the progress of a session. For instance, information on connection status can indicate whether a *TCP* session is in the three-way handshake procedure, has established connection, or is at its termination phase. By correlating the data of streams within the same session, *RAT Catcher* can rapidly determine whether an

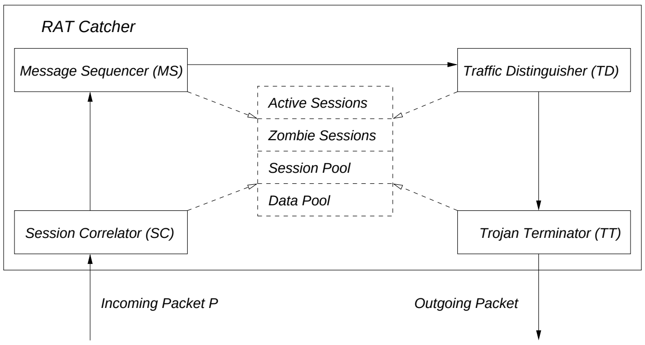

Figure 3. *RAT Catcher* architecture

attacker's operations are successful. In addition, by correlating the information from prior sessions with an ongoing connection, a *RAT* data channel could be readily associated with its control channel, thereby prompting the same preventive/punitive action by the *RAT Catcher*. Furthermore, we use the information of defunct sessions stored in the history repository for carrying out session correlation computations that help determine the traffic type of currently active sessions.

The storage of all transport packets facilitates their remapping to Trojan messages by the packet re-assembly procedure that takes place with the help of the syntax and semantics of known Trojan protocols. The objective of this aggregation also known as *Message Sequencing*, is to conceal the packet demarcation imposed by the transport layer and restore the boundaries of Trojan messages. Without sequencing, Trojan sessions may go undetected if their constituent messages happen to span multiple transport packets. Given the large number of existing *RATs* and the variety of protocols used, it becomes challenging to identify a *RAT* session effectively and efficiently. To classify data streams, the *RAT Catcher* resorts to multiple techniques that include traffic correlation, application protocol analysis, and stateful inspections in addition to fine-tuned signatures. Once a session is confirmed as Trojan, our framework may immediately block, terminate or take over the session besides log generation. Figure 3 shows the architecture of our framework that entails the following modules: *Session Correlator (SC), Message Sequencer (MS), Traffic Distinguisher (TD)*, and *Trojan Terminator (TT)*.

Once a packet P arrives at the *RAT Catcher*, the *Session Correlator (SC)* module determines whether there exists a session S in which P belongs to; if there is none, a new session S is created for P. Based on information about S or correlation with other active or defunct sessions, it may be possible to immediately determine whether P is part of a *RAT* such as Back Orifice, SubSeven, NetBus, or DeepThroat. P is then handed over to the *Message Sequencer (MS)* along with its session information S, where P is re-assembled with other existing packets of the same stream to form a sequence of application messages. In turn, this message sequence is transferred to the *Traffic Distinguisher (TD)* to determine the specific application type of the session. Finally, P arrives at the *Trojan Terminator (TT)* where information on S is appropriately updated, P is stored in S to help forthcoming re-assembly efforts and a corrective action may be taken if *RAT* traffic has been found.

4.2. *Session Correlator (SC)*

To maintain information for each session, we use the *session* data structure whose key fields are shown in first part of Table IX. Each connection is assigned a *session* structure and is uniquely identified by its first five fields: source and destination *IP* addresses (*SIP*, *DIP*), source and destination port numbers (*SPORT, DPORT*) and protocol (*PROTO*). Field *TYPE* indicates the application type of the session such as DeepThroat, NetBus, WanRemote etc.; this field can assume the value *bypass*, should the application type cannot be determined after a certain amount of traffic in the session has been inspected or the session is generated by a normal application. The field *CONFIRM* indicates whether the value in *TYPE* has been derived from correlating streams in both directions of the same session, obtained by association with other sessions (active or zombie), or simply drawn based on different messages in a uni-directional traffic. Clearly, should a session application type be drawn using multiple criteria –data streams, sessions, and correlations– the classification accuracy is improved. The two data streams in a session, the one from client to server and its reverse counterpart are stored in fields *CLIENT* and *SERVER* respectively; these pointers to *stream* structures are discussed in Section 4.3.

field name	*size (bytes)*	*description*
Key fields of the *session* data structure		
SIP	4	*IP* address of the host at one end of the connection
DIP	4	*IP* address of the host at the other end of the connection
SPORT	4	port number of the host with *IP* address *SIP*; may assume special value of *unknown*
DPORT	4	port number of the host with *IP* address *DIP*; may assume special value of *unknown*
PROTO	1	protocol utilized by the session (*TCP* or *UDP*)
TYPE	4	identify traffic type, such as DeepThroat, NetBus, SubSeven; can be *unknown*
CONFIRM	4	*TYPE* is drawn from uni-or bi-directional streams, intra- or inter-session correlations
SERVER	4	pointer to server *stream* data structure
CLIENT	4	pointer to client *stream* data structure
Key fields of the *stream* data structure		
state	4	state of the stream (*TCP* only) such as *CLOSED, LISTEN, SYN_SENT, SYN_RCVD, ESTABLISHED*
next-seq	4	next sequence number expected (*TCP* only), computed based on information of sender
ISN	4	initial sequence number of the current stream (*TCP* only)
data	4	pointer to the root of *interval-tree* storing all packets of the stream
total-size	4	total size of data transferred in the stream so far (*UDP* only)
data-size	4	number of bytes stored in “data” buffer (*UDP* only)

Table IX. Key fields in data structures *session* and *stream*

Figure 4 shows an hierarchical structure termed *active sessions table* that we use to organize session pertinent information; it provides efficient session insertions, retrievals, deletions, and facilitates intra-session and inter-session correlations. We first use a hash-table to group sessions with hash function H(*SIP,DIP,PROTO*)=((h>>16) *xor* (h>> 8)) *mod h_size* where auxiliary function h is defined as h=(*SIP xor DIP xor PROTO*), “>>” is the *right shift* operation, and *h_size* is the size of the hash table. In computing the hash value, *SIP, DIP*, and *PROTO* are treated as integers with protocols *TCP* and *UDP* assuming values *0x06* and *0x11* respectively. Although simple, function H has exhibited near-uniform distribution in our experiments. Next, each hash-table entry points to a *splay tree* T that complies with the binary search tree property and attains an amortized time by moving a tree node closer to the root every time it is accessed [48]. Clearly, frequently accessed elements are more likely to be closer to the root.

A splay tree T anchored off each entry of the hash table helps organize all sessions that present the same hash value. In our framework, each node in T represents all connections established between a source/destination pair described by fields *SIP, DIP* and *PROTO* of the structure *session* (of Table IX). More specifically, every node of T is associated with a *port mapper* consisting of two tables; one organizes the ports (*SPORT*) used by *SIP* and the other stores ports (*DPORT*) used by *DIP*. If a port number is active and occupied by a session, the appropriate record for the session is stored in the corresponding slot of the *port mapper*. Multiple connections sharing the same port number are organized with a linked-list under the slot indicated by the port number. Finally, all session specific data are maintained in the session pool of Figure 4 and are organized as linked-lists as well.

To facilitate session retrieval, we designed function *session-find(*P*, wildcard)* whose goal is to locate the session S that packet P belongs to. This function initially searches the *active session table* with the help of data in tuple <*SIP, SPORT, DIP, DPORT, PROTO*> available through P. If the outcome is an existing session S, P is client-initiated and S is marked by *SC* as having forward direction “*DIR: forward*”; otherwise, *SC* looks up the

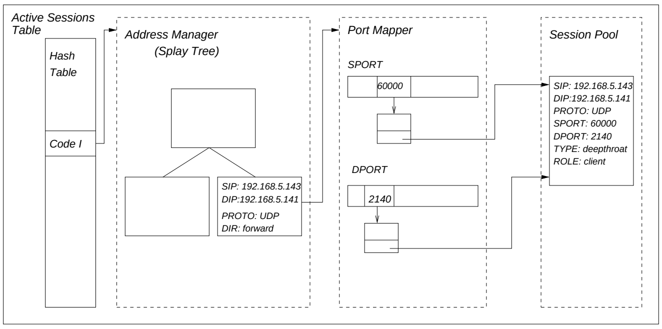

Figure 4. Session information before data channel is established for traffic in Table VIII

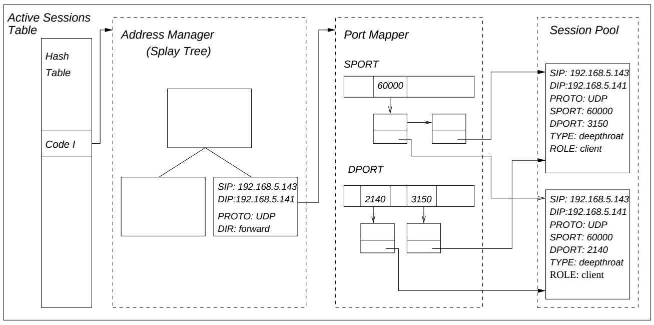

Figure 5. Session information after data channel is established for traffic in Table VIII

active session table again with a new tuple <*DIP, DPORT, SIP, SPORT, PROTO*> formed by exchanging the roles of source and destination ends. A non-empty result S in this second attempt indicates that P is part of a server-originated stream tagged with direction "*DIR: reverse*". The *wildcard* argument of *session-find()* specifies whether wildcard matching is conducted in the session lookup. By assuming value "*source-port*" or "*destination-port*" for parameter *wildcard*, *session-find()* considers any active source or destination port to be a match. Should *wildcard* be designated as "*none*", an exact port match is performed.

Algorithm 1 outlines the three-step operation *SC* carries out. First, the function call *session-find(*P*,none)* is used to extract the session S corresponding to packet P by traversing the active session table; a non-null S indicates that P belongs to an existing session and *SC* simply exits by returning S. Next, *SC* verifies whether P initiates a new session that acts as a data channel associated with an existent *RAT* control channel. Finally, the application type of the newly created session S for packet P is set by correlating S with active/defunct session tables.

We use the DeepThroat traffic of Table VIII as an example to describe the procedure followed by *SC*. The traffic segment indicates that the attacker establishes two *UDP* sessions with *2140* and *3150* as their corresponding destination ports; the former acts as the command channel and the latter as the data channel. By routing the traffic of Table VIII to our *RAT Catcher*, *SC* creates session <*A,60000,V,2140,UDP*> immediately after packet at row 1 is encountered. When *SC* deals with packet P of row 2, the invocation of *session-find(*P*, none)* yields session <*A,60000,V,2140,UDP*> whose mark "*DIR: reverse*" indicates P to be a server-originating packet. We assume that after processing the first two packets, the *SC* records information on session <*A,60000,V,2140,UDP*>,

Algorithm 1 Procedure followed by *Session Correlator (SC)*

```
1:  $P$ is a newly arriving packet; SIP and DIP are the source/destination IP addresses of $P$; sport and dport are
    the source/destination ports of $P$; PROTO is the protocol of $P$;
2:  find $S$ associated with $P$ in active session table by calling function session-find($P$, none)
3:  if ($S$ is not null) then
4:      return $S$ and exit;
5:  end if
6:  find session $T$ in the active sessions table with function call session-find($P$, destination-port);
7:  if ($T$ is not null) then
8:      if dport of session $T$ is any, replace it with that of $P$, return $T$ and exit;  otherwise, create a new session $S$
        for $P$ with tuple <SIP,sport,DIP,dport,PROTO>; fields TYPE and CONFIRM of $S$ are set to the same as
        $T$; return $S$ and exit;
9:  end if
10: create a new session $S$ for $P$ with tuple <SIP,sport,DIP,dport,prot>;
11: find session $T$ in the defunct session table with tuple <DIP,dport,PROTO>;
12: if ($T$ is not null) then
13:     set TYPE of $S$ to that of $T$ and return $S$;
14: end if
```

constructs the session table of Figure 4 and tentatively marks the session as DeepThroat. When the packet in row 4 is encountered, the *RAT Catcher* establishes that no session <*A,60000,V,3150,UDP*> exists as evidenced by Figure 4. Before creating a new session <*A,60000,V,3150,UDP*>, *SC* looks up in the *active sessions table* for any session matching <*A,60000,V,any,UDP*> with the help of the call *session-find(P, destination-port)*. By checking the type of the session in the lookup result, <*A,60000,V,2140,UDP*>, the *RAT Catcher* establishes that a DeepThroat control channel indeed exists and therefore marks the newly created session as DeepThroat (Figure 5). Algorithm 1 carries out this session correlation between *RAT* control and data channels in lines 6 to 9.

Information about terminated or *zombie* sessions, especially those recently torn down, may be helpful to determine the application type of currently active sessions. For example, if a service provided on a specific network port of a host identified with the tuple <*DIP,DPORT,PROTO*> has been determined as a Trojan server by some previous sessions, its application type is not expected to change abruptly. Based on the above observation, our *RAT Catcher* keeps information about *zombie* sessions of Trojan servers accessible for a period of time –configurable but set to 5 minutes by default– and such zombie sessions are organized in the defunct session table with key of <*DIP,DPORT,PROTO*> and value of *TYPE*. Any time a new session <*SIP,SPORT,DIP,DPORT,PROTO*>, is generated by the *SC*, the defunct session table is consulted with query <*DIP,DPORT,PROTO*>, and the application type of the returned session, if any, is assigned to the newly created session. For instance, by the time *RAT Catcher* processes the connection beginning at row 3 of Table V, session <*A,1071,V,80,TCP*> has become zombie and therefore, it is stored in the defunct session table. By simply searching for <*V,80,TCP*> in the defunct session table, the application type of the session established by row 3 of Table V can be quickly determined as *WanRemote*. Such a correlation between ongoing connections and already-known *zombies* can establish relevant temporal session associations and rapidly identify the application types of ongoing sessions.

4.3. *Message Sequencer (MS)*

We design the *stream* structure –key elements of which appear in the lower part of Table IX– so that packet re-assembly and state tracking for a data stream can be easily carried out. Different types of information are stored for connection-oriented and connectionless channels. For *TCP* streams, field *state* tracks the connection state of its originator and can be *SYN-SENT*, *SYN-RCVD*, *ESTABLISHED*, or *CLOSE*; fields *ISN* and *next-seq* maintain the initial sequence number and the next expected sequence number, respectively. Last, *data* is a pointer to an *interval-tree* [17] used to organize all encountered stream packets according to a search key [n_1,n_2] where n_1, n_2 represent start and end sequence numbers; the value of every interval-tree node is a single packet of the stream in question. In a *UDP* stream, the *data* points to a buffer that stores data received but unprocessed for the stream in question thus far, whose size is indicated by field *data-size*. The field *total-size* indicates the number of bytes transmitted helping track the volume of data encountered by the session. We define function call *stream-find(S, P)* to retrieve the stream I corresponding to packet P within session S.

The *Message Sequencer* uses an interval-tree I, implemented as a red-black tree to store stream-pertinent data. The key [V_s, V_e] of a specific node V in I represents the start and end sequence numbers (*SSN* and *ESN*) of the corresponding packet P; V_s can be directly obtained from field *sequence number* of P's *TCP* header while V_e can be derived with the help of the fields *total length*, *IP header length* as well as *TCP header size* of the packet. For *UDP* streams, we simply assign the current value in *total-size* of the stream as the start sequence number (*SSN*) of the incoming packet and its *ESN* can be derived from *SSN* and the packet size. For any two packets P and Q of the same stream, we define different relationships based on their sequence intervals. By denoting P_s and P_e as the start and end sequence numbers for P as well as Q_s and Q_e for Q respectively, we can determine that: **a)** Q is a duplicate of P if P_s=Q_s and P_e=Q_e; **b)** P and Q overlap if P_s<Q_s and Q_s<P_e < Q_e or Q_s<P_s and P_s< Q_e<P_e; **c)** P contains Q if Q_s>P_s and Q_e<P_e or Q contains P if P_s>Q_s and P_e<Q_e; and **d)** P precedes or follows Q if P_e<Q_s or P_s>Q_e, respectively. With the help of these interval relations, we design functions to manipulate the interval tree: *interval-insert(I, P)* inserts a node representing packet P into I, *interval-delete(I, P)* removes the node of packet P from I, and *interval-retrieve(I, P)* returns a pointer to a node Q of the tree I provided that a duplicate, overlap, or containment relationship between Q and P can be established; otherwise, the outcome is a *NULL*. Evidently, these operations maintain complexity $O(\log(n))$ where n is the number of nodes in I. In addition, the function *packet-build(I, SSN, ESN)* creates and returns a new packet Q with interval indicated by [*SSN*, *ESN*]. Finally, function *interval-traversal(I)* performs an in-order tree walk of I and lists all intervals (i.e., packets) in sorted order by their *SSN*; this function is useful for logging packets into permanent storage.

The stream re-assembly process used by *MS* is presented in Algorithm 2 and works as follows: for an arriving packet P, *MS* obtains the session S of P and function *stream-find(S, P)* is invoked to fetch its stream I. Next, *MS* checks the freshness of P by calling function *interval-retrieve(I, P)*, which returns a pointer to node Q of tree I. A *NULL* Q implies that P is a new packet and can be inserted into I with the help of function *interval-insert(I, P)*. If P and Q are duplicates, their payloads are compared to ensure that

Algorithm 2 The procedure followed by *Message Sequencer (MS)*

```
$P \leftarrow$ incoming packet; $S \leftarrow$ session of $P$ returned by Session Correlator (SC); $I \leftarrow$ stream-find($S$, $P$);
if (TYPE and CONFIRM of $S$ are set) then
    $P$ is part of a RAT or normal session; hand it over to the protocol analyzer indicated by TYPE or Traffic
    Terminator (TT); exit;
end if
$Q \leftarrow$ interval-retrieve($I$, $P$);
if ($Q$ is empty) then
    $P$ is a brand new packet and function interval-insert($I$, $P$) is used to add $P$ into $I$;
else
    check whether the overlapping parts of $P$ and any packet in $Q$ have the identical contents; if not, generate
    alerts and exit;
end if
$t_s \leftarrow$ initial sequence number (ISN) of $I$; $t_e \leftarrow$ (sequence number of $Q$)+(payload size of $Q$), where $Q$ is the
packet with largest sequence number in $I$
if (($t_e$ - $t_s$) is larger than $MAX\_SIZE$ (default $MAX\_SIZE$=5 KB)) then
    TYPE and CONFIRM of $S$ are set to be bypass; and exit;
end if
$O \leftarrow$ packet-build($I$, $t_s$, $t_e$) and is handed over to Traffic Distinguisher (TD)
```

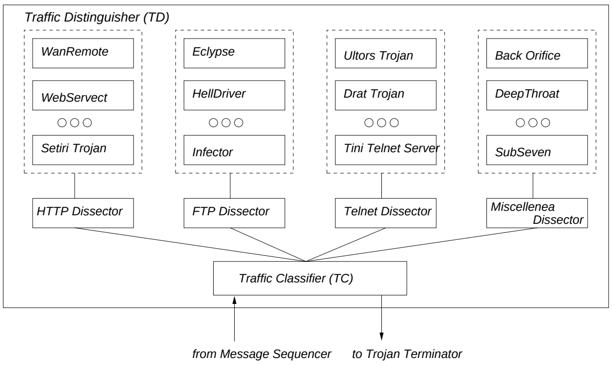

Figure 6. *Traffic Distinguisher (TD)* components

they have identical content before P is inserted into I with a different timestamp. In a similar manner, for cases where P and Q overlap and either P contains Q or Q contains P, their contents on the common sequence interval are compared; if they share the same content for the overlapping part, P is inserted into I since P is a normal overlap packet or retransmission of Q; otherwise, P is suspicious as it may have been crafted with evasive tools [‖] and administrator-specified counter measures such as dropping can be applied. Finally, function *packet-build(I, SSN, ESN)* is called, with SSN the initial sequence number of the stream I and ESN the largest sequence number thus far in I, to re-assemble the data stream and the resulting aggregation, termed *super-packet* O, is handed to the *Traffic Distinguisher (TD)* module.

4.4. *Traffic Distinguisher (TD)*

As every Trojan follows its own protocol, it would be unrealistic to use a monolithic mechanism for detecting all possible *RATs*. The *Traffic Distinguisher (TD)* module addresses this challenge by using a multi-phase traffic classification scheme. First, all incoming traffic is categorized by the *Traffic Classifier (TC)* into four general types of streams: *HTTP*, *FTP*, *Telnet* and *Miscellanea*. Subsequently, each type is handled by its own traffic *Dissector*. In the *Miscellanea Dissector*, we employ specific Trojan analyzers to detect streams that potentially belong to *RATs* using proprietary protocols as such Trojans hardly manifest any commonalities. Figure 6 shows the *TD* components and their organization. *Traffic Classifier (TC)* component mainly employs heuristic rules to classify traffic. For every incoming packet P, along with its session S, stream I, and the *super-packet* O constructed by module *Message Sequencer (MS)*, *TC* determines the traffic type of P with the help of the following rules:

1. packet P is client-initiated: The *super-packet* O is matched against pattern "*method URL HTTP/[0.9|1.0|1.1]*", where *method* is any legitimate *HTTP*-method such as *GET*, *POST*, and *PUT* [25]; a positive result causes session S to be marked as *HTTP*. Similarly, S is claimed as *FTP*-compliant traffic if O follows pattern "*command parameter|0D 0A|*". S is marked as *Telnet*-compliant traffic if application payload of P is less than 3 bytes and O follows pattern "*command parameter|0D 0A|*". Otherwise, S is marked as *Miscellanea*.

[‖] such as fragroute

2. packet P is server-initiated: If O follows pattern "*HTTP/[0.9*$|$*1.0*$|$*1.1] 3-digits status-text*", S is marked as *HTTP*-compliant traffic. S is deemed as *FTP* traffic if O conforms with syntax "*text-string*$|$*0D 0A*$|$". In order to identify *Telnet*-compliant traffic, O is first split into lines with demarcation symbols $|$*0D 0A*$|$, the first line is then rewritten by simulating the effects of function-keys "*backspace*" and "*delete*", and the outcome is compared against the client-dispatched command. If all above checks fail, S is marked as *Miscellanea*.

Once *TC* determines the type of P, it invokes the corresponding *Dissector*.

The *Dissectors* analyze the header and body of exchanged messages in both directions of a session and correlate messages within the same session to further improve classification accuracy. In particular, the *HTTP Dissector* inspects the header and body of *HTTP* messages, attempting to identify *RATs* that follow the *HTTP* protocol such as WANRemote. The *FTP Dissector* identifies *FTP*-based Trojans including Eclypse, HellDriver and Infector while the *Telnet Dissector* detects the Drat, Ultors and Tini Telnet Server Trojans that follow *Telnet* protocol specifications. Similarly, *RATs* including SubSeven and NetBus that use their own protocol specifications are handled by *Miscellanea Dissector*; the latter is mostly based on analyzers that use *RAT* specifications often obtained through reverse engineering.

By and large, *RAT* streams that are transported via the same protocol appear to be very similar except in their use of ports, banners, and server-replies received. For instance, all *FTP*-based Trojans such as Eclypse (Table II) and HellDriver use *TCP*-ports for their control and data channels and their server use banners that differ only in content. For example, the Eclypse banner is *"220 EclYpse 's FTP Server is happy to see u !"* and that of HellDriver's is *"220 ICS FTP Server ready"*. Although *FTP*-compliant *RATs*support different command sets, they overall follow the *FTP* specification. Based on these observations, our *RAT Catcher* detects *FTP*-compliant *RAT* types by mainly using their server-banners and client-command sets. As soon as the banner and command set of a newly-established *FTP*-based *RAT* become available, *RAT Catcher* can successfully identify the Trojan in question through proper augmentation of the *FTP Dissector*. This also applied to *HTTP* and *Telnet*-compliant *RATs* as well. Algorithm 3 shows the overall operation of the *FTP Dissector*; the dissectors for *Telnet* and *HTTP* are laid out similarly. Algorithm 3 treats

Algorithm 3 Procedure for *FTP Dissector*

```
$P$ is the newly arrival packet, $S$ and $I$ are session and data stream that $P$ belongs to $O$ is the re-assembled "super-packet"
if ($I$ is from server to client of session $S$) then
    $O$ is split into multiple lines demarcated by |0D 0A|, $banner$ is assigned the first line
    for (each banner $telltale$ of FTP-based RATsidentified by RAT Catcher) do
        match banner against telltale and TYPE of $S$ is set to the RATtype corresponding to telltale if a match is found
    end for
else
    $O$ is split into multiple lines demarcated by |0D 0A|, command is assigned the first token of the first line separated by empty space
    if ($command$ is PORT) then
        calculate port number pt based on the parameter of the PORT command – refer to Table II for calculation formula;
        create a new session with tuple <DIP,any,SIP,pt,TCP>, where $SIP$ and $DIP$ are the source/destination IP addresses of $P$
    end if
    for (each command $instruction$ used by FTP-based RATs) do
        match $command$ against $instruction$ and CONFIRM of $S$ is set if a match is found
    end for
end if
$P$ is handed over to Trojan Terminator (TT)
```

FTP command *PORT* in a special way as this command specifies the port used for the data channel. A pseudo-session <*DIP,any,SIP,pt,TCP*> is created where SIP, DIP are the source/destination IP addresses of the packet in question, and *pt* is the port number specified in command *PORT*. As the network port that would be used by Trojan server data channel in the near future is yet unknown, a placeholder *any* is used instead.

The function of *Miscellanea Dissector* is inherently different from that of its counterparts in Figure 6 as it attempts to detect proprietary *RAT* communication protocols that hardly demonstrate any commonalities. This dissector essentially acts as a scheduler for all registered analyzers whose objective is to dissect exclusively proprietary protocols. For each incoming packet P, *Miscellanea Dissector* invokes in sequence all *RAT*-analyzers that have been implemented and incorporated in our framework. The process continues until the application type of P is either determined or all *RAT*-analyzers have been used with no outcome.

Algorithm 4 Protocol Analyzer for NetBus Pro

```
$P$ is the newly arrival packet, $S$ and $I$ are session and data stream that $P$ belongs to $O$ is the re-assembled super-packet
verify that $O$ is at least 10 bytes in size and starts with string "BN", otherwise, exit from the procedure
$len$ = $O[2, 3]$, that is, the second and third bytes of $O$; $version$ = $O[4, 5]$; $code$ = $O[8,9]$
check conditions ((size of $O >= len$) and ($version$ = 2) and ($code < 200$)) satisfied; otherwise, exit from the procedure
if ($I$ is from server to client of session $S$) then
  TYPE of $S$ is set to NetBus Pro
else
  CONFIRM of $S$ is set and a NetBus Pro session is detected
end if
$P$ is handed over to Trojan Terminator (TT)
```

Analyzers within the *Miscellanea Dissector* share similar working mechanisms and Algorithm 4 shows the skeleton of the NetBus Pro analyzer. In brief, Algorithm 4 first finds boundaries of *RAT*-messages from the re-assembled *super-packet O* provided by module *MS*. In particular, Algorithm 4 inspects whether *super-packet* O satisfies the minimum message size of 10 Bytes and starts with telltale "*BN*" as necessitated by NetBus Pro specification. It then examines whether the restored messages yield NetBus Pro traffic. This is done by extracting well-defined fields including *message-size*, *version-number*, and *command-code* and tentatively marking the flow as NetBus Pro. Finally, Algorithm 4 uses the first server-message to confirm the initial marking by the first client message and set the field *CONFIRM*.

4.5. *Trojan Terminator (TT)*

The *Trojan Terminator (TT)* module of the *RAT Catcher* allows for counter measures to be taken for different types of detected *RATs*. The *TT* module examines the application type of a received packet P in conjunction with the status of its session S and stream I and take administrator-specified actions for the various types of traffic. If fields *TYPE* and *CONFIRM* of session S are not set, *TT* simply forwards P to the next hop enroute to its destination. Otherwise, *TT* uses a number of options including alert generation, logging of P as well as its data stream I and session S, blocking of subsequent messages from the same session, and/or take-over by acting as a *RAT* server to the initiator of the session. Information shown in Table IX for session S is also updated based on P to help subsequent re-assembly operations and improve the accuracy of *RAT Catcher*.

In addition, *TT* can become more proactive by disabling identified *RATs*. In particular, it may remove all Trojan-related components from the victim's file system. For instance, NetSphere has a unique feature that allows for the purging of its server through the client-command *KillServer*. The command <*KillServer*> issued by the NetSphere client forces the server to disconnect itself from the network; in addition, the server un-installs itself from the victim system by removing all NetSphere pertinent files. Similarly, Trojan GateCrasher also provides commands "*uninstall;*" and "*end;*" to terminate the execution of servers and purge from victim machines all pertinent files. By simulating the roles of

RAT-clients, our *TT* can take over confirmed Trojan sessions and purge *RATs* from victim machines. In the same manner, our *TT* can play the role of the *RAT*-servers, helping collect vital information about attackers without suffering the destructive consequences of *RAT*-servers. In order to take over a detected *RAT* session, the *TT* sends a *TCP RESET* packet or an *ICMP destination unreachable* message to the server. Subsequently, *TT* crafts fake replies for all client-generated commands, and record all input from the attackers. Finally, it is worth pointing out that differentiated actions can be taken according not only on the *RAT* types but also on the transport protocols.

5. Experimental Evaluation of the *RAT Catcher*

We have implemented the proposed *RAT Catcher* in C as a subsystem in the *IPS*-module of *FortiGate*, a multi-function security protection system and a standalone network device providing firewall, AV, and IDS/IPS functionalities [32]. The modular architecture of *FortiGate* forms the basis for its extensibility and scalability, allowing for the seamless coupling of all our *RAT Catcher*-related components. In our experiments we used *FortiGate-300* that operates in inline fashion, has 2 *Gigabyte* main memory, can manage prorated 400 *Mbps* traffic, and maintains upto one million concurrent network connections. We subjected the *RAT Catcher* to a wide range of experiments based on the testbed shown in Figure 7 with a number of test machines undertaking the roles of either *RAT*-servers or clients. Test machines run either *Linux* or *Windows* as a large number of *RAT*-clients and servers are available for these platforms. All test machines were connected to the *FortiGate-300* via two switches supporting 100/1,000 *Mbps* ports: the first simulates the *internal* network where *RAT*-servers are found while the second switch plays the role of *external* network where various *RAT*-clients are operated by attackers. To verify the behavior of our *RAT Catcher*, we installed *Ethereal* traffic sniffers [23] –denoted as *Sniffer*s in Figure 7– to capture data exchanges among *RAT*-clients, *RAT Catcher* and *RAT*-servers. In what follows, we report on our laboratory-based effort to establish the accurate operation of the *RAT Catcher* and baseline performance characteristics. We also report on the deployment and performance of the framework in actual networks.

5.1. Accuracy on *RAT* Detection

Our initial focus was on establishing the accuracy of our *RAT Catcher* in detecting Trojans that use either the *HTTP*, *FTP*, *Telnet* standard specifications or proprietary protocols with the help of the test environment of Figure 7. We also intended to compare the behavior of *RAT Catcher* versus that of *Snort*, an open-source IPS that predominantly uses signatures for *RAT* detection [44]. In [9], we discuss how *Snort* can be used for this purpose and outline specific rules. Every time we implanted a new Trojan server or client on a test machine, we re-installed the OS, network and regular applications of the machine in question to avoid any accidental interference. For all experiments we discuss in this section, the action taken by *RAT Catcher* on identified Trojan sessions is configured to be *pass*, meaning that the *RAT Catcher* only generates alerts for the detected *RAT* sessions and simply forwards all traffic. We proceed with our experiments in three stages with different test procedures: manual test, automated test, and tests designed specially for encrypted *RATs*.

In the manual testing stage, we initially installed all *FTP*-based Trojan servers identified by our *RAT Catcher*, including Eclypse, HellDriver, and Infector, then manually execute their corresponding clients and enter randomly selected *RAT* commands. Our *RAT Catcher* successfully identified all such Trojan communications and generated appropriate alerts. Next, we installed a subset of *RAT* servers of different types on a single test machine denoted as A, carefully configured each of them to avoid any conflict on communication ports, installation locations, clashing file modifications, and activated them

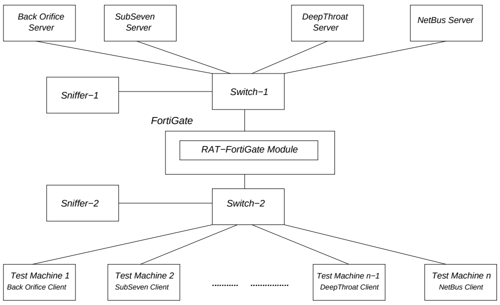

Figure 7. Testbed for *RAT Catcher* subject to real traffic

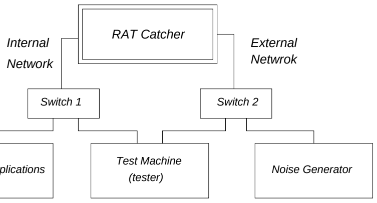

Figure 8. Testbed for *RAT Catcher* subject to synthetic traffic

all simultaneously. By executing *RAT* client programs one at a time to communicate with their own servers on test machine A, our *RAT Catcher* revealed all Trojan activity. We observed the same accurate detection from the *RAT Catcher* when multiple *RAT*-clients were invoked at the same time. The *RAT Catcher* still achieved a perfect detection accuracy even when *background* or attack-free traffic generated by WU-FTPD co-existed during the testing. We successfully performed manual tests with *RATs* based on *HTTP*, *Telnet*, and proprietary protocols by repeating the above test procedure. Using the sniffers of Figure 7, we captured all *RAT* traffic generated during this manual test stage and used it in forthcoming tests. Evidently, the large number of commands available in the repertoires of *RATs* renders manual testing very tedious and time-consuming.

As every individual *RAT* command performs a well-defined operation and there is rarely dependency between different commands, a *RAT* server may not need to be “aware” of the sequence of client-issued commands and it is essentially memory-less. We exploit this memory-less characteristic to automatically generate test cases as follows: for each *RAT* under test, a session template (*TCP* or *UDP*) is first selected, then the template is filled with a sequence of appropriate command/response messages [**]. The ensued traffic is injected into our *RAT Catcher* via a home-made *IPS* testing system [8] termed *tester* as shown in Figure 8. The *tester* can be configured to manipulate the traffic before it is replayed to our *RAT Catcher*; this manipulation includes modification of protocol fields such as *IP* addresses, ports, checksums, and sequence numbers, replacement of packet payloads with arbitrary data, and/or re-shuffling packet orders.

For a *TCP*-based *RAT* connection, the session template consists of three parts: connection establishment, message exchange, and connection termination. The first part contains the three-way handshake procedure with three packets, client-initiated *SYN*, *SYN*|*ACK* from the server, and client’s confirmation *ACK*. The message exchange part

[**] based on the command set supported by individual *RATs*.

contains a series of “command/response(*c/r*)” pairs while the termination part is made up of two “*FIN,ACK*” pairs – one originated from the client and the second from the server. Clearly, packets in both connection establishment and termination have no payload and are fixed; packets in the message exchange part were generated automatically with the help of command sets and possible responses for the *RAT* under test. The *UDP* session template simply consists of a sequence of “command/response (*c/r*)” pairs specified by testers or automatically generated. Such sequences of commands are randomly chosen from the command sets for the *RATs* under testing. In addition, the number of *RAT* sessions to be created and the appearance frequency of each command in the generated traffic are configurable. To simulate multiple concurrent sessions, our *tester* can generate an arbitrary number of test cases, interleave them together, and shuffle the replay order before feeding into the *RAT Catcher*.

We employed SubSeven in the automated testing stage to show the detection accuracy of both *RAT Catcher* and *Snort*. Overall, we generated 100,000 of *TCP*-based SubSeven sessions; each session contained a random number –in the range of [1, 20]– of “command/reply (*c/r*)” pairs whose sizes and payloads were randomized unless specific format/parameter requirements are necessitated by the protocol. The generated *RAT* sessions were injected into both *RAT Catcher* and *Snort*. The outcome of the experiment appears in Table X. Sessions are grouped according to the number of their “command/reply (*c/r*)” pairs and column *cnt* shows the number of Trojan-sessions per group. Column *no-reply* shows the number of sessions from each group that do not trigger any server-reply. Our *RAT Catcher* initially uses one or more messages of a session to determine the potential *RAT* type of the session which subsequently is confirmed by additional messages received. The two columns *RAT Catcher detect* and *RAT Catcher confirm* indicate the number of sessions tentatively labeled as Trojan and those confirmed as such by our *RAT Catcher*. Table X also shows respective results obtained with *Snort(v.2.2)* having all specific signatures for detecting SubSeven enabled.

“c/r”	*cnt*	*no-reply*	*reply*	*RAT Catcher detect (%)*	*RAT Catcher confirm (%)*	*Snort detect (%)*	*Snort miss (%)*
1	4865	118	4747	4865 (100.00)	4747 (97.57)	95 (1.95)	4770 (98.05)
2	5062	2	5060	5062 (100.00)	5062 (100.00)	176 (3.48)	4886 (96.52)
3	5069	0	5069	5069 (100.00)	5069 (100.00)	276 (5.44)	4793 (94.56)
4	5318	0	5318	5318 (100.00)	5318(100.00)	349 (6.56)	4969 (93.44)
5	4911	0	4911	4911 (100.00)	4911 (100.00)	410 (0.083)	4501 (91.65)
6	5171	0	5171	5171 (100.00)	5171 (100.00)	556 (10.75)	4615 (89.25)
7	4979	0	4979	4979 (100.00)	4979 (100.00)	570 (11.45)	4409 (88.55)
8	5306	0	5306	5306 (100.00)	5306 (100.00)	683 (12.87)	4623 (87.13)
9	5038	0	5038	5038 (100.00)	5038 (100.00)	761 (15.11)	4277 (84.89)
10	4857	0	4857	4857 (100.00)	4857 (100.00)	723 (14.89)	4134 (85.11)
20	4879	0	4879	4879 (100.00)	4879 (100.00)	1503 (30.81)	3376 (69.19)

Table X. Testing *RAT Catcher* and *Snort* using SubSeven-based test cases

Table X shows that the *RAT Catcher* detects all SubSeven sessions and its overall *confirm* rate is nearly perfect. Only in the first group, a few instances of unconfirmed Trojan connections appear as single message-sessions offering no option for confirmation. Nevertheless, all SubSeven are properly tagged and only 2.43% of *RAT*-sessions cannot be confirmed. On the contrary, the accuracy of *Snort* is far from satisfactory with worst 1.95% detection rate for group 1, best 30.81% for group 20, and average 16.59% for all cases. This is attributed to the fact the *Snort* uses only three rules for SubSeven. Clearly, we could craft signatures in *Snort* to cover all SubSeven commands but this would greatly burden its operation and deteriorate its performance to an unacceptable level. We repeated our testing for traffic generated by *FTP*, *HTTP*, and *Telnet*-based Trojans using synthetic background noise with the layout of Figure 8. We experimented with various intensities of background noise traffic regulated by the machines making up the *noise generator* and established very similar results to those shown in Table X.

In the last stage of our baseline experimentation, we dealt with Trojans that use encryption algorithms. As such *RATs* have extra dimensions of freedom–encryption algorithms and encryption keys–and share little commonality in their encryption processes,

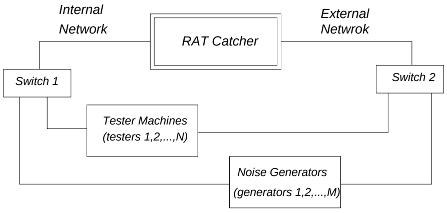

Figure 9. Testbed with multiple testers to generate *RAT* traffic and multiple testers for background noise

we designed special testing procedures that we discuss with the help of Back Orifice (BO)- and NetBus Pro-generated traffic. We used the Internet available source-code for the BO-client and changed its encryption seed for each newly created session. The various streams of ensued traffic –resembling that of Table III– were fed into both *RAT Catcher* and *Snort*. In all instances, *RAT Catcher* and *Snort* maintain the same detection accuracy as both resort to application-layer protocol dissection. While experimenting with NetBus Pro, we deployed a synthetic approach as NetBus Pro uses a proprietary encryption algorithm and its source code is not known. We generated the 10-byte application protocol header and randomly selected payload for every created NetBus Pro packet. We generated upto 1,000,000 NetBus Pro sessions and injected this synthetic traffic to both *RAT Catcher* and *Snort*. Our experiments showed that the *RAT Catcher* creates no false negatives achieving the perfect detection rate while *Snort* fails to recognize most of the generated sessions. *Snort* appears to be ineffective as it can detect only those NetBus Pro sessions that happen to have client-initiated and server-originated packets with sizes of *0x20* and *0x10* Bytes, respectively. In contrast, the NetBus Pro protocol analyzer [††] of *RAT Catcher* treats the *message-size* field as a variable and therefore can recognize NetBus Pro messages of any size. Moreover, the *RAT Catcher* resorts to message structure analysis and correlation of bi-directional traffic within a session to further improve its detection accuracy. We have repeated the same approach in experimenting with combinations of *RAT* and attack-free traffic flows and obtained similar results for the performance of *RAT Catcher* and *Snort*. In summary, for Trojans based on *HTTP, FTP, Telnet*, or proprietary protocols, the *RAT Catcher* creates no false positives/negatives.

5.2. Scalability and Performance Under Various Workloads

As most *RAT*-clients are human-operated, the time gap between two consecutive commands submitted to the server is often long reflecting the attacker's thinking time. The time-stamped SubSeven message exchanges of Table VII indicate that this thinking time can be as long as *60 seconds* –between messages 13 and 14– while on average is *40 seconds*. This interactive nature of Trojans seems to indicate that processing overheads incurred by our *RAT Catcher* should not be a concern. However, as *FortiGate* is to be typically deployed at the edge between internal and external networks, it may often encounter in excess of half a million concurrent sessions and response times of corresponding sessions may be noticeably affected.

We evaluate the capabilities of the *RAT Catcher* under the above circumstances with the help of the testbed of Figure 9 and the traffic of Table IV. We use the latter as a template to generate various test cases and we split it into two parts: the first consists of the packets making up the normal *TCP* three-way handshake –not shown in the Table for brevity– and

[††] of Algorithm 4

packet 1; the second part contains the remaining packets 2–7 as well as the four packets of the normal disconnection (also not shown). We configure our system to generate two alerts for each NetBus Pro session: the first is created when the session is marked as NetBus Pro by using data stream from the client to server and the second is raised when the session is confirmed as a true NetBus Pro with the help of data stream from the server to client. *RAT Catcher* generates the first alert when packet 1 is encountered and subsequently confirms the session after it has processed packet 2. In our experimentation, we configure the *RAT Catcher* to forward all traffic even if Trojans have been detected and in our *RAT Catcher* implementation, we use the *LRU* to replace sessions when the memory is full. As the generation of the second alert depends on the presence of the first alert and the availability of the session information, *RAT Catcher* may fail to raise this second alert if the *RAT* session in question is evicted due to memory congestion.

In all the tests performed in this section, N=20 machines are used as *testers* to create *foreground traffic* as follows: they replay the first half of the trace in Table IV for n times, therefore generating n *RAT* sessions; then pause for *1 second* and then replay the second part of the trace for n times; n assumes values in the range of [10,000, 700,000]. Meanwhile, we use M=2 noise-generator machines whose purpose is to create m sessions of *background traffic* using the attack-free *WU-FTPD* application – m takes values in the range [1, 50,000]. We also elect to introduce artificial delays in the replay of background traffic by splitting each *WU-FTPD* session into two parts and stall the noise-generators for *1 second* in between replaying these two parts. Each time a trace is replayed, the *testers* modify *IP*s and port numbers of both source and destination of connections to avoid conflicts among different replayed sessions. Similarly, the noise-generators also change protocol fields accordingly so that source and destination *IP*s and ports do not present conflicts. To simplify the coordination among different test machines, we assign non-overlapping IP address ranges to different machines. For every test, we observe the behavior of *RAT Catcher* by recording the number of sessions that are correctly marked as NetBus Pro with both alerts generated, and calculate the ratio of such correctly marked sessions over the total n replayed NetBus Pro sessions.

case #	↓ m *; FTP delay*	n*=10,000*	n*=50,000*	n*=75,000*	n*=100,000*	n*=200,000*	n*=500,000*	n*=700,000*
1	1; no	100.00	100.00	100.00	100.00	100.00	100.00	100.00
2	1; 1 second	100.00	100.00	100.00	100.00	100.00	100.00	100.00
3	10,000; no	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4	10,000; 1 second	100.00	100.00	100.00	100.00	100.00	100.00	99.99
5	20,000; no	100.00	100.00	100.00	100.00	100.00	100.00	100.00
6	20,000; 1 second	100.00	100.00	100.00	100.00	100.00	100.00	99.99
7	30,000; no	100.00	100.00	100.00	100.00	100.00	100.00	100.00
8	30,000; 1 second	100.00	100.00	100.00	100.00	100.00	100.00	99.98
9	40,000; no	100.00	100.00	100.00	100.00	100.00	100.00	100.00
10	40,000; 1 second	100.00	100.00	100.00	100.00	100.00	100.00	99.98
11	50,000; no	100.00	100.00	100.00	100.00	100.00	100.00	100.00
12	50,000; 1 second	100.00	100.00	100.00	100.00	100.00	100.00	99.80

Table XI. Test results of our *RAT Catcher* under various traffic workloads

Table XI shows the results of all our tests conducted under diverse workloads. The *RAT Catcher* demonstrates the correct behavior when no artificial delays are introduced by the noise-generators regardless of the volume of *RAT* traffic and intensity of the background noise. When artificial delays are in place while replaying *WU-FTPD* packets, the *RAT Catcher* fails to generate the second required alert –confirmation marking– for a small number of Trojan sessions as we increase the volumes of foreground and background traffic. Nevertheless, even for this small fraction of sessions, *RAT Catcher* still carries out the tentative marking. Overall, there is no *RAT* session completely missed by our framework. In addition, we also manually verify that no background traffic is mistakenly identified as *RATs* or dropped due to *RAT Catcher* malfunction. In all cases, the latency for SubSeven traffic is statistically the same with and without *RAT Catcher* present in *FortiGate*. By repeating the aforementioned procedure with other *RATs* including SubSeven, DeepThroat, BO2K, and NetSphere we obtain similar results.

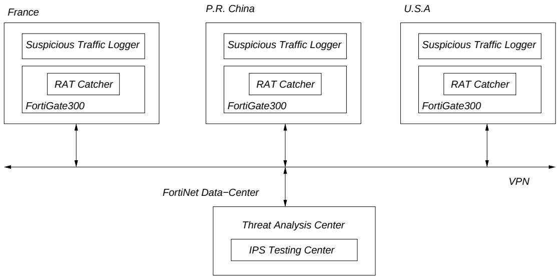

Figure 10. Deployment of *RAT Catcher* in France, P.R. China and United States

5.3. The *RAT Catcher* in the Real World

To evaluate the effectiveness of our approach in a real world setting, we have deployed *FortiGate-300* devices in 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 10 to store and forward both confirmed and suspicious traffic streams containing attacks to a corporate *Threat Analysis Center (TAC)* for manual processing and verification purposes. In order to help discover new types of attacks and better ascertain the false-negative rate of *FortiGate-300*, we augment *RAT Catcher* with the *Suspicious Traffic Logger (STL)* module that simply logs likely malicious yet not-locally resolved streams/sessions. Regional devices periodically transfer data on detected *RATs* and un-identified but suspect connections to *TAC* where sampling and analyses on the traffic are performed. In a similar fashion, the collected suspicious traffic by *STL* is manually analyzed for new strains of Trojans. Once the confirmed Trojan traffic is stored, we also use *Snort* to assess its effectiveness by computing the false negative rate while portions of confirmed attack-free traffic from *STL* is replayed to *Snort* for gauging its false positive rate. From the forwarded traffic to *TAC*, we present all detected *RAT* instances for Back Orifice, NetBus and SubSeven during the period of June 26th, 2006 to July 9th, 2006 in Table XII.

day	*Back Orifice*			*NetBus*			*SubSeven*		
	US,CN,FR/Total	*Catcher*	*Snort*	*US,CN,FR/Total*	*Catcher*	*Snort*	*US,CN,FR/Total*	*Catcher*	*Snort*
1	204, 115, 46/365	365/0	365/0	55, 25, 17/97	97/0	63/34	32, 15, 4/51	51/0	33/18
2	87, 75, 29/191	191/0	191/0	86, 68, 36/190	190/0	143/47	26, 12, 16/54	54/0	32/22
3	197, 132, 104/433	433/0	433/0	53, 32, 10/95	95/0	64/31	11, 6, 8/25	25/0	16/9
4	168, 64, 40/272	272/0	272/0	108, 47, 32/187	187/0	117/70	46, 34, 11/91	91/0	62/29
5	265, 119, 14/398	398/0	398/0	56, 34, 10/100	100/0	74/26	36, 24, 2/62	62/0	36/26
6	96, 51, 67/214	214/0	214/0	89, 36, 4/129	129/0	84/45	17, 8, 2/27	27/0	19/8
7	230, 95, 14/339	339/0	339/0	51, 28, 17/96	96/0	47/49	33, 25, 7/65	65/0	49/16
8	205, 121, 99/425	425/0	425/0	115, 50, 29/194	194/0	90/104	50, 26, 15/91	91/0	59/32
9	43, 25, 17/85	85/0	85/0	65, 41, 16/122	122/0	56/66	12, 4, 3/19	19/0	15/4
10	218, 107, 18/343	343/0	343/0	66, 29, 11/106	106/0	77/29	28, 20, 14/62	62/0	27/35
11	125, 64, 37/226	226/0	226/0	79, 53, 32/164	164/0	109/55	42, 16, 3/61	61/0	37/24
12	182, 93, 43/318	318/0	318/0	77, 30, 19/126	126/0	67/59	42, 18, 19/79	79/0	45/34
13	246, 123, 51/420	420/0	420/0	104, 53, 24/181	181/0	122/59	49, 27, 13/89	89/0	38/51
14	96, 54, 13/163	163/0	163/0	56, 31, 30/117	117/0	80/37	38, 17, 17/72	72/0	40/32

Table XII. Detection rates by *RAT Catcher* and *Snort* in our real-world traffic

Table XII points out most *RAT* instances occur in U.S. followed by China and France; Back Orifice is the most frequently found *RAT* in the three regions, followed by NetBus and SubSeven as shown in column *US,CN,FR/Total*. In *TAC*, we manually establish that all *RAT* sessions shown in Table XII detected and reported by our regionally placed *RAT Catcher*s have been indeed correctly identified.

Table XII shows the detection/prevention results and false negatives by *RAT Catcher* and *Snort* in Columns *Catcher* and *Snort* in the format of *correct/error*, should all identified *RAT* sessions at *TAC* be replayed to *RAT Catcher* and *Snort* when both are configured to block all *RAT* sessions. Clearly, *RAT Catcher* does not create any false positives/negatives and successfully blocks all identified *RAT* connections. Moreover, it typically determines the application type of a session after inspecting the first message from each direction of the session; so, it effectively forwards no attacker-submitted command and minimizes potential damage to victims. On the other hand, *Snort* detects all Back Orifice connections as it integrates a dedicated protocol dissector which performance-wise is equivalent to that of our *RAT Catcher*. However, *Snort* misses a number of NetBus and SubSeven connections, generating false negatives since the injected traffic has been manually evaluated to be malicious. For instance, *Snort* rule 3009 [9] used to identify NetBus Pro contains pattern "*BN*|*20 00 02 00*|" where the byte sequence "|*20 00*|" immediately following string "*BN*" designates the size of the message. Obviously, *Snort* rule 3009 assumes that all NetBus Pro messages with *command-code 0x05* should be *0x20* bytes. However, the sizes of NetBus Pro messages with *command-code 0x05* can vary in different sessions as demonstrated by the captured traffic shown in the first part of Table XIII; this leads to a false negative by the *Snort* rules in place as the connection request from the NetBus Pro-client has size of 31 bytes (*0x1F*) instead of 32 bytes (*0x20*).

#	*dir*	*payload*	*description*
NetBus Pro traffic – protocol: *TCP*; attacker denoted as A; victim denoted as V			
1	A:35821− >V:20034	*BN* |1F 00 02 00 DC 33 05 00 41 0C 69 1F| |5D 28 5B 95 9C AD 95 A8 E6 28 FD...|	msg starts with *BN*; size: 0x1F; ver: 0x02; cmd: 0x05 (connection establishment);
2	V:20034− >A:35821	*BN* |10 00 02 00 DC 33 05 00 41 0C 69 1F 5D 28|	reply msg; size: 0x10; ver: 0x02; cmd: 0x05 (con. est.);
Normal *FTP*-traffic – protocol: *TCP*; *FTP* server as S; FTP client denoted as C			
1	C:46943− >S:21	PASV|0D 0A|	client requests "passive" mode
2	S:21− >C:46943	227 Entering Passive Mode (x, y, z, w, 66, 63)	server listens on port 16959 (66 * 256 + 63)
3	C:46944− >S:16959	(SYN)	client requests data connection
4	S:16959− >C:46944	(SYN|ACK)	server accepts the data connection
5	C:46944− >S:16959	(ACK)	client confirms data connection
6	C:46943− >S:21	RETR commands.txt|0D 0A|	client requests file "commands.txt"
7	S:21− >C:46943	150 ASCII data connection for commands.txt	response from server
8	S:16959− >C:46944	PWD - print name of current/working directory ...	content of file "commands.txt"

Table XIII. Traffic causing *Snort* to generate false positives and negatives

By replaying *STL*-collected traffic that has been manually verified as legitimate, we can establish that *Snort* also generates false-positives. For example, the second part of Table XIII shows a normal *FTP* session and its data connection for a file transfer. In message 1, the *FTP*-client requests "passive mode" which is approved by the server in message 2. At the same time, the server also informs the client of its intent to use the port *16959* for the data connection. In messages 3 to 5, a new *TCP* connection is established and subsequently used to deliver the content of the file *commands.txt* requested by the client in message 6. As the pattern *PWD* is part of the transported content of message 8, it triggers *Snort* rule 107 discussed in [9] which is obviously a false alarm. Overall, we have observed that *RAT Catcher* correctly identifies and subsequently blocks traffic streams known to be the result of *RATs* whereas *Snort* lags behind due to limited number of specific rules for each Trojan, yielding both false positives and negatives.

6. Conclusions and Future Work

A Remote Administration Trojan (*RAT*) is a malicious program that allows an attacker to remotely control a computing system often creating irrevocable damage. Existing techniques including fingerprinting, auto-start monitoring, surveillance of network activities and packet analysis using static signatures and/or fixed communication ports are limited in both scope and effectiveness. Today, traffic obfuscation, port hopping, file renaming and compression, information encryption along with evasion techniques work counter to the effectiveness and efficiency of anti-Trojan systems. In this paper, we propose the *RAT Catcher*, a network-based framework for Trojan detection that operates in *inline* fashion at the edge of the network and reliably identifies *RAT* activities.

The *RAT Catcher* inspects every passing packet and maintains information for the entire lifetime of sessions created by both Trojans and normal applications. This session tracking improves detection accuracy by providing stateful inspection as well as intra-session and inter-session data correlation. The *RAT Catcher* stores all packets in every data stream, reassembles them, and interprets the resulting data aggregations according to known Trojan protocols. To this end, the framework performs deep inspection on data streams by scanning protocol fields and whenever feasible message content. The *RAT Catcher* dissects both data streams within a session and correlates them in order to ensure that the traffic in both directions complies with protocol specifications defined by *RAT* systems in terms of syntax and semantics. By analyzing the syntax/format of application messages and inspecting the exchange order of messages between clients and servers, our *RAT Catcher* can defeat evasion techniques. By associating *RAT* control and data channels, correlating active with defunct sessions, and restoring boundaries of application messages through reassembly, the *RAT Catcher* does not generate false positives or negatives. Actions imposed on identified Trojan sessions include alerting, packet blocking, session take-over, and connection termination.

Experiments showed that the proposed framework is both effective and efficient. Subjected to comprehensive testing, the *RAT Catcher* demonstrated wide *RAT* coverage, excellent detection accuracy, and low processing overheads. We plan to further pursue our work in the area in at least three directions: first, we will keep enhancing our *RAT Catcher* so that it can deal with new types of Trojans as the latter become known; second, develop advanced techniques to identify *RATs* that use strong cryptographic mechanisms and finally, explore the integration of our *RAT Catcher* with other security systems including firewalls, anti-virus, and anti-malware programs to more effectively combat aggregate malicious activities resulting from the mixture of *RAT* and popular worms known as *Blended Threats*.

Acknowledgments: We are very grateful to the anonymous reviewers for their detailed comments and suggestions that helped us significantly improve our work. We also thank Qinghong Yi, Gary Duan, Ping Wu, Joe Zhu, Yelin Guan, and Hong Huang for helping us with various aspects of the *RAT Catcher* implementation and testing and Prof. Boris Aronov for comments on earlier versions of the manuscript.

REFERENCES

1. C. M. Adams and S. E. Tavaris. Designing S-Boxes for Ciphers Resistant To Differential Cryptanalysis. In *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, pages 181–190, Rome, Italy, Feb. 1993.
2. L. Adelman. An Abstract Theory of Computer Viruses. In *Proceedings of Advances in Cryptology (CRYPTO'88)*, pages 354–374, New York, NY, Aug. 1988. Springer-Verlag.
3. Korea Information Security Agency. Security Incident Statistics in Korea. *http://www.kisa.or.kr/english/statistics/hack/*, 2000.
4. D. Bell and L. LaPadula. Secure Computer Systems: Unified Exposition and MULTICS Interpretation. Technical report, MITRE Corporation, Bedford, MA, July 1975. MTR-2997.
5. M. Bishop. A Model of Security Monitoring. In *Proceedings of the Fifth Annual Computer Security Applications Conference*, pages 46–52, Dec. 1989.
6. CERT. Advisory CA-1999-01: Trojan Horse Version of TCP Wrappers. *http://www.cert.org/advisories/CA-1999-01.html*, 1999.
7. CERT. Advisory CA-1999-02: Trojan Horses. *http://www.cert.org/advisories/CA-1999-02.html*, 1999.
8. Z. Chen, P. Wei, and A. Delis. A Pragmatic Methodology for Testing Intrusion Prevention Systems (IPSs). Technical report, Deprt. of Informatics & Telecommunications, Univ. of Athens, Athens, Greece, August 2004.
9. Z. Chen, P. Wei, and A. Delis. Catching Remote Administration Trojans. Technical report, Deprt. of Informatics & Telecommunications, Univ. of Athens, Athens, Greece, May 2007. www.di.uoa.gr/∼ad.
10. D. Clark and D. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 184–194, Apr. 1987.
11. F. Cohen. Computer Viruses: Theory and Experiments. *Computers and Security*, 6(1):22–35, Feb. 1987.
12. F. Cohen. Computational Aspects of Computer Viruses. *Computers and Security*, 8(4):325–344, Jun. 1989.
13. F. Cohen. Practical Defenses Against Computer Viruses. *Computers and Security*, 8(2):149–160, Apr. 1989.
14. Symantec Com. Norton Anti-Virus System. *http://www.symantec.com/*.
15. D. E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

16. Commodon. Threats to Your Security on the Internet-SubSeven. http://www.commodon.com, 2001.
17. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Boston, MA, 1997.
18. Privacy Software Corporation. Anti-Trojan Program: BOClean. *http://www.nsclean.com/trolist.html*, 2004.
19. H. DeMaio. Viruses - Management Issue. *Computers and Security*, 8(5):381–388, Oct. 1989.
20. D. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, Feb. 1987.
21. D. Denning. The Science of Computing: Computer Viruses. *American Scientist*, 76(3):236–238, May 1988.
22. T. Duff. Experiences with Viruses on UNIX Systems. *Computing Systems*, 2(2):155–172, Spring 1989.
23. Ethereal. Ethereal: Powerful Multi-Platform Analysis. *http://www.ethereal.com*, May 2004.
24. R. Farrow. *UNIX System Security*. Addison-Wesley Publishing Co., Reading, MA, 1991.
25. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. B. Lee. Hypertext Transfer Protocol – HTTP/1.1. *Internet Engineering Task Force*, June 1999.
26. S. Garfinkel and G. Spafford. *Practical UNIX Security*. O'Reilly and Associates, Sebastopol, CA, 1991.
27. W. Gleissner. A Mathematical Theory for the Spread of Computer Viruses. *Computers and Security*, 8(1):35–41, Feb. 1989.
28. S. Gordon and D. M. Chess. Attitude Adjustment: Trojans and Malware on the Internet. In *Proceedings of the EICAR*, pages 183–204, Copenhaguen, Denmark, May 1999.
29. R. Hansen. Moose Test for Windows: NetBus Pro and How it Happened. *http://www.heise.de/ct/english/99/17/088/*, 1999.
30. H. M. Heys and S. E. Tavares. On the Security of the CAST Encryption Algorithm. In *Proc. of Canadian Conf. on El. & Comp. Eng.*, Halifax, Canada, Sep. 1994.
31. L. Hoffman. *Rogue Programs: Viruses, Worms, and Trojan Horses*. Van Nostrand Reinhold, New York City, NY, 1990.
32. Fortinet Inc. Intrusion Prevention System. *Web Site: www.fortinet.com*, Oct. 2004.
33. Foundstone Inc. FPort: Intrusion Detection Tool. *http://www.foundstone.com/*.
34. M. Joseph and A. Avizienis. A Fault Tolerant Approach to Computer Viruses. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 52–58, Oakland, CA, Apr. 1988.
35. G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communication Security*, pages 18–29, Fairfax, VA, 1994.
36. S. Kulakow. NetBus 2.1: Is It Still a Trojan Horse or an Actual Valid Remote Control Administration Tool? *Web page*, 2001.
37. S. Lipner. Non-Discretionary Controls for Commercial Applications. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 2–10, Apr. 1982.
38. Zone Labs LLC. ZoneAlarm Security Suite. *http://www.zonelabs.com/*, 2004.
39. T. Lunt and R. Jagannathan. A Prototype Real-Time Intrusion-Detection Expert System. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 59–66, Apr. 1988.
40. PC Magazine. StartupCop Pro. *http://www.pcmag.com/*.
41. H. Nussbacker. Israeli Internet Hacking Analysis for 2000. In *Proceedings of the Internet Society of Israel Conference*, Tel Aviv, Israel, March 2001. http://www.isoc.org.il/conf2001/presentations/nussbackerl.ppt.
42. The Cult of the Dead Cow. Back Orifice 2000. *http://www.bo2k.com or http://www.cultdeadcow.com/*, 2004.
43. PestPatrol. About RATS: SubSeven and Remote Administration Trojans. http://www.pestpatrol.com/whiterpapers, 2006.
44. M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *USENIX 13-th Systems Administration Conference – LISA'99*, Seattle, Washingto, 1999.
45. J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.
46. B. Schneier. *Applied Cryptography, Protocols, Algorithms, and Source Code in C, Second Edition*. John Wiley & Sons, Inc., 1996.
47. Mischel Internet Security. TrojanHunter vs. the Parasitic Beast Trojan. *http://www.misec.net/papers/thvsbeast/*, 2004.
48. D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.
49. Winternals Software. TCPView Pro. *http://www.winternals.com/*.
50. D. Song, G. Shaffer, and M. Undy. Nidsbench – A Network Intrusion Detection Test Suite. In *Second International Workshop on Recent Advances in Intrusion Detection (RAID 1999)*, West Lafayette, 1999.
51. Inc. Sourcefire. Snort 2.0: Detection Revisited. *White Paper*, February 2003.
52. E. Spafford. The Internet Worm Program: An Analysis. *ACM Computer Communications Review*, 19(1), Jan. 1989.
53. R. C. Summers. *Secure Computing Threats and Safeguards*. McGraw-Hill, 1997.
54. M. Swimmer. Dynamic Detection and Classification of Computer Viruses. In *Proceedings of the Sixth International Virus Bulletin Conference*, pages 149–159. Virus Bulletin Ltd, 1996.
55. Internet Security Systems. ISS Security Alert: Windows Backdoor Update III. *http://www.xforce.iss.net*, 1999.
56. H. Teng, K. Chen, and S. Lu. Adaptive Real-Time Anomaly Detection Using Inductively Generated Sequential Patterns. In *Proceedings of the 1990 Symposium on Research in Security and Privacy*, pages 278–284, May 1990.
57. I. Whalley. Testing Times for Trojans. In *Proceedings of the Ninth International Virus Bulletin Conference*, pages 55–67, September/October 1999.
58. C. Young. Taxonomy of Computer Virus Defense Mechanisms. In *Proceedings of the Tenth National Computer Security Conference*, pages 220–225, Oakland, CA, Sep. 1987.