An Inline Detection and Prevention Framework for Distributed Denial of Service Attacks

Zhongqiang Chen*
Department of Computer & Information Science
Polytechnic University
Brooklyn, NY 11201
zchen@milos.poly.edu

Zhongrong Chen
ProMetrics Consulting Inc.
480 American Ave.
King of Prussia, PA 19406
zhongrongchen@prometrics.com

Alex Delis†
Deprt. of Informatics & Telecommunications
University of Athens
15771, Athens, Greece
ad@di.uoa.gr

July 18, 2006

Abstract

By penetrating into a large number of machines and stealthily installing malicious pieces of code, a distributed denial of service (*DDoS*) attack constructs a hierarchical network and uses it to launch coordinated assaults. *DDoS* attacks often exhaust the network bandwidth, processing capacity and information resources of victims, thus, leading to unavailability of computing systems services. Various defense mechanisms for the detection, mitigation and/or prevention of *DDoS* attacks have been suggested including resource redundancy, traceback of attack origins, and identification of programs with suspicious behavior. Contemporary *DDoS* attacks employ sophisticated techniques including formation of hierarchical networks, one-way communication channels, encrypted messages, dynamic ports allocation, and source address spoofing to hide the attackers' identities; such techniques make both detection and tracing of *DDoS* activities a challenge and render traditional *DDoS* defense mechanisms ineffective.

In this paper, we propose the *DDoS Container*, a comprehensive framework that uses network-based detection methods to overcome the above complex and evasive types of attacks; the framework operates in "inline" mode to inspect and manipulate ongoing traffic in real-time. By keeping track of connections established by both potential *DDoS* attacks and legitimate applications, the suggested *DDoS Container* carries out stateful inspection on data streams and correlates events among sessions. The framework performs stream re-assembly and dissects the resulting aggregations against protocols followed by various known *DDoS* attacks facilitating their identification. The traffic pattern analysis and data correlation of the framework further enhance its detection accuracy on *DDoS* traffic camouflaged with encryption. Actions available on identified *DDoS* traffic range from simple alerting to message blocking and proactive session termination. Experimentation with the prototype of our *DDoS Container* shows its effectiveness in classifying *DDoS* traffic.

Indexing Terms: Distributed Denial of Service (*DDoS*) attacks, *DDoS* handlers and agents, flooding attacks, *DDoS* detection, mitigation, and prevention mechanisms

*Work done while the author was with Fortinet Inc., in Sunnyvale, CA.

†Partially supported by Pythagoras grant No.7410 and a Univ. of Athens–Research Foundation grant.

1 Introduction

Distributed Denial of Service (*DDoS*) attacks exploit host vulnerabilities to initially break into a large number of systems [26, 40]. A subset of these systems termed secondary victims, function as daemons or agents and dispatch useless traffic to specific network nodes known as primary victims. The work of agents is coordinated by a core of compromised sites which become the masters or handlers of an assault and are under the direct control of attackers. In this manner, hierarchical attack networks are formed and *DDoS* attacks can be launched [26, 23, 19]. Vulnerabilities exploited by *DDoS* are mainly due to the ambiguities in network protocols and flaws in their implementations [55]. For instance, the *Targa* flooding attack used by many *DDoS* tools crashes primary victims with malformed packets or illegal sequences of messages [55, 40]. Moreover, logic errors in programs such as *echo* and *chargen* services, system mis-configurations including support of direct broadcasts, and an enormous number of authentication-unaware applications frequently facilitate the formation of *DDoS* attack networks [9, 32, 43].

In a typical *DDoS* attack, high volume of artificial traffic is generated in order to exhaust network bandwidth, waste CPU processing, and/or inundate critical information resources, rendering the victim system inaccessible to its legitimate users [10, 40]. By following the end-to-end design paradigm, the current Internet architecture places minimum functionality in intermediate switches, routers and gateways to achieve high-throughput and gives little emphasis to security and accountability of such backbone elements [10, 40, 25]. The Internet's inherent distributed control makes the defense of a single site irrelevant when it comes to *DDoS* attacks [26, 55]. The synergy of multiple autonomous systems, the asymmetric placement of computing resources and the lack of any intelligence in intermediate nodes make it impractical to impose Internet-wide security policies that control cross-domain attacks [12, 25]. It is in this context that *DDoS* networks can bombard victims with attacks that use IP addresses from unallocated Internet address blocks making their origin nearly impossible to locate [48, 18].

The availability of *DDoS* scripts and their ever improving user-interfaces literally make an attack a click away [10, 40]. It is noteworthy that various *DDoS* attack stages, including discovery of weak Internet links, penetration of vulnerable sites, installation of malicious codes, and ignition of coordinated attacks, can be highly automated and performed in a "batch" fashion. It thus becomes a straightforward task to establish a large *DDoS* attack network with minimal effort [40, 31]. Furthermore, the increasing population of "always-on-but-unattended" Internet systems substantially contributes to the success of large-scale coordinated attacks that have appeared in dramatically increasing rates since 1999 [45, 31, 55]. Well publicized attacks targeted popular e-commerce sites including Yahoo, Ebay, and E*trade in 2000 [65]; Microsoft's name service was entirely crippled for days in 2001 [16]; while in 2002, thirteen top-level Internet domain name servers were flooded simultaneously and seven of them were entirely shut down [45]. In order to better understand the severity and intensity of *DDoS* attacks, backscatter analysis was used [42]; it is speculated that 12,805 *DDoS* attacks occurred in a period of three weeks in 2001 with more than 5,000 distinct victims belonging to 2,000 different domains. Among them, 90% lasted for one hour or less, 90% were TCP-based, and approximately 40% were launched with intensity larger than 500 packets per second (pps), with the maximum rate at around 500,000 pps.

A number of mechanisms have been proposed to prevent, mitigate, and curb the immensely destructive effects of *DDoS* attacks [22, 4]. Preventive measures attempt to eliminate the necessary conditions for the formation of *DDoS* networks with the help of vulnerability assessment tools, periodic network penetration tests, and validation mechanisms against malicious pieces of code [58, 31]. By deploying distinct server pools, load-sharing, traffic policing via shaping, and dynamic network reconfiguration [3, 54], computing systems try to mitigate *DDoS* attack effects. Reactive mechanisms initially detect by searching for unique byte patterns termed telltales [5] and subsequently block malicious activities [22, 5]. Identifying the origins of attacks is also critical to attack accountability and a number of strategies including *ICMP Traceback*, *IP Traceback* and *CenterTrack* have been proposed to this effect [48, 53, 18, 14]. However, the hierarchical nature of *DDoS* networks which separates control flow from attacking traffic in connection with identity spoofing makes network path tracing extremely difficult effectively shielding the assault instigators [23, 46, 35]. Consequently, traceback systems often lead only to zombies instead of intruders, inevitably limiting their usefulness [6, 57, 48]. Intrusion detection/prevention systems (IDSs/IPSs) also do not fare well as *DDoS* attacks camouflage their traffic [10] using one-way channels not only between attackers and masters but also between handlers and daemons; such unidirectional flows make it a challenge for IDSs/IPSs to identify culprits [40]. The use of strong cryptographic algorithms including the advanced encryption standard (*AES*) and *Blowfish* to obfuscate traffic in *DDoS* tools renders many IDSs/IPSs ineffective [49, 40]. In addition, the use of dynamically assigned TCP/UDP ports, covert channels and multiple transport services (e.g., TCP, UDP, and ICMP) also affects the detection accuracy of most IDSs/IPSs as they typically employ fixed-port detection mechanisms [47, 5].

In order to counter the above-mentioned evasive and complex techniques, we propose the *DDoS Container*, a network-based detection/prevention framework that functions in "inline" mode, inspects every passing packet, and therefore blocks any *DDoS* traffic in real-time. In order to track suspicious activity, our framework monitors sessions established among *DDoS* attackers, handlers, and zombies as well as legitimate applications, records and maintains state information for the lifetime of each session, and finally archives such information once sessions terminate to help conduct post-mortem intra-session data fusion and inter-session correlation. Our *DDoS Container* stores encountered packets in every data stream, reassembles them in correct order, and interprets the resulting aggregations against protocols followed by *DDoS* tools such as *Stacheldraht*, *TFN2K*, and *Trinoo*. This type of message sequencing morphs segmented data streams into sequences of comprehensive *DDoS* messages, facilitating the analysis and classification of pertinent traffic. To further enhance its reliability and detection accuracy, our framework performs application layer or "deep" inspection by scanning both protocol headers and payloads; the *DDoS Container* analyzes the syntactic structures and patterns of traffic flows to identify *DDoS* activities that may use encryption. As soon as a session has been identified as *DDoS* traffic, our framework can alert the user, block the flow and/or even pro-actively terminate the connection. In ascertaining the effectiveness of our approach, we carry an experimental evaluation with our *DDoS Container* prototype. Our results show that our framework accurately identifies *DDoS* control traffic among attackers, masters, and agents, and detects flooding attacks quickly, therefore delivering its functionality in a robust and efficient way.

The remainder of this paper is organized as follows: Section 2 discusses key features manifested by *DDoS* attacks. Section 3 presents the functionalities and components of our framework while Section 4 outlines the operation of *DDoS* analyzers. Results of our experimental evaluation are discussed in Section 5 while related work and concluding remarks are found in Sections 6 and 7 respectively.

2 Key Features of Contemporary *DDoS* Attacks

In this section, we outline basic mechanisms used to deploy a *DDoS* network and discuss its communication channels, message encryption methods, multiple evasion techniques and diverse attack types. We interchangeably use the terms intruder, attacker, or *DDoS*–client to refer to either the owner of a *DDoS* attack network or the program used to control the network; the terms handler or master [1] designate the nodes at the first level of the *DDoS* attack network that are under direct control of an attacker; similarly, the terms agent, daemon, zombie, or bcast[2] are used to describe entities at the second level of the *DDoS* network. Moreover, *secondary victims* refer to handlers and zombies and *primary victims* portray the direct targets of an attack.

2.1 Phases and Organizational Aspects of *DDoS* Attacks

A *DDoS* entails discovery and penetration of vulnerable systems, implantation of *DDoS* codes, and attack launching [40]. In the course of vulnerable site discovery, computers that harbor well-known defects in network services, vulnerabilities in applications, or mis-configurations in security policies are detected and recorded [10, 40]. Tools such as *nmap*, *nessus* and *sscan* are typically used to speed up the discovery procedure and control the volume of scanning traffic in order to avoid detection by firewalls and IDSs/IPSs [9]. System reconnaissance may occur by probing specific, random, or designated sub-networks as well as by carrying out topological, permutation, or signpost scanning [40]. To go undetected, a cautious intruder may initially penetrate a few vulnerable sites which are used as launching pads for recruiting of secondary victims; this process may be recursively repeated.

During the implantation phase, the attacker installs malicious codes on compromised systems. This is accomplished by transporting codes either from a central storage location or from other compromised machines in the previous phase [28]. The attacker may also remove break-in traces, set up passwords to safeguard compromised systems from further attacks, or install traps to help detect whether administrators of the victim systems are aware of the penetrations. Some *DDoS* tools may rename their executables so that they are perceived as regular processes; for instance in the *Stacheldraht DDoS* network, the handlers and agents are named *kswapd* and *nfsiod* and appear as legitimate processes often escaping the administrator's attention. While at the attack stage, a culprit specifies the type, duration, intensity as well as targets for the attack; such instructions and their parameters are delivered to the handlers of the established *DDoS* network, further relayed to all zombies, and subsequently executed by zombies. Each handler or zombie may only have limited information regarding its siblings in a *DDoS* network. In this respect, even if some nodes of a *DDoS* network are detected and eventually recovered by administrators, other nodes may still continue their malicious work unabated.

In an effective *DDoS* network, each compromised host undertakes a specific role while often forming the popular multiple-level hierarchical network of Figure 1. At the root, attackers, typically through client programs, directly control all handlers to perform various tasks, such as launching attacks, stopping on-going activities and/or collect statistics about the network. *DDoS*-clients may feature specialized interfaces such as *tubby* in *Stachelhardt*, use standard network utilities such as *telnet* and *SSH*, and/or consist of customized code. A number of handlers in *DDoS* networks are able to control multiple agents and monitor specific

[1] a compromised system that can penetrate and manage other machines by installing daemons on the latter

[2] a compromised computer that is implanted with a daemon controlled by *DDoS* masters and waits for commands to launch *DDoS*attacks

TCP/UDP ports for instructions from clients as well as responses from zombies. Agents execute commands on behalf of handlers and generate attack traffic streams that are ultimately dispatched to the primary victims. As Figure 1 indicates, different handlers can share the same subset of agents as is the case with handlers 2 and 3; similarly, an agent may accept instructions from multiple handlers as is the case with agents i and j. Signaling channels are established between clients and handlers as well as between handlers and agents

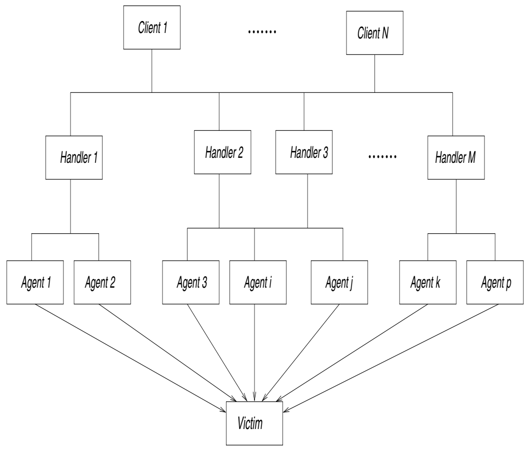

Figure 1: Hierarchical *DDoS* attack network with Handler/Agent paradigm

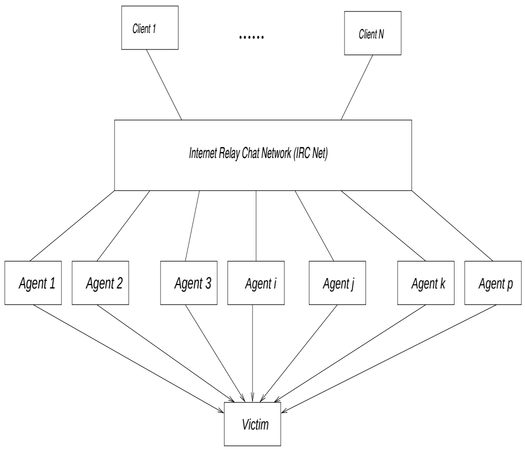

Figure 2: An *IRC*-instigated *DDoS* network

while attack paths are formed between agents and primary victims; in this regard, the two types of channels can follow different communication formats and utilize different transport services, making tracing handlers or clients from primary victims or agents a challenge.

To improve their anonymity, *DDoS* systems employ additional overlay networks such as Internet Relay Chat (IRC) or Peer-to-Peer (P2P) channels between handlers and agents making it harder to be detected by firewalls and IDSs/IPSs [10]. Figure 2 presents an attack network constructed with the help of an IRC network where agents establish outbound connections to the legitimate service port 6667, making it difficult to distinguish communications induced by *DDoS* network from legitimate traffic. To further enhance their robustness, attackers frequently deploy channel-hopping, using any given IRC channel for only short periods of time. Multiple IRC channels can be used to control the *DDoS* network, the discovery of some agents may lead no further than the identification of one or more IRC servers and their channel names used in the *DDoS* network. However, the *DDoS* network as an entity still remains intact. A *DDoS* network can be also formed using a Peer-to-Peer overlay constructed among compromised hosts as Figure 3 depicts. As soon as a peer joins in, it announces its presence and becomes aware of the topology of the entire network via its attacking machine. Information regarding active peers and ongoing activities are continually exchanged among peers and updated throughout the entire network. In this context, the peer-to-peer UDP *DDoS* tool (*PUD*) can connect compromised nodes over UDP on user-specified ports to form a P2P *DDoS* network [51].

Furthermore, *DDoS* networks can be constructed based on specific applications such as Web services and *DNS* systems. To this effect, the *Distributed DNS Flooder (DDNSF)* tool generates a large number of *DNS* queries to overwhelm *DNS* servers [51], while the *Webdevil* can be used to launch *DDoS* attacks by opening and keeping alive multiple HTTP connections to a web server simultaneously, ultimately causing server saturation [51].

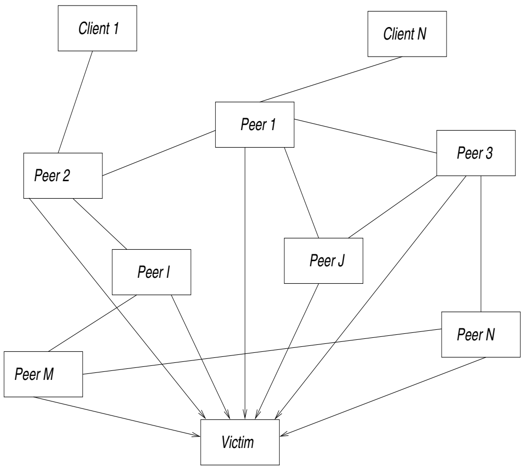

Figure 3: *DDoS* network on P2P network

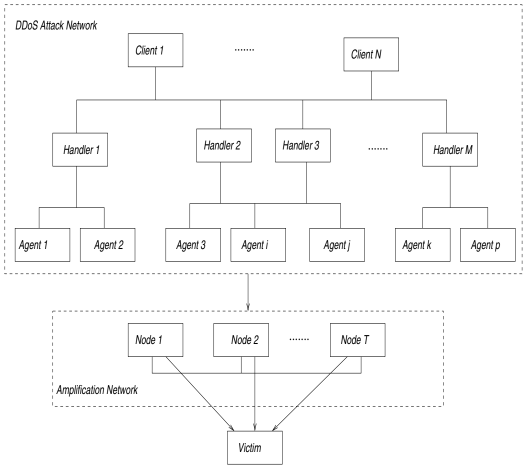

Figure 4: Amplification often used in *DDoS* attacks

2.2 One-Way Communication Channels

In early *DDoS* attack networks such as *TFN*, all messages exchanged between handlers and daemons are camouflaged as *ICMP Echo* reply messages in order to escape detection. Each handler-to-daemon command or reply is assigned a unique identifier which is included in the protocol field *icmp_id* of *ICMP* messages. By exchanging such commands, various tasks related to *DDoS* attacks can be performed. Table 1 describes a sample communication session between a handler and a daemon in a *TFN* network. In the first message, the handler (with IP 192.168.5.143) instructs the daemon (with IP 192.168.5.142) to launch an attack against a victim (with IP 192.168.5.37) with the *flood* method. The attack method or *flood* is indicated by the value of 890 (in decimal) in the protocol field *icmp_id* of the *ICMP* header; while the victim's IP (i.e., 192.168.5.37) is sent to the daemon as *ICMP data*. In its reply, as message 2 shows, the daemon indicates that the command from the handler has been executed successfully as shown in the data section of the *ICMP* message. With message 3, the handler delivers the instruction *stop current flood* with command identifier 567 (in decimal) to the daemon, and the latter eventually answers with message 4 to confirm that the current flooding attack is terminated.

Evidently, communications are bidirectional exposing much information to the more sophisticated security protection systems and making their detection straightforward as their telltales patterns can be mapped out to signatures. In addition, the source addresses of *ICMP* messages in *TFN* are not spoofed rendering the discovery of handlers easy once daemons are identified. It is worth pointing out that messages exchanged between handlers and daemons in *TFN* as shown in Table 1 violate the conventional schema of *ICMP Echo reply* messages, which requires that distinct *icmp_seq*s should be used for different messages and fields *icmp_id* and *icmp_seq* should be echoed back by recipients. Last but not least, commands and their respective parameters in *TFN* appear in clear text, making them easily identifiable by security devices.

TFN2K, the successor of *TFN*, establishes one-way channels between handlers and daemons as Table 2 depicts. Messages are transported from a handler with IP address 192.168.5.143 to a daemon with IP 192.168.5.142. In message 1, the handler instructs the daemon to launch a *UDP* flood attack against the victim located at 192.168.5.37; in message 2, command *stop the current flood* is delivered to the same dae-

#	dir	len	payload	description
protocol: ICMP; handler (denote as H):192.168.5.143; daemon (denote as D): 192.168.5.142				
1	H→D	41	IP header (20 bytes): |45 00 00 29 00 00 40 00 40 01 AE 66 C0 A8 05 8F C0 A8 05 8E| ICMP header (8 bytes): |00 00 C7 54 03 7A 00 00| ICMP data (13 bytes): |31 39 32 2E 31 36 38 2E 35 2E 33 37 00|	ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 192.168.5.143; dst: 192.168.5.142; icmp_type: 0 (Echo reply); icmp_code: 0; icmp_check: 0xC754; icmp_id: 0x037A (dec. 890); icmp_seq: 0; icmp_data: "192.168.5.37";
2	D→H	53	IP Header (20 bytes): |45 00 00 35 00 00 40 00 40 01 AE 5A C0 A8 05 8E C0 A8 05 8F| ICMP header (8 bytes): |00 00 CE CA 00 7B 00 00| ICMP data (25 bytes): |55 44 50 20 66 6C 6F 6F 64 3A 20 31 39 32 2E 31 36 38 2E 35 2E 33 37 0A 00|	ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 192.168.5.142; icmp_type: 0 (echo reply); icmp_code: 0; icmp_check: 0x98B1; icmp_id: 0x007B (dec. 123); icmp_seq: 0; icmp_data: "UDP flood: 192.168.5.37"
3	H→D	30	IP header (20 bytes): |45 00 00 1E 00 00 40 00 40 01 AE 71 C0 A8 05 8F C0 A8 05 8E| ICMP header (8 bytes): |00 00 CF C8 02 37 00 00| ICMP data (2 bytes): |2E 00|	ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 192.168.5.143; dst: 192.168.5.142; icmp_type: 0 (echo reply); icmp_code: 0; icmp_id: 0x0237 (dec. 567); icmp_seq: 0;
4	D→H	50	IP Header (20 bytes): |45 00 00 32 00 00 40 00 40 01 AE 5D C0 A8 05 8E C0 A8 05 8F| ICMP header (8 bytes): |00 00 FF 09 00 7B 00 00| ICMP data (22 bytes): |55 44 50 20 66 6C 6F 6F 64 20 74 65 72 6D 69 6E 61 74 65 64 0A 00|	ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 192.168.5.142; dst: 192.168.5.143 icmp_type: 0; icmp_code: 0; icmp_check: 0xFF09; icmp_id: 0x007B (dec. 123); icmp_seq: 0; icmp_data: "UDP flood terminated".

Table 1: Handler/daemon *TFN* messages with *ICMP*

#	dir	len	payload	description
protocol: ICMP; handler (denote as H):192.168.5.143; daemon (denote as D): 192.168.5.142				
1-20	H→D	70	IP header (20 bytes): |45 00 00 46 F7 8E 00 00 F5 01 DB 59 46 97 E6 00 C0 A8 05 8E| ICMP header (8 bytes): |00 00 A5 96 00 00 4F F7| ICMP data (42 bytes): |6D 56 37 49 63 43 42 76 4A 6D 74 47 57 72 6D 68 31 72 2F 49 4F 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41|	ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 70.151.230.0; dst: 192.168.5.142; icmp_type: 0 (Echo reply); icmp_code: 0; icmp_check: 0xA596; icmp_id: 0x0000; icmp_seq: 0x4FF7; cmd: "tfn -P ICMP -h 192.168.5.142 -c 4 -i 192.168.5.37"; 21 trailing As.
21-40	H→D	55	IP header (20 bytes): |45 00 00 37 6D B1 00 00 FB 01 6D B0 23 2D FB 00 C0 A8 05 8E| ICMP header (8 bytes): |00 00 62 19 37 6E 00 00| ICMP data (27 bytes): |77 77 4C 47 6A 2F 43 7A 2F 36 62 6F 2F 79 6D 4D 34 6B 59 64 75 51 41 41 41 41 41|	ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 35.45.251.0; dst: 192.168.5.142; icmp_type: 0; icmp_code: 0; icmp_check: 0x6219; icmp_id: 0x376E; icmp_seq: 0x0000; cmd: "tfn -P ICMP -h 192.168.5.142 -c 0"; 5 trailing As.
41-60	H→D	59	IP header (20 bytes): |45 00 00 3B 3D D8 00 00 F1 01 C7 BB 84 F7 79 00 C0 A8 05 8E| ICMP header (8 bytes): |00 00 EB 60 00 00 00 00| ICMP data (31 bytes): |4E 30 53 36 6E 4D 57 66 79 39 6D 71 71 4F 61 54 34 65 36 54 43 77 41 41 41 41 41 41 41 41 41|	ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 132.247.121.0; dst: 192.168.5.142; icmp_type: 0; icmp_code: 0; icmp_check: 0xEB60; icmp_id: 0x0000; icmp_seq: 0x0000; cmd: "tfn -P ICMP -h 192.168.5.142 -c 3 -i 1024"; 9 trailing As.

Table 2: Handler/daemon *TFN2K* Messages with *ICMP*

mon. Clearly, in *TFN2K*, commands are no longer transferred in protocol field *icmp_id* of the *ICMP header*; instead, they are embedded in the data section of *ICMP* messages with fields *icmp_id* and *icmp_seq* taking random values (or zero). The source IPs of *ICMP* messages are also randomized (i.e., spoofed), effectively hiding the identity of the handler; for instance, in message 1 of Table 2, the source IP is 70.151.230.0 and then becomes 35.45.251.0 in message 2. The lack of any daemon feedback makes it impossible for a handler to ensure proper command dispatch and execution. To enhance the chances of their success, handlers transport each of their commands multiple times, 20 by default, as Table 2 shows.

The one-way communication option can be also established with the help of TCP/UDP transport. Table 3 shows partial UDP uni-directional traffic emanating from a *TFN2K* handler. In this scenario, instruction for a *UDP attack* against the victim located at 192.168.5.37 is delivered from the handler with IP 192.168.5.143 to the daemon at IP 192.168.5.142. For brevity, we present the UDP header and payload for the first message while UDP headers only for other messages of Table 3 as all messages have the same UDP payload to message 1. Although Table 3 essentially delivers the same command as the first message shown in Table 2, UDP messages in *TFN2K* have spoofed source IP addresses, source ports, and destination ports, implying that the daemon should work in raw mode in order to monitor all incoming UDP traffic. Similar observations can be drawn from TCP-based communication channels. It is finally worth noting that checksums of the

#	dir	len	*spoofed src IP*	*UDP header*	*description*
protocol: UDP; handler (denote as H):192.168.5.143; daemon (denote as D): 192.168.5.142					
1	H→D	70	157.253.87.0	UDP header: |86 9F ED 7C 00 35 81 71| UDP data: |6D 56 37 49 63 43 42 76 4A 6D 74 47 57 72 6D 68 31 72 2F 49 4F 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41|	udp_sp: 34463; udp_dp: 60796; udp_check: 0x8171; cmd: "tfn -P UDP -h 192.168.5.142 -c 4 -i 192.168.5.37"; total 21 'A'.
2	H→D	70	157.253.87.0	|3A 5C EC 32 00 35 CE FE|	udp_sp: 14940; udp_dp: 60466; udp_check: 0xCEFE;
3	H→D	70	157.253.87.0	|F0 6F A8 9A 00 35 5C 83|	udp_sp: 61551; udp_dp: 43162; udp_check: 0x5C83;
4	H→D	70	157.253.87.0	|3B 7E 75 73 00 35 44 9C|	udp_sp: 15230; udp_dp: 30067; udp_check: 0x449C;
5	H→D	70	157.253.87.0	|13 FB 1C 71 00 35 C5 21|	udp_sp: 5115; udp_dp: 7281; udp_check: 0xC521;
6	H→D	70	157.253.87.0	|43 D6 ED C7 00 35 C3 EF|	udp_sp: 17366; udp_dp: 60871; udp_check: 0xC3EF;
7	H→D	70	157.253.87.0	|7E C1 C8 03 00 35 AE C8|	udp_sp: 32449; udp_dp: 51203; udp_check: 0xAEC8;
8	H→D	70	157.253.87.0	|A6 11 14 F2 00 35 3A 8A|	udp_sp: 42513; udp_dp: 5362; udp_check: 0x3A8A;
9	H→D	70	157.253.87.0	|D3 A9 D4 52 00 35 4D 91|	udp_sp: 54185; udp_dp: 54354; udp_check: 0x4D91;
10	H→D	70	157.253.87.0	|84 6D 7B A1 00 35 F5 7E|	udp_sp: 33901; udp_dp: 31649; udp_check: 0xF57E;

Table 3: UDP messages from handler to daemon in *TFN2K*

UDP packets created by *TFN2K* (i.e. field *udp_check*) are incorrectly calculated as the required pseudo-header is not included in the checksum computation. The 12-byte pseudo-header consists of fields *source IP* (4 bytes), *destination IP* (4 bytes), *reserved* (1 byte and should be zero), *protocol* (1 byte) and *total length* (2 bytes). The same design flaw also exists in TCP-based *TFN2K* messages and can be used as a metric to identify such traffic.

2.3 Encryption of Communication Messages

Another enhancement that *TFN2K* maintains over its predecessor is the use of cryptographic methods including the advanced encryption standard (AES), the international data encryption algorithm (IDEA), and variants of CAST algorithms [49]. The obfuscation followed by *TFN2K* initially involves a message encryption stage with the 16-byte block-oriented CAST-256 algorithm (along with the needed padding at the end of the message), followed by an encoding stage using a base-64 scheme so that the output content is in the printable range [A-Z, a-z, 0-9, +/]. The procedure used by *TFN2K* to encrypt and encode messages is presented in Algorithm 1. It can be observed that the size of the encrypted message *elen* generated by the encryption stage is different from the size of the input, *clen*, to the encoder. Given that *plen* represents the length of the original text, clearly $clen > elen$ as $clen = plen + 16$; *elen* can be either *plen* or $plen + (16 - plen\%16)$ depending on whether or not the condition ($plen\%16 = 0$) is satisfied. During the encoding stage, the input is padded with *(clen-elen)* zeros whose content is ultimately turned into "A"s through the 64-base encoding scheme; this yields an artifact at the end of the encoded cipher text that we could exploit to identify *TFN2K* traffic. For example, in message 1 of Table 2 there are 21 trailing characters "A" while in messages 2 and 3 there are 5 and 9 "A"s at the end of the *ICMP* payload. The difference *(clen-elen)* essentially determines the number of the trailing "A"s and takes values in the [1, 16] range. The base-64 encoding essentially transfers every three bytes into sequences of four bytes and at the end of this stage the difference is in the range [1,21]. We can overall compute the number of trailing "A"s based on the size of an encoded and encrypted *TFN2K* message[3]. It is in general futile to attempt to recover original traffic without the encryption keys

[3] We discuss how we take advantage of this in our Algorithm 8

Algorithm 1 Encryption Procedure in the *TFN2K DDoS* tool

Input: vector ($plain$, $plen$) where $plain$ is the plain text to be encrypted, and $plen$ is its length
$elen \leftarrow plen$, which is the real length of cipher text
if ($plen$ is not a multiplier of 16) **then**
    rem = $plen$ mod 16; $elen$ = $plen$ + rem; $plain$ is padded with rem zeros;
end if
$plain$ is divided into 16-byte blocks; $cipher$ is used to store encrypted version of $plain$ and initially set to be empty
for (each 16-byte block B in $plain$) **do**
    encrypt B with CAST-256 algorithm; resulting cipher text is appended to $cipher$;
end for
variable $clen$ is the size of $cipher$ and $clen$ = $plen$ + 16;
if ($clen$ is larger than $elen$) **then**
    rem = $clen$ - $plen$; $cipher$ is padded with rem zeros;
end if
$cipher$ is divided into 3-byte blocks; the last block may contain 0, 1, or 2 bytes; out_encode is set to empty and its length $blen$ is set to zero;
for (each 3-byte block B in $cipher$) **do**
    encode B with base-64 algorithm; resulting block, 4-byte long, is appended to out_encode; $blen$ = $blen$ + 4;
end for
if (the last block contains 1 byte) **then**
    the single byte in the block is encoded into two bytes and appended to out_encode; $blen$ = $blen$ + 2;
else if (the last block contains 2 byte) **then**
    the two bytes in the block is encoded into three bytes and appended to out_encode; $blen$ = $blen$ + 3;
end if
Output: (out_encode, $blen$)

used [1, 29, 49], and thus, it is rather pragmatic to identify traffic based on its message sizes, handshake procedures and unique traffic patterns instead of its syntax.

In a similar fashion, *Stacheldraht* messages are encrypted using the Blowfish secret-key block cipher algorithm that iterates a simple encryption function 16 times with the help of a Feistel network [49]. Using 64 bits block size and 448 bits keys, the algorithm features an expansion phase where a given key is converted into several subkey arrays totaling 4,168 bytes and a data encryption phase where a 16-round Feistel network is carried out [49]. Provided that the only possible way to break Blowfish is a keyspace exhaustive search [49], the only realistic option that remains for traffic identification is to extract specific characteristics and/or patterns of messages and their exchange procedure. In *Stacheldraht*, attacker-handler connections are password protected with attacker commands, handler responses, and passwords being Blowfish-encrypted. The encryption keys are specified when the *DDoS* codes are compiled and may be changed when *DDoS* codes are installed by attackers in compromised machines; this process makes any clear text unavailable and thus renders the search for telltales impossible. Algorithm 2 outlines the *Stacheldraht* encryption procedure for communications between its attackers and handlers and reveals a few artifacts that may help us identify pertinent traffic. First, *Stacheldraht* uses a fixed length of 1024 bytes for its messages and consequently, short messages may be padded with zeros before transmission. In most instances, the encrypted part of the message is typically less than 100 bytes, leaving a very long string of zeros at the end. Second, the handler-employed *echo-back* mechanism for processing the attacker password may also help detect *Stacheldraht* traffic through correlation and/or sequence matching of information flows in both directions. Finally, handler-banner features presented to the attacker can be exploited as well; among the four banner lines, two have identical content. Although the two identical strings "***...****|0A|" are encrypted, encoded, and transferred with different keys at different times, they are obfuscated with the same key within the same connection, clearly offering an opportunity for detection.

Algorithm 2 *Stacheldraht* Handler Encryption Procedure

Input: a TCP connection from the attacker
get password from the attacker; the password is encrypted with Blowfish, encoded with base-64 scheme, and padded to 1024 bytes with zeros; decrypt the received password;
if (entered password is incorrect) **then**
    echo back the incorrect and decrypted password to the attacker; close the connection and exit;
end if
echo back the encrypted and base-64 encoded password to the attacker (padded with zeros to 1024 bytes); present the attacker with the following banner:
"******************************|0A|"
" welcome to stacheldraht |0A|"
"******************************|0A|"
"type .help if you are lame|0A 0A|";
each line of such greeting is encrypted with Blowfish and encoded with base-64 individually, padded to be 1024 bytes with zeros, and flushed to the connection;
for (each command entered by the attacker) **do**
    execute the requested command; the result of the command is encrypted with blowfish and encoded with base-64, padded with zeros to 1024 bytes, and send back to the attacker;
    close the connection and exit if the command is ".quit";
end for

Table 4 shows a segment of traffic between a *Stacheldraht* client and its handler using TCP transport mechanism. Once the attacker-to-handler connection is established –not shown for brevity– the attacker dispatches its Blowfish-encrypted and base-64 scheme encoded password padded with zeros at the end as message 1 depicts. Due to their base-64 encoding, TCP message payloads are in the range [./, 0-9, a-z, A-Z] followed by trailing zeros. Padding is a must for most messages as *Stacheldraht* follows a fixed-size format. In this respect, the length of the encrypted/encoded password in message 1 is 24 bytes requiring 1,000 bytes of zeros for padding. The handler in message 2 echoes back the same encrypted/encoded password to the attacker once authentication process completes. Obviously, the Blowfish-encrypted password and its echo-back instance are identical. In messages 3, 4 and beyond, the handler transfers to the attacker its 4-line

<table>
<tr><th>#</th><th>dir</th><th>IP:TCP:data</th><th>TCP payload</th><th>description</th></tr>
<tr><td colspan="5">protocol: TCP; client (denote as C):192.168.5.143:58712; handler (denote as H): 192.168.5.142:65512</td></tr>
<tr><td>1</td><td>C→H</td><td>20:32:1024</td><td>|5A 54 74 72 4B 30 30 63 30 31 61 30 31 66
58 30 69 31 62 73 65 46 66 30 00 00 ...|</td><td>attacker enters password encrypted with
blowfish; length: 24; padded with (1024 - 24) |00|;</td></tr>
<tr><td>2</td><td>H→C</td><td>20:32:1024</td><td>|5A 54 74 72 4B 30 30 63 30 31 61 30 31 66
58 30 69 31 62 73 65 46 66 30 00 00 ... 00|</td><td>handler echoes back the password;
encrypted with blowfish; padded with (1024 - 24) zeros;</td></tr>
<tr><td>3</td><td>H→C</td><td>20:32:1024</td><td>|44 64 4F 51 48 31 32 4B 59 50 4D 30
44 64 4F 51 48 31 32 4B 59 50 4D 30
44 64 4F 51 48 31 32 4B 59 50 4D 30
37 71 65 7A 46 2E 74 4A 44 6D 49 2F 00 00 ...|</td><td>handler presents banner to attacker;
first, string (encrypted with blowfish):
"******************************|0A|";
encrypted len: 48 bytes; padding (1024 - 48) zeros;</td></tr>
<tr><td>4</td><td>H→C</td><td>20:32:1448</td><td>|6F 73 4F 45 52 31 4A 51 41 77 6E 2F
4D 35 63 74 71 2E 69 69 50 43 37 30
71 41 78 37 73 2E 45 78 62 4A 4C 2E
6B 65 2E 4E 6A 31 4A 6B 4D 46 56 2F 00 00 ...|
|44 64 4F 51 48 31 32 4B 59 50 4D 30
44 64 4F 51 48 31 32 4B 59 50 4D 30
44 64 4F 51 48 31 32 4B 59 50 4D 30
37 71 65 7A 46 2E 74 4A 44 6D 49 2F 00 ... 00|</td><td>then, string (encrypted with blowfish):
" welcome to stacheldraht |0A|";
encrypted len: 48 bytes; padding
(1024 - 48) bytes zeros; string (encrypted with blowfish):
"******************************|0A|"
encrypted len: 48 bytes; pad (1024 - 48) zeros;
cipher text is the same as that in previous
message; padded data continue to next msg;</td></tr>
<tr><td>5</td><td>H→C</td><td>1500</td><td>|00 00 ...|
|73 72 65 6C 79 2E 34 2F 49 78 73 30
38 39 72 72 65 2F 4A 54 39 69 4B 2F
76 33 30 2E 73 30 75 56 4B 46 54 2E
58 46 66 53 58 2F 66 4D 47 50 44 31 00 00 ...|</td><td>blowfish encrypted string:
"type .help if you are lame|0A 0A|";</td></tr>
</table>

Table 4: Messages between client and handler with TCP in *Stacheldraht*

greeting message in encrypted/encoded form which gets displayed to the attacker by the client program after decryption/decoding. Clearly, the TCP transport service may violate application message boundaries by packing multiple *Stacheldraht* messages into a single TCP packet, even though such messages are handed to the TCP/IP stack separately. For instance, the message-line "*welcome to* ..." and part of the following line "*** ..." are merged into a single TCP packet as message 4 shows; the remaining of the "*** ..." appears in the subsequent TCP packet (not shown). Hence, boundary inconsistencies between TCP packets and application messages lead to the fact that TCP stream reassembly is required to identify boundaries of application messages and then possibly search for patterns.

2.4 Evasive Techniques of *DDoS* Attacks

Decoy packets and/or use of dynamic ports are frequently employed by *DDoS* attackers to avoid detection by firewalls and IDSs/IPSs. In this regard, *TFN2K* is capable of transmitting an arbitrary number of decoy packets for every "real" packet obscuring the actual *DDoS* attack elements, such as launching points and attack targets. Table 5 presents a segment of traffic generated by TCP-*TFN2K* where for every real packet a decoy is created. Here, a handler dispatches the commands *start UDP flood attack* and *stop current flood attack* through messages 1 and 3 while messages 2 and 4 are the respective decoys. Nearly all TCP header fields of

<table>
<tr><th>#</th><th>len</th><th>payload</th><th>description</th></tr>
<tr><td colspan="4">protocol: TCP; handler (denote as H):192.168.5.143; daemon (denote as D): 192.168.5.142; direction: H→D</td></tr>
<tr><td>1</td><td>82</td><td>IP header (20 bytes): |45 00 00 52 65 F3 00 00
CC 06 86 55 3F 26 FD 00 C0 A8 05 8E|
TCP header (20 bytes): |33 1D 35 F1 00 00 00 00
00 0C B6 F8 00 12 7D 75 FC D5 00 00|
TCP data (42 bytes): |4B 4B 39 6B 68 6D 68 71 79 66 33 42
4A 64 76 2B 5A 52 47 6B 74 77 41 41 41 41 41 ...</td><td>ip_tos: 0; ip_id: 0; ip_ttl: 64;
src: 63.38.253.0; dst: 192.168.5.142;
tcp_sp: 0x331D; tcp_dp: 0x35F1; tcp_seq: 0;
tcp_ack: 0x000CB6F8; tcp_off: 0; tcp_flags: ACK|SYN;
tcp_win: 0x7D75; tcp_check: 0xFCD5;</td></tr>
<tr><td>2</td><td>82</td><td>IP header (20 bytes): |45 00 00 52 52 8D 00 00 D3 06 B4 51
3F 26 FD 00 0D A0 97 00|
TCP header (20 bytes): |F5 A6 27 6E 00 0B B3 D0 00 00 00 00
00 12 37 7A 91 F3 00 00| TCP data (42 bytes): (same as # 1)</td><td>ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 63.38.253.0;
dst: 13.160.151.0; tcp_sp: 0xF5A6; tcp_dp: 0x276E;
tcp_seq: 0x000BB3D0; tcp_off: 0; tcp_win: 0x377A;
tcp_ack: 0; tcp_check: 0x91F3; tcp_flags: ACK|SYN;</td></tr>
<tr><td>3</td><td>67</td><td>IP header (20 bytes): |45 00 00 43 6C 5C 00 00 E1 06 84 60
0A C2 18 00 C0 A8 05 8E|
TCP header (20 bytes): |FF 29 78 EF 00 E7 F9 51 00 07 06 29
00 10 C1 52 87 1B 00 00|; TCP data (27 bytes): |45 38 74 39 61 42
76 71 6C 54 6D 4C 78 38 78 6F 71 52 48 6A 66 51 41 41 41 41 41</td><td>ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 10.194.24.0;
dst: 192.168.5.142; tcp_sp: 0xFF29; tcp_dp: 0x78EF;
tcp_seq: 0x00E7F951; tcp_off: 0;
tcp_ack: 0x00070629; tcp_flags: ACK;
tcp_win: 0xC152; tcp_check: 0x871B;</td></tr>
<tr><td>4</td><td>67</td><td>IP header (20 bytes): |45 00 00 43 A6 62 00 00 D0 06 DF 54
0A C2 18 00 45 3B FD 00|
TCP header (20 bytes): |7A 8E 00 2E 00 00 00 00 00 00 00 00
00 10 00 00 46 34 00 00| TCP data (27 bytes): (same as # 3)</td><td>ip_tos: 0; ip_id: 0; ip_ttl: 64; src: 10.194.24.0;
dst: 69.59.253.0; tcp_sp: 0x7A8E; tcp_dp: 0x002E;
tcp_seq: 0; tcp_ack: 0; tcp_off: 0; tcp_flags: ACK;
tcp_win: 0; tcp_check: 0x4634;</td></tr>
</table>

Table 5: *TFN2K* handler/daemon TCP messages

handler-originated messages are spoofed with random or zero values; as expected, the source IP addresses in IP headers are also fake. While the destination IP addresses of "real" messages are set to be the true daemon IP values, respective addresses are spoofed in decoy packets. More specifically, the destination of the command *start UDP flood attack* in message 1 displays the true 192.168.5.142 daemon IP; this is not the case with message 2 where the destination IP address has been randomly assigned the value 13.160.151.0. It is worth noting that the field *tcp_off* of the TCP header in all messages is zero; in regular circumstances, this field indicates the starting point of TCP data and should be at least five reflecting the minimum size of a legitimate TCP header without any TCP option (i.e., 20 bytes, or 5 32-bits). This abnormality may create problems to some network devices if they fail to handle such unexpected messages, or may escape from the detection by security systems if the latter do not perform protocol anomaly analysis. The values of the field *tcp_check* in TCP headers are also incorrectly calculated since in all *TFN2K*-generated TCP messages the required pseudo-header is not included in the checksum calculation. It is finally noteworthy that the content of decoy packets is identical to the real packets; this can be readily identified, should a protection system features deep-inspection or full-content scanning.

The use of dynamic ports for both handlers and agents is another common technique to evade detection. For instance, by default, *Mstream* handlers listen to TCP port 6723 for attacker/client commands and they monitor UDP port 9325 for daemon-originating information; changing these ports is rather straightforward. Similar observations can be drawn for agents in *Mstream* and handlers/agents in other *DDoS* attack tools. Although IP addresses are generally randomized, (i.e., spoofed), they can take any values specified by attackers in order to elude detection by ingress/egress filtering mechanisms[4].

In the course of an attack, agents dispatch streams of packets to primary victims in either constant or variable rate. After the onset of an attack, agents frequently generate packets as fast as their resources permit. However, abrupt increases in traffic volume can easily raise suspicion. By adjusting attack rates of individual agents so that only small traffic volumes are generated, detection by security mechanisms may be avoided. Even under such light volumes, the resulting traffic may be intensive enough to bring a victim down if the number of agents is very large. In addition, *DDoS* attacks vary values in protocol header fields to evade fixed traffic patterns whose signatures can be detected by security devices. The use of different agents in different attacks reduces the probability for identification as well. *DDoS* attack tools also use the self-updating characteristic in order to change their communication patterns and enhance their functionality; to this effect, the *Stacheldraht* command *.distro* can be used by an attacker to instruct and coordinate all handlers and agents to install new versions of their code. Similarly in *TFN2K*, the same objective is attained through the use of command *remote command execution* (i.e., command code 10 of Table 14) that allows the execution of arbitrary shell commands on all *DDoS* entities.

2.5 Diverse Types of *DDoS* Attacks

The UDP flooding attacks exploit the fact that for every incoming UDP packet, the recipient sends back an ICMP destination unreachable message if the destination port is closed; otherwise, wasteful processing of junk-packets occurs. In *TCP SYN* flood attacks, steady bogus connection requests fill up TCP connection tables of victim systems; should victims attempt to send *TCP-RST*-packets to connection initiators that have provided spoofed addresses, further network congestion is generated. The timeout mechanism associated with a pending connection proves ineffective if an attacker continues to generate IP-spoofed packets faster than the rate with which the victim's pending connections expire. The *ICMP echo request/reply* attacks dispatch requests to specified targets with fake source IP addresses, forcing the victims to generate an equal number of replies. Various *Targa* attacks create packets with malformed or abnormal values at different protocol fields, transmit them to specified targets, and in this manner cause the victims to crash, freeze, or manifest unexpected behavior. In amplifier or reflector attacks such as those instigated by *Smurf* and *Fraggle*, traffic with the primary victim IP as its source is created and transferred to networks supporting direct broadcast; each host of the networks then generates replies to the primary victim congesting the network [40]. Figure 4 shows a *Smurf* attack on a hierarchical *DDoS* network; every host creates replies to messages originated in the *DDoS* network but disguised as coming from the primary victim. With every packet of the attack being repeated by every host, the amplifying effect may become grave. Clearly, such

[4]For example in *TFN2K*, this can be done for packets between handlers and agents if the option *-S* is specified when these components are activated.

IP-direct-broadcast packets should be blocked at LAN borders.

Table 6 presents *TFN2K*-daemon generated traffic using a mixed attack strategy where packets are created based on ICMP, TCP, and UDP protocols with ratio 1:1:1. A number of unique artifacts emerge: firstly, the IP header field *ip_flags* of every attack packet is set to zero regardless of the transport protocol used; this implies that the *don't fragment* bit is not set by *TFN2K*. Secondly, the *time-to-live (ip_ttl)* field of ICMP packets has zero value; this causes the packet to be dropped by any router along the attack path if agents are not co-located in the subnet of victims. Thirdly, all TCP-generated attack packets feature zero values in their *tcp_off* fields rendering them malformed. Finally, checksums for TCP/UDP packets are incorrect as

#	pro	len	payload	description
daemon (denote as D):192.168.5.142; victim (denote as V): 192.168.5.37				
1	ICMP	92	IP header (20 bytes): |45 00 00 5C 28 09 00 00 00 01 FD D0 CE FA 00 00 C0 A8 05 25| ICMP header (8 bytes): |08 00 F7 FF 00 00 00 00| ICMP data (64 bytes): |00 00 ... 00|	ip_flags: 0; ip_ttl: 0; ip_src: 206.250.0.0; ip_dst: 192.168.5.37; icmp_type: 8 (Echo request); icmp_code: 0; icmp_check: 0xF7FF; icmp_id: 0; icmp_seq: 0;
2	TCP	40	IP Header (20 bytes): |45 00 00 28 AB 69 00 00 E8 06 F8 21 1A 77 4F 00 C0 A8 05 25| TCP header (20 bytes): |69 D8 3D 55 00 0A DB 7F 12 3B 00 00 00 22 15 9D 26 85 1F 13|	ip_flags: 0; ip_ttl: 232; ip_src: 26.119.79.0; ip_dst: 192.168.5.37; tcp_sp: 0x69D8; tcp_dp: 0x3D55; tcp_seq: 0x000ADB7F; tcp_ack: 0x123B0000; tcp_off: 0; tcp_flags: 0x22;
3	UDP	29	IP header (20 bytes): |45 00 00 1D 60 18 00 00 CE 11 F1 42 D1 A7 04 00 C0 A8 05 25| UDP header (8 bytes): |FF FE 00 02 00 09 FF F5| UDP data (1 bytes): |00|	ip_flags: 0; ip_ttl: 206; ip_src: 209.167.4.0; ip_dst: 192.168.5.37; udp_sp: 0xFFFE; udp_dp: 2; udp_len: 9; udp_check: 0xFFF5; udp_data: |00|.

Table 6: Mixed attack created by *TFN2K DDoS* attack tool

pseudo-headers are not included in their computation.

By examining both packet header and payload and taking into account the targeted protocols and applications at a site, we may be able to derive specific characteristics of an attack. Table 7 shows *TFN2K*-agent-generated traffic using *Targa* attack type. All packets have zero-value in their *time-to-live (i.e., ip_ttl)* field, some have invalid values in their *protocol* field (e.g., 0x94 is an invalid protocol identifier in message 6), and others may show random values in the *ip_flags* field (i.e., 0x2 in message 4). In addition, a number of packets may have non-zero values in their *fragment_offset* field as message 5 shows. The above discrepancies render the packets malformed and readily identifiable through protocol analysis. As *TFN2K* attack packets do not conform with ICMP, TCP and UDP specifications, values in some protocol fields are abnormal and all TCP/UDP packets have incorrect checksums. Last but not least, a number of techniques used in traditional denial-of-service attacks can be used in *DDoS* attacks as well including the *land, teardrop* and *ping-of-death* methods. In the *land* attack, the victim is bombarded by packets having identical source and destination IPs. The *teardrop* attack exploits known weaknesses in the IP defragmentation of the TCP/IP implementation; by creating a series of IP fragments with overlapping offsets, *teardrop* causes a victim to crash if it is unable to properly handle this overlapping problem. The *ping-of-death* crafts fragmented ICMP messages larger than the allowed maximum IP frame size of 65,536 bytes, causing some systems to either freeze or crash.

3 A Framework for Containing *DDoS* Attacks

To address the aforementioned deficiencies of available systems, we proposed an extensible framework termed *DDoS Container*, which functions in "in-line" manner and employs network-based detection/prevention methods to reliably identify and manipulate *DDoS* traffic in real-time. Our framework

#	*pro*	*len*	*payload*	*description*
daemon (denote as D):192.168.5.142; victim (denote as V): 192.168.5.37				
1	IP	449	IP header (20 bytes): |45 00 01 C1 0F 23 00 00 00 FF D2 4B 53 02 BE 00 C0 A8 05 25| IP data (429 bytes): |D1 0D 6F 33 65 0E 4D cE 9C 14 ...|	ip_flags: 0x0; ip_ttl: 0; proto: 0xFF; ip_src: 83.2.190.0; ip_dst: 192.168.5.37;
2	IGMP	430	IP header (20 bytes): |45 00 01 AE 72 67 00 00 00 02 B4 80 E6 98 E6 00 C0 A8 05 25| IGMP payload (410 bytes): |EF 4F 5D F7 76 40 E4 31 ...|	ip_flags: 0; ip_ttl: 0; proto: 0x02 (IGMP) ip_src: 230.152.230.0; ip_dst: 192.168.5.37;
3	IDP	228	IP header (20 bytes): |45 00 00 E4 7F 33 00 A3 00 16 BD 65 C5 FA F1 00 C0 A8 05 25| IDP payload (208 bytes): |A8 13 D4 E2 C4 46 D5 F5 ...|	ip_flags: 0; offset: 1304; ip_ttl: 0; proto: 0x16; ip_src: 197.250.241.0; ip_dst: 192.168.5.37;
4	ICMP	286	IP header (20 bytes): |45 00 01 1E 9C 0E 20 00 00 01 98 09 7C FA 23 00 C0 A8 05 25| ICMP header (8 bytes): |8A 76 4F D1 E2 FE 7B 09| ICMP data (258 bytes): |23 0D 8D 0C 46 3D 03 82 94 24 ...|	ip_flags: 0x2; ip_ttl: 0; proto: 0x1 (ICMP); ip_src: 124.250.35.0; ip_dst: 192.168.5.37; icmp_type: 138; icmp_code: 118; icmp_check: 0x4FD1; random content;
5	TCP	158	IP header (20 bytes): |45 00 00 9E D9 C5 00 01 00 06 15 27 97 9F 6E 00 C0 A8 05 25| IP payload (138 bytes): |D7 16 95 6F BA 83 38 57 ...|	ip_flags: 0; ip_frag: 8; ip_ttl: 0; ip_src: 151.159.110.0; ip_dst: 192.168.5.37; random content;
6	IP	194	IP header (20 bytes): |45 00 00 C2 2B E7 00 00 00 94 1C 5B 2F 99 7C 00 C0 A8 05 25| IP payload (174 bytes): |7F 46 C6 09 8E 73 A3 35 ...|	ip_flags: 0; ip_frag: 0; ip_ttl: 0; proto: 0x94; ip_src: 47.153.124.0; ip_dst: 192.168.5.37; random content;

Table 7: *Targa* attack created by *TFN2K DDoS* attack tool

monitors the progress of all connections initiated by either normal applications or *DDoS* tools, conducts data correlation among different sessions or messages in the same traffic, and performs stateful and layer-7 inspection. For any identified *DDoS* session, multiple action options can be specified to manipulate it, such as packet dropping, session termination, or connection blocking. In this section, we outline the proposed extensible framework and discuss its components.

3.1 The Architecture of the Proposed *DDoS Container*

The main transport mechanisms used to communicate among clients, handlers, and agents in a *DDoS* network are TCP, UDP and ICMP. In the context of our *DDoS Container* such traffic has to be uniformly represented so that we can effectively process packets and/or message streams established in various *DDoS* sessions. We represent TCP, UDP, and ICMP connections as follows:

- TCP sessions are delimited by their distinct connection and disconnection phases. The connection involves a three-way handshake procedure in which a client initiates a connection with TCP-SYN packet, the recipient or server responds with a TCP-SYN-ACK packet that finally incurs a TCP-ACK packet from the initiating client. The disconnection procedure typically involves a four-message exchange with each side dispatching a TCP-FIN packet and corresponding acknowledgments for receipts of the other end's TCP-FIN message[15]. Therefore, a TCP connection can be uniquely identified by the tuple: *<client-IP,client-port,server-IP,server-port,TCP>*.
- UDP sessions can be similarly identified with this five-element tuple *<client-IP,client-port,server-IP,server-port,UDP>*. However, there is no specific connection or disconnection procedure for a UDP session, we use a timer to control the lifetime of a UDP session.
- ICMP sessions also comply with the above format, should we replace the elements of *client-port* and *server-port* with the concatenation of fields *icmp_type* and *icmp_code* and field *icmp_id* respectively. The tuple now has the format: *<client-IP,icmp_type+icmp_code,server-IP, icmp_id,ICMP>*.

TCP/UDP sessions establish bi-directional data streams between clients and servers while ICMP sessions create only uni-directional streams from the originator (client) to recipient (server). Each such data stream 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 port number for the source of the stream and IP_d and $PORT_d$ are their destination counterparts. Messages exchanged among *DDoS* clients, handlers, and agents are generated according to their own syntax rules and specific semantics; the three underlying transport services may destroy application message boundaries and thus demarcation discrepancies between *DDoS* message borders and transport packet inevitably occur. In addition, TCP may deliver duplicate or out-of-order packets, which are reassembled to obtain the original data stream at the destination. In general, we expect that multiple concurrent sessions exist among the elements of *DDoS* attacks at any specific time and coexist with sessions of regular applications.

The "in-line" operation mode for our *DDoS Container* renders it an indispensable component of the network infrastructure that effectively intercepts and inspects packets. Our framework dissects every packet according to the TCP/IP suite to locate anomalous or evasive traffic based on manipulation of protocol fields. Our *DDoS Container* keeps state records for all established connections and this information remains accessible beyond the lifetime of a session for correlation analyses that lead to *DDoS* traffic detection. The ability to track the state of all active sessions facilitates stateful inspection, intra-session data fusion, and inter-session correlation for *DDoS*-traffic. By correlating data streams within a single session, we determine the success of an attacker's operation. With the help of information from different sessions, we can associate an attacker's control and data channels.

To remap a sequence of packets to possible *DDoS* messages, our *Container* re-assembles (or sequences) all stored packets in their correct orders and interprets the resulting aggregations based on syntax, characteristics, and semantics of *DDoS* systems. The message sequencing helps restore message demarcations imposed by the application layer; without it, *DDoS* sessions may go undetected if their messages span over multiple packets or multiple messages are packed in a single packet. The large number of existing *DDoS* systems and their variety of underlying protocols essentially leads the design of our *DDoS Container* in using multiple techniques to classify data streams that include fine-tuned signatures, traffic correlation, protocol dissection, anomaly analysis, and stateful inspections. To this effect and as Figure 5 shows, our *DDoS Container* integrates a number of needed modules including TCP/IP *Protocol Decoder, Behavior Police, Session Correlator, Message Sequencer, Traffic Distinguisher*, and *Traffic Arbitrator*.

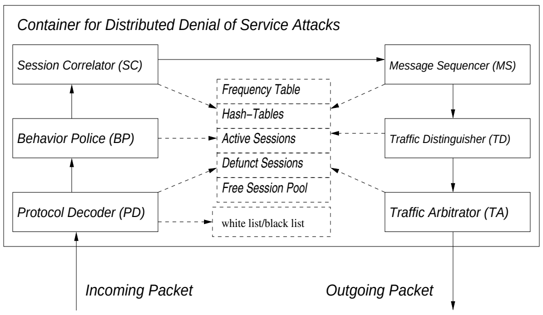

Figure 5: *DDoS Container* Architecture

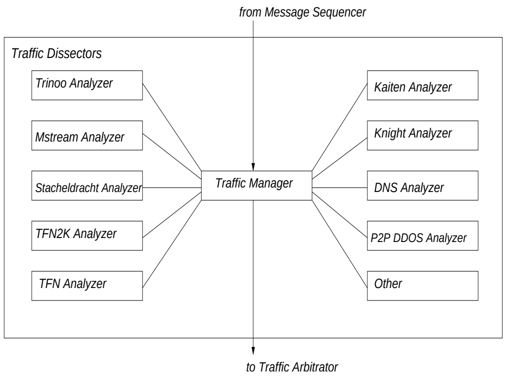

Figure 6: *Traffic Distinguisher (TD)* Components

When a packet P arrives at our *DDoS Container*, the *Protocol Decoder (PD)* initially dissects it according to the TCP/IP-suite, and violations of the standard specifications are identified leading to the immediate dropping of P. To detect traffic generated by TCP/UDP/ICMP flooding attacks, the *Behavior Police (BP)* attempts to correlate P with different existing traffic flows and evaluates the behavior of the aggregated data streams. For instance, if the total number of packets, including P, to the same target machine exceeds a specified threshold (e.g., 100 pps), P is flagged as part of an ongoing flooding attack. Once processed by *BP*, packet P is then forwarded to module *Session Correlator (SC)* which determines the session S for packet P; if no such session exists, a new one is created. Based on S and correlation results with other sessions, the *DDoS Container* may be able to determine whether P belongs to a *DDoS* session. Next, P is handed to module *Message Sequencer (MS)* along with its session information S; here, P is re-assembled with other packets of the same stream to form a sequence of packets or a *super*-packet. This super-packet is surrendered to *Traffic Distinguisher (TD)* that essentially certifies that the application type of the session is either regular or generated by an *DDoS* tool. Finally, the *Traffic Arbitrator (TA)* module stores P, updates information of S according to P and may create alerts if S is part of a *DDoS* session. It is worth pointing out that our *DDoS Container* offers a fast processing option for certain origins of traffic which we certainly know are either trusted or untrusted. Trusted origins are specified in a *white-list* and their traffic is forwarded without any inspection as Figure 5 shows. Untrusted points make up a *black-list* and traffic originated from such nodes is blocked. If our *DDoS Container* cannot keep up with the ongoing network traffic, it can be configured to use either a “fail-open” or “fail-close” policy; the former forwards packets without any inspection while the latter simply drops all packets.

3.2 *Protocol Decoder (PD)*

It is often the case that *DDoS* tools use raw sockets to fan out packets as quickly as possible, bypassing normal traffic procedures including retransmission, finite-state-machine inspections, and congestion control. This essentially means that attackers should fill every protocol field for their crafted packets including IP, TCP, UDP and ICMP headers. To speed matters up, *DDoS* tools usually insert constant values for the same protocol fields for all packets generating artifacts in traffic flows. For instance, TCP-flood attack packets generated by *Stacheldraht* always assume the same sequence number (i.e., 0x28374839) and bits *don’t fragment* and *more fragment* in field *ip_flags* of their IP headers are not set in clear violation of the TCP/IP specifications. The control traffic among *DDoS* attack entities shows similar characteristics. For example messages sent from clients to handlers over TCP in *TFN2K* always have value of zero in their *tcp_off* field creating a zero-size TCP header, an evident sign of malformed packet. In addition, the incorrect checksums for all *TFN2K* TCP/UDP packets reveal inconsistencies with the TCP/IP standards.

Algorithm 3 shows the main functions of the *PD* module whose objective is to decode every incoming packet P according to TCP/IP specifications and detect existing anomalies. Initially, *PD* decodes the IP header of P, inspects its checksum, and drops P if its checksum is in error. Then, *PD* decodes P’s transport protocol header based on its field *protocol* (i.e., ICMP, TCP, or UDP). Packet P may be dropped according to the configuration by the system administrator if it contains any protocol anomaly. Finally, *PD* invokes all relevant *DDoS* tool analyzers that identify *DDoS* session exclusively based on irregularities presented in protocol fields of P. To speed up this examination, *DDoS* analyzers register with *PD* their handlers used to perform protocol anomaly verification. If P is detected as part of *DDoS* traffic streams, the *PD* bypasses all other components and hands P to *Traffic Arbitrator* for alerting, blocking and/or disconnection action.

Algorithm 3 Skeleton for the *TCP/IP Protocol Decoder (PD)*

$P \leftarrow$ newly arrival packet
IP-checksum-org $\leftarrow$ original IP checksum of P; *IP-checksum-new* $\leftarrow$ computed checksum of P based on IP header of P;
drop P and exit if *IP-checksum-new* is not the same as *IP-checksum-new*;
decode other IP protocol fields of P, including *time-to-live*, *prot*, *ip_flags*, *ip_id*, *ip_off*, and option;
if (*prot* is TCP) **then**
    tcp-checksum-org $\leftarrow$ original TCP checksum of P; *tcp-checksum-new* $\leftarrow$ computed checksum of P based on its TCP header and pseudo-header; drop P and exit if *tcp-checksum-new* is not the same as *tcp-checksum-new*;
    analyze other TCP protocol fields of P, including *tcp_sp*, *tcp_dp*, *tcp_seq*, and *tcp_ack*.
else if (*prot* is UDP) **then**
    check its checksum just like TCP; analyze other UDP protocol fields of P, including *udp_sp*, *udp_dp*, *udp_len*.
else if (*prot* is ICMP) **then**
    check its checksum just like TCP; analyze other ICMP protocol fields of P, including *icmp_type*, *icmp_code*, *icmp_id*, and *icmp_seq*.
end if
for (each registered analyzer A for a *DDoS* in the framework) **do**
    invoke the protocol anomaly handler of A with P; if the return code for invocation of A is negative drop P and exit;
end for
P is handed over to *Behavior Police (BP)*

3.3 *Behavior Police (BP)*

The objective of this module is to identify illegitimate activities, especially various flooding attacks, using heuristics such as thresholding, statistics as well as rules and profiling. To accomplish its overseeing work, *BP* maintains a *FREQUENCY_TABLE* that records occurrences and timestamps of traffic groups designated with the help of flow templates; these templates specify the protocol fields to be checked, including protocol types, source and destination IPs, as well as source and destination ports. In addition, flow templates can define relationships among different protocol fields or packets in the same traffic group. For instance by using the *tcp_seq* field, we can specify that all packets in the traffic group should have the same or monotonically increasing sequence. Similarly, the source *tcp_sp* and destination *tcp_dp* TCP ports can be used to group packets that satisfy the condition *tcp_sp*+*tcp_dp*=constant.

By defining a flow template as "all packets with the same destination IP", we can identify single-target flood attacks. For the traffic of Table 6, *BP* creates a *FREQUENCY_TABLE* entry for traffic group "destination IP=192.168.5.37" also indicating 3 observed packets for the flow in question. A similar traffic group can be designated with template "destination IP=192.168.5.37" for the traffic of Table 7 and the corresponding *FREQUENCY_TABLE* entry maintains 6 observed packets. If both the above streams occur simultaneously, only one traffic group with template "destination IP address = 192.168.5.37" is formed with 9 encountered packets. In order to reduce the memory consumption by the *FREQUENCY_TABLE*, we use the sliding window mechanism for each traffic group; here, only those data within the current time window –typically one second– are stored.

Using the *FREQUENCY_TABLE*, *BP* computes various metrics and compares them against administrator-set thresholds to detect traffic anomalies. One such metric is the flow intensity and is defined as the number of observed packets per second (pps) within a specific traffic group. Should *BP* be monitoring the *Trinoo* UDP-flood attack of Table 8 using the template "destination IP=192.168.5.37" and a threshold of 1,000 pps, it can compute the flow intensity by recording the timestamp of every arriving packet; an intensity value of 26,820 UDP pps would certainly point to an ongoing attack. Similarly, *BP* may monitor traffic originating from specific IP addresses, ports or services. By defining the flow template

#	timestamp	sport	dport	size	payload	description
protocol: UDP; daemon (D): 192.168.5.141; victim (V): 192.168.5.37;						
1	38.280830	32770	41577	4	|00 00 00 00|	begin the UDP flood by *Trinoo*
2	38.280834	32770	40361	4	|00 00 00 00|	same src port but different dst port
3	38.280839	32770	52178	4	|00 00 00 00|	
4	38.280980	32770	41786	4	|00 00 00 00|	attack rate is not constant
5	38.281024	32770	6756	4	|00 00 00 00|	packet size always the same
6	38.281068	32770	11412	4	|00 00 00 00|	payload is set to be zero
7	38.281111	32770	50010	4	|00 00 00 00|	
8	38.281155	32770	10055	4	|00 00 00 00|	dst port is random
9	38.281199	32770	10262	4	|00 00 00 00|	
10	38.281243	32770	61703	4	|00 00 00 00|	
11	38.281288	32770	4461	4	|00 00 00 00|	
12	38.281298	32770	14226	4	|00 00 00 00|	
13	38.281302	32770	35838	4	|00 00 00 00|	
14	38.281352	32770	6760	4	|00 00 00 00|	attack intensity: 14/(38.281352 - 38.280830) = 26820 (pps)

Table 8: UDP flooding created by *Trinoo*

"traffic from the same IP and the same UDP-port", a traffic group for "192.168.5.141:32770" can be created for the traffic of Table 8 and the corresponding intensity rate can be computed.

3.4 *Session Correlator (SC)*

The purpose of this module is to maintain *DDoS Container* session-wide information; each connection is represented with the *session* structure depicted in Table 9. As mentioned in Section 3.1, a session is uniquely identified by the first five fields of Table 9 or tuple <*SIP, SPORT, DIP, DPORT, PROTO*>. *TYPE* indicates the session application type such as *Trinoo*, *Mstream*, or *TFN2K*; *TYPE* is set to *bypass* if the application type has not been determined after inspecting a certain amount of traffic or the session belong to a regular application. *CONFIRM* indicates whether *TYPE* is determined using correlation of both streams of a single session, through association with other active or defunct sessions, or simply set using different messages in uni-directional traffic. Correlation of different messages within the same data stream is valuable in situations where asymmetric routing occurs and the two traffic streams in a connection take different network paths with only one of the paths being visible to our *DDoS Container*. *START* and *LAST* represent a session's creation and most recent access time. The messages originated from the initiator of the session and the corresponding ones from the recipient are stored in *CLIENT* and *SERVER* respectively and the utility of these two lists is discussed in Section 3.5.

To offer efficient operations on sessions and facilitate intra- and inter-session correlations, we employ a two-staged approach in organizing pertinent data. We first use a hash function $H(IP_s, PORT_s, IP_d, PORT_d, PROTO)$[5] to "scatter" sessions in space. Next, a splay tree T anchored off each entry of the hash table helps organize all sessions that present the same hash value. Each node of T represents a session as described by *session* (of Table 9) and the tuple $< IP_s, PORT_s, IP_d, PORT_d, PROTO>$ acts as the key for accessing T. The advantage of splay trees is that more frequently accessed items move closer to the root amortizing future look-up costs [52]. The call *session-find*(P) performs hash table lookup and followed by retrieval of the corresponding splay tree and information about the session in which P belongs to. Information about recently terminated or defunct sessions are helpful to determine the application

[5] our $H()$ is based on $h=((IP_s xor PORT_s) xor (IP_d xor PORT_d) xor PROTO)$ and is defined in a similar to Linux Kernel manner [36] as $H()=(h xor (h >> 16) xor (h >> 8)) mod (h\_size-1)$ where ">>" is the right-shift operation, and *h_size* is the size of the hash table.

field name	*size bytes*	*description*
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 (TCP or UDP) of the host with IP address *SIP*; or (icmp_type | icmp_code) for ICMP.
DPORT	4	port number of the host with IP address *DIP*; or (icmp_id) for ICMP.
PROTO	1	protocol utilized by the session (TCP, UDP, or ICMP)
TYPE	4	identify traffic type, such as *Trinoo, Mstream, Stacheldraht*; can be "bypass"
CONFIRM	4	*TYPE* is drawn from uni-or bi-directional streams, intra- or inter-session correlations
START	4	creation time of the session
LAST	4	last active time of the session in either direction (i.e., transmission of packet)
SERVER	4	pointer to server *stream* data structure
CLIENT	4	pointer to client *stream* data structure

Table 9: Key fields used in the *session* structure

type of currently active sessions. Defunct sessions are maintained in a similar manner as active ones, but in separate hash table and splay trees. To reduce resource consumption, we only maintain the TCP/UDP ports and ICMP type/code for defunct *DDoS* sessions. Once a new session is activated, the information on defunct sessions are consulted in hope that via data-correlation we can determine the application type of the session in question. We base this correlation on the premise that services at specific IP addresses and ports may not frequently change; for example, they remain bound for periods of less than 5 minutes.

SC helps monitor unsolicited traffic including ICMP-echo replies without corresponding requests, *TCP SYN|ACK* packets without corresponding *SYN* packets in place, and DNS replies without matching requests; all these may reveal *DDoS* attacks in formation. For example, when Message 1 of Table 1 is inspected, *SC* determines that it is an ICMP-echo reply and the tuple $<$ 192.168.5.142, 192.168.5.143, 0x0800, 0x037A, ICMP $>$ is formed. Here, string 0x0800 in the concatenation of the *icmp_type* and *icmp_code* fields (0x08 and 0x00), while 0x037A is the *icmp_id* of the message. The tuple assists in accessing the session structure and corresponding ICMP-echo request elements, if exists. As an ICMP-echo request is never created and transmitted in Table 1, the result of look-up is negative; subsequently, *SC* identifies Message 1 as unsolicited and may drop it according to an administrator-set configuration. Lastly, flows into the un-populated IP-space and unexpected TCP/UDP ports as well as inactive services are detected by the *SC* module as well.

3.5 *Message Sequencer (MS)*

The *Message Sequencer (MS)* facilitates packet re-assembly, traffic normalization, and enables stream state tracking. We use the structure *stream* whose key elements are shown in Table 10 to organize packets of a traffic stream within a session. For TCP streams, the field *state* tracks the originator's connection state which can be *SYN-SENT*, *SYN-RCVD*, *ESTABLISHED*, or *CLOSE*; fields *next-seq*, *last-ack*, and *window-size* maintain the next expected sequence number, acknowledged sequence number by the stream recipient, and the amount of transmitted data without acknowledgment. The *data* field is a pointer to an *interval tree* [17] used to organize all encountered packets in a single stream. Each *interval tree* node represents a packet P in the stream; the search key for the tree is in the range [SSN_p, ESN_p] where SSN_p and ESN_p are the start- and end-sequence numbers of the packet P in the node. SSN_p takes its value directly from the *tcp_seq* field in the TCP-header of P, while ESN_p is computed as (*ip_total*-(*ip_hlen*+*tcp_off*)$\ll$2), where *ip_total* is the total length of packet P, *ip_hlen* is the size of the IP-header of P, and *tcp_off* is the size of the TCP-header of P.

We use the second part of the *stream* structure of Table 10 to handle UDP/ICMP streams. The pertinent *interval tree* is anchored at *data* and fields *data-size* and *total-size* indicate the bytes stored in the tree and total number of bytes transferred in the stream. Based on this information, *MS* attempts to determine the type of a session; if more than a configurable amount of traffic–typically set to 5 KBytes–for a data stream of a session has been inconclusively inspected, the session type is declared *bypass*; no further analysis is carried out on the session's subsequent data. On the other hand, as soon as a session's type is determined, all its stored packets are flushed out and any forthcoming packets are not stored to achieve lower memory consumption and network latency. For each tree-stored UDP/ICMP packet P, its SSN_p and ESN_p values are assigned as follows: SSN_p is the current value of variable *total-size*, which is initialized to zero for every newly established session; ESN_p is the sum of *total-size* and the size of the UDP-payload in P. Upon storing a packet P, the *total-size* is updated appropriately.

field name	*size bytes*	*description*
		TCP stream
state	4	state of the stream such as *CLOSED, LISTEN, SYN_SENT, SYN_RCVD, ESTABLISHED*
next-seq	4	next sequence number expected, computed based on information of sender
last-ack	4	most recently acknowledged sequence number based on information of receiver
win-size	4	size of window advertised by receiver
data	4	pointer to a *interval tree* storing all packets of the stream, flush out periodically or after acknowledgment
		UDP or ICMP stream
total-size	4	total size of data transferred in the stream so far
data	4	pointer to an *interval tree* storing packets of the stream, periodically flushed out
data-size	4	number of bytes stored in the *interval tree*

Table 10: Key fields of the *stream* data structure

We define relationships between two intervals based on their start- and end-sequence numbers. For any two intervals [SSN_x, ESN_x] and [SSN_y, ESN_y], the former is less than the latter if ESN_x is smaller than SSN_y; similarly, the former is larger than the latter if the SSN_x is higher than ESN_y. The two intervals duplicate each other if their SSNs and ESNs are exactly the same; they overlap if they share a common sequence range and each has its own distinguished sequence range as well; or are contained if one's interval is a true subset of the other. Given that we deal with a packet P and an interval tree I, we use the above relationships in our interval-tree operations as follows:

- *interval-insert(I, P)*: inserts a node for packet P into I.
- *interval-delete(I, P)*: removes the node for packet P from I.
- *interval-retrieve(I, P)*: finds the set of nodes in I that may duplicate, overlap, or contain packet P.
- *packet-build(I, T_{start}, T_{end})*: creates a super-packet O from I; the sequence number interval of O is $[T_{start}, T_{end}]$.
- *interval-traversal(I)*: performs an in-order walk of tree I and lists all packets according to their non-decreasing SSN orders; this helps in writing the stream into permanent storage.
- *session-find(P)*: locates the session S of a packet P.
- *stream-find(S, P)*: finds the stream that a packet P of a session S belongs to.

Algorithm 4 shows the stream re-assembly process that *MS* carries out. For an arriving packet P, *MS* retrieves information on its session S and data stream I with the help of calls *session-find()* and *stream-find()*. Through the *interval-retrieve(I, P)* call, *MS* checks the relationship between P and any received packet in the same stream I while ensuring that there is no protocol anomaly in I. If the outcome of *interval-retrieve(I, P)* is empty, then P is a new packet as evidently no packet retransmission occurs; otherwise, the common parts of P and those packet(s) produced by *interval-retrieve* should have the same content; this is the case where P is a result of a TCP retransmission. Should the content comparison reveal differences,

Algorithm 4 *Message Sequencer* Algorithm

$P \leftarrow$ incoming packet;
$S \leftarrow$ *session-find*(P)
if (*TYPE* and *CONFIRM* of S are set) **then**
    P is part of *DDoS* session; hand it over to *Traffic Arbitrator (TA);* **exit**;
end if
$I \leftarrow$ *stream-find*(S, P); $Q \leftarrow$ *interval-retrieve*(I, P);
if (Q is empty) **then**
    P is a brand new packet and function *interval-insert*(I, P) is invoked to add P into I;
else
    check whether or not the common parts of P and any packet in Q have the same contents; if not, generate alerts and **exit**;
end if
$t_s \leftarrow$ initial sequence number of I; $t_e \leftarrow$ SSN$_R$+MAX_SIZE (default MAX_SIZE=5 KB); $O \leftarrow$ *packet-build*(I, t_s, t_e);
O is handed over to *Traffic Distinguisher (TD)*

P is part of evasive traffic produced by tools such as fragroute; the packet should be dropped (by default) and its corresponding connection be either terminated or manipulated and normalized with "favor-old" or "favor-new" policy which is configurable. If no suspicious evidence is found for P and its traffic stream I, P is inserted into I with the help of *interval-insert*(I, P). *MS* uses *packet-build()* to re-assemble received packets from stream I into super-packet O passed along with P to the *Traffic Distinguisher (TD)* module. One of the key tasks of *MS* is to establish boundaries for *DDoS* messages. For example, the *Stacheldraht* TCP-generated attacker-handler traffic of Table 4 violates boundaries of *DDoS* messages as packets 3, 4, and 5 points out. *MS* aggregates these packets together and helps restore the *DDoS* message boundaries.

3.6 *Traffic Distinguisher (TD)*

As individual *DDoS* attack systems follow their own protocols, we design *DDoS* specific analyzers to carry out application-oriented inspection and improve detection accuracy. Figure 6 depicts the elements of *Traffic Distinguisher (TD)* with the *Traffic Manager* playing the role of a scheduler which in turn invokes the analyzers for various *DDoS* tools if the application type of the session for the incoming packet P has not yet been determined. For each incoming packet P, *TD* obtained information on the session S of P, its traffic stream I, and the re-assembled super-packet O. If both *TYPE* and *CONFIRM* fields of S are set, P is forwarded directly into *Traffic Arbitrator* for additional processing; otherwise, *Traffic Manager* is invoked and Algorithm 5 identifies the application type of S as well as P. To improve performance, we restrict an upper limit to the total amount of transport data (TCP, UDP, or ICMP) examined in each traffic stream of a session before a decision on the application type of the session is made. Our experience shows that 5 KBytes of inspected traffic data in each direction of a session is very satisfactory. Should the application type of a session be un-determined yet after inspecting such amount of traffic, the session is pronounced *bypass* and no further processing occurs in its subsequent data transmissions.

The individual analyzers account for all elementary operations used by individual *DDoS* attacks as far as transport services, messages exchanged among clients, handlers, and agents as well as use of cryptographic algorithms, decoys and dynamic ports are concerned. In *Stacheldraht* for instance, TCP-based channels are used between clients and handlers, ICMP-based covert channels are used for exchanges between handlers and agents, while TCP/UDP/ICMP packets make up the actual attack traffic; moreover, *Stacheldraht* uses Blowfish to encrypt its messages. In the next section, we discuss in depth the analyzers for *Stacheldraht*, *TFN2K* and *DNS* amplification attacks.

Algorithm 5 *Traffic Manager* Algorithm

$P \leftarrow$ arriving packet; S, $I \leftarrow$ session and traffic stream that P belongs to; $O \leftarrow$ super-packet re-assembled with help of I;
if (*TYPE* and *CONFIRM* of S are set) **then**
    P is part of an identified *DDoS* session and is handed over to *Traffic Arbitrator (TA)*; **exit;**
end if
$t_I \leftarrow$ initial sequence number of I, $t_P \leftarrow$ start sequence number of P,
if (t_I-t_P) $\geq MAX\_SIZE$ (default MAX_SIZE=5KB) then S is marked as *bypass*; **exit;**
for (each *DDoS* analyzer *DA* implemented in the framework) **do**
    invoke *DA* with P, S, I, and O;
    break from the loop if *CONFIRM* of S has been set by *DA*;
end for
P is handed over to *Traffic Arbitrator (TA)* along with its S and I

3.7 *Traffic Arbitrator (TA)*

The task of the *Traffic Arbitrator (TA)* is to examine the application type of session S that the arriving packet P belong to, and take the prescribed actions on P and S. Should fields *TYPE* and *CONFIRM* of S not be set, *TA* simply forwards P to the next hop en route to its destination; otherwise, the *TA* proceeds according to policies set. Such policies include alert generation, logging of P as well as its data stream and session, blocking of subsequent messages from the same session, or even taking-over S by having *TA* act as a *DDoS* element such as a handler or agent. The *TA* also updates session information for S of Table 9 based on P so that subsequent re-assembly operations are facilitated and accuracy is enhanced. The module can log session, application type, creation time, all packets in the session, and session transmission statistics of a *DDoS* session for future forensic analyses. Algorithm 6 outlines the functionality of the *TA* which guides our *DDoS Container* deal with traffic. *TA* can also take over an identified *DDoS* session by playing the role

Algorithm 6 Outline for the operation of *Traffic Arbitrator (TA)*

Input: $P \leftarrow$ incoming packet; S, $I \leftarrow P$'s session and data stream
update information of S and I based on P
if (*CONFIRM* of S is not set) **then**
    application type of S has not been determined, P is forwarded, and current procedure stops
end if
if needed, generate an event log for the identified *DDoS* session along with information on S and I; and invoke *interval-traversal(I)* to dump all packets of I into permanent storage
if (action for *TYPE* of S is "proactive") **then**
    P is dropped, pertinent command is sent to the *DDoS* handlers or agents
else if (action for *TYPE* of S is "take-over") **then**
    P is dropped and fake reply is sent to the initiator of S
else if (action for *TYPE* of S is "terminating") **then**
    P is dropped and TCP RESET or ICMP "destination unreachable" packets are sent accordingly
else if (action for *TYPE* of S is "blocking") **then**
    P is dropped and all subsequent messages from S is dropped
else if (action for *TYPE* of S is "dropping") **then**
    P is dropped; however, subsequent messages from S may be forwarded it they do not contain malicious activities
else if (action for *TYPE* of S is "forwarding") **then**
    P is forwarded
end if

of a *DDoS* handler in reference to the attacking client [6]. To this effect, the *TA* dispatches either an *TCP RESET* packet or an *ICMP destination unreachable* message to the handler; subsequently, the *TA* may craft

[6]To avoid legal issues, such a feature is disabled in product versions. In addition, only *DDoS* sessions without encryption are taken over

fake replies to attacker-initiated commands, collecting valuable forensic information regarding the attack. The *TA* can be more proactive as it can disable attacks by purging all *DDoS*-related components from victim systems. This is assisted by *DDoS* tools themselves as they feature commands instructing handlers and agents to terminate activity and/or entirely remove themselves from compromised systems. For example, the *Stacheldraht* repertoire includes command *mdie* which terminates agents and command *msrem* which removes handlers.

4 Protocol Analyzers for *DDoS* Tools

In this section, we provide detailed discussion for the analyzers of two very common contemporary *DDoS* networks namely, *Stacheldraht* and *TFN2K* as well as the *DNS amplification attacks* . We have also developed analyzers for all components in Figure 6 and present them in [13].

4.1 *Stacheldraht*

The *DDoS* network *Stacheldraht*, German for barbed-wire, consists of agents, handlers and clients implemented with files *td.c*, *mserv.c* and *client.c* respectively. A client's interface performs *telnet*-like operations and uses a password protected channel for attacker–handler communications; messages in such sessions are Blowfish-encrypted. By default, handlers listen to TCP port 65512 for client instructions; each such handler may serve a configurable number of clients (default 200) and control a certain set of agents (default 6000). Agents monitor all incoming ICMP-echo reply messages awaiting for commands from handlers, at the same time, agents also listen to TCP port 65513 –also configurable– in order to exchange *keep-alive* messages with handlers. The attacker-provided password is initially DES-encrypted using a two character salt string whose default value is *zA*. The resulting 13-byte ASCII string is then Blowfish-encrypted using a pass-phrase hard-coded in the *client.c* and set to *authentication* by default; the encrypted password is padded to 1024 bytes before dispatched to handler. The handler reciprocates the above operation to verify the password as Algorithm 2 shows. Table 4 shows an client/handler session protected with password *iamnobody*. The string *zAGOe46FrqqVk* is the result of the *iamnobody* DES-encryption using salt string *zA*; Blowfish then uses pass-phrase *authentication* and pads the outcome with zeros for the trailing bytes of the message. All messages of Table 4 are similarly Blowfish-encrypted. Once an attacker provides the correct password, the handler displays the numbers of both active and inactive agents along with respective greeting messages and may accept commands on behalf of the client.

Stacheldraht handlers manage an attack network through a rich repertoire as Table 11 depicts. Commands are dot-prefixed and accept arguments. In particular, commands *.msrem* and *.msadd* help adjust the size of the attack network; *.micmp*, *.msyn*, and *.mudp* launch ICMP, TCP, and UDP flood-attacks, respectively; while *.setisize*, *.setusize*, *.sprange*, and *.mtimer* specify the size of attack packets, range of source ports for spoofing, and attack duration. Automatic agent updates are attained through command *.distro user server* issued by an attacker/client and delivered to all agents via handlers; the command instructs all agents to obtain and run a new version of *Stacheldraht* code from a host specified with the *server* parameter by using a copy facility such as *rcp* and an account indicated with the *user* parameter. For instance, upon receiving command *.distro user server*, a Linux–based agent executes in order shell commands *rm -rf agent*, *rcp user@server:linux.bin agent*, and *nohup ./agent* with *agent* being the name of the *Stacheldraht* executable

cmd from client	*parameters*	*cmd to agent*	*description*
.distro	user server	DISTROIT (6662)	Instruct agent to install and run a new version of system
.quit			Exit from the program.
.madd	ip1[:ip2[:ipN]]		Add IP addresses to list of attack victims.
.mdie		DIEREQ (6663)	Sends die request to all agents.
.mping		ICMP echo request	Pings all agents (bcasts) to see if they are alive.
.msadd	IP address	ADDMSERVER (5555)	Adds a new master server (handler) to the list of available servers.
.msrem	IP address	REMMSERVER (5501)	Removes a master server (handler) from list of available servers.
.mstop	ip1:ip2:ipN or all	STOPIT (3)	Stop attacking specific IP addresses, or all.
.mtimer	seconds	TIMESET (9011)	Set timer for attack duration.
.micmp	ip1[:ip2[:ipN]]	ICMP (1155)	Begin ICMP flood attack against specified hosts.
.msyn	ip1:ip2:ipN	SENDSYN (9)	Begin SYN flood attack against specified hosts.
.mudp	ip1:ip2:ipN	SENDUDP (6)	Begin UDP flood attack against specified hosts.
.setisize	size	SETISIZE (9010)	Sets size of ICMP packets for flooding. (max:1024, default:1024).
.setusize	size	SETUSIZE (8009)	Sets size of UDP packets for flooding (max:1024, default:1024).
.sprange	lowport highport	SETPRANGE (8008)	Sets port range for SYN flooding (defaults to [0, 140]).

Table 11: Clients to handlers *Stacheldraht* commands

daemon; *rm* purges the old version of the agent, *rcp* obtains a new copy of the agent from host *server* and finally *nohup* invokes the new code. All *Stacheldraht* messages are well-formed as Table 4 shows; however, due to the fact that client/handler messages are Blowfish-encrypted, their pertinent traffic-data appear to be random sequences. Thus, we resort to the communication behavior between clients and handlers in order to identify their traffic.

Algorithm 7 outlines our analyzer for interactions between clients and handlers and exploits two key *Stacheldraht* traffic characteristics. First, each message is Blowfish encrypted, encoded with a base-64

Algorithm 7 Analyzer for *Stacheldraht* traffic between clients and handlers

Input: packet P, its session S, stream I, and the assembled message O
if (field *CONFIRM* of S has been set) **then**
  P is handed over to *Traffic Arbitrator* module and exit;
end if
if (P is a TCP packet) **then**
  check payload size of O, if O is less than 1024 bytes, then exit from the procedure due to short of data;
  verify that O consists of two parts: one containing characters in [./, 0-9, a-z, A-Z] only; the other containing sequence of zeros only (may be empty). Otherwise, clean corresponding bits in "TYPE" and *CONFIRM* as it cannot be *Stacheldraht* session;
  if (P is from client to handler) **then**
    set bit in *TYPE* corresponding to *Stacheldraht*; store the non-zero part of O;
  else
    set bit in *CONFIRM* corresponding to *Stacheldraht* if the non-zero part of O is the same as that stored in the session
  end if
else if (P is a ICMP packet) **then**
  check its *icmp_id* and payload of P with the help of Table 12; set fields *TYPE* and *CONFIRM* of S accordingly
end if

scheme, and padded with zero to 1024 bytes before transmitted to its recipient; this yields a message consisting of a series of characters in the [./, 0-9, a-z, A-Z] range followed only by *NULL* characters. Provided that passwords, commands and handler-replies are short (typically less than 100 Bytes) and Blowfish does not change their size, the padding part of a message is extremely long offering a reliable traffic pattern. Second, the handler always echoes back the password in encrypted-form back, forcing the first message in both directions to have the same payload. Our analyzer takes advantage of this effective two-message correlation to identify *Stacheldraht* client/handler traffic. Clearly, the same correlation technique can be used on different messages within the same stream in anticipation of enhanced accuracy in traffic identification. In this regard, greeting messages from handlers to clients may be viable candidates for data correlation. Our experimental evaluation demonstrates that the correlation of the first message in each direction is for all practical purposes very effective. With attacker/handler connections established through the TCP/IP stack, *Stacheldraht* message boundaries are not always respected as Table 4 points out. Algorithm 7 uses the TCP-stream reassembly of the *Message Sequencer (MS)* module to reconstruct messages out of TCP packets.

All client-originated commands "staged" at handlers are ultimately forwarded to agents via ICMP messages; the *icmp_id* field of these messages contains the command identifiers while all relevant parameters are placed in the ICMP payload. Column *cmd to agent* of Table 11 presents some command identifiers and their default values in *Stacheldraht*. All ICMP-delivered commands and parameters between handlers and agents are neither encrypted nor recipient-authenticated. For example, the client-issued command *mudp ip1:ip2* is delivered via a handler to an agent with an ICMP-echo reply whose *icmp_id* field is set to SENDUDP (6 by default) and the payload contains the integer representation for *ip1* and *ip2*. In addition, handlers and agents exchange additional messages for maintenance tasks shown in Table 12. Once an agent becomes

cmd	*dir*	*size*	*type*	*icmp_id*	*payload*	*description*	*check?*
Agent is denoted as A; Handler is denoted as H							
spoofing probe	A− >H	112	8	0	agent's IP	test spoofing level, src IP address is 3.3.3.3	yes
spoofing probe reply	H− >A	1044	0	1016	spoofworks	reply to spoofing probe	yes
agent ping	A− >H	1044	0	666	skillz	test availability of handler	yes
agent pong	H− >A	1044	0	667	ficken	reply to ping	yes
handler ping	H− >A	1044	0	668	gesundheit!	test availability of agent	yes
handler pong	A− >H	1044	0	669	sicken|0A|	response to handler ping	yes
kill agent	H− >A	1044	0	666	skillz	kill agent, src IP address set to 3.3.3.3	yes
kill reply	A− >H	1044	0	1000	spoofworks	reply to kill agent	yes
stop attack	H− >A	1044	0	9015	niggahbitch	stop any ongoing attack	yes

Table 12: *Stacheldraht* handler/agent messages transported via ICMP

operational, it tries to locate handlers by examining a Blowfish-encrypted[7] file named *.ms*. The agent may also try to connect to handlers hard-coded in its source to achieve the same goal. To determine handler availability, an agent sends ICMP-echo reply messages to candidates with field *icmp_id* set to 666 and payload containing string *skillz*. An active handler replies with an ICMP-echo reply whose *icmp_id* is 667 and the payload contains the string *ficken*. Similarly, a handler uses the message *handler ping* to test the availability of an agent; this may trigger a *handler pong* reply from an active agent.

To find out whether network devices such as routers forward packets with spoofed IP addresses, an agent crafts and dispatches to a handler a "spoofing probe". The latter is typically an ICMP-echo message which has the forged source IP address 3.3.3.3 and the payload carries its real source IP address; apparently, such a spoofed-message should not be forwarded by routers with egress filtering. In the case that they are, the handler forms an ICMP-echo reply whose *icmp_id* field is set to *SPOOF_REPLY* (1016 by default as shown in Table 12); the message payload contains the string *spoofworks* and its destination address is set to the one contained in the payload of the probing ICMP. Upon receipt, the agent becomes aware that network devices allow spoofed-messages and commences using fake source addresses for its subsequent messages. Otherwise, the agent falsifies only the last octet of the IP addresses. Through Algorithm 7, our analyzer pursues relevant protocol fields and payloads shown in Table 12 to discern ICMP-based *Stacheldraht* sessions.

The *Stacheldraht* network can also mount multiple type attacks such as ICMP, SYN, UDP and Smurf floods. Table 13 shows unique characteristics of TCP, UDP, and ICMP attack packets that can be exploited to effectively identify *Stacheldraht* attack traffic. Regardless of the protocol used, the fields *ip_tos* and *ip_flags* for all packets, are set to zero. In TCP packets, the sequence number is always the same which constitutes a

[7] with passphrase *randomsucks*

field	*characteristics*	*check?*
ip_tos	always set to zero	*Protocol Decoder*
ip_flags	set to zero, meaning bits "don't fragment" and "more fragment" are unset	*Protocol Decoder*
	attack packets based on TCP	
ip_ttl	always set to 30, a relatively small value	*Protocol Decoder*
src_port	in [1001, 2024], a relatively small range	*Protocol Decoder*
tcp_flags	only SYN is set	*Protocol Decoder*
tcp_seq	always 0x28374839 (in host order), an obvious anomaly	*Protocol Decoder*
tcp_urg	random number (rarely zero), an error since bit "URG" not set	*Protocol Decoder*
tcp_win	always 65535	*Protocol Decoder*
	attack packets based on UDP	
ip_ttl	set to 0xFF	*Protocol Decoder*
udp_sport	in [1, 10000], decrement by one for each subsequent packet	*Behavior Police*
dst_port	in [0, 9999], increment by one for each subsequent packet	*Behavior Police*
udp_checksum	random number or zero, not calculate at all	*Protocol Decoder*
	attack packets based on ICMP	
ip_id	fixed number (process ID of the agent)	*Behavior Police*
ip_ttl	set to 0xFF	*Protocol Decoder*
icmp_checksum	fixed number and incorrect	*Behavior Police*

Table 13: Unique characteristics of TCP, UDP and ICMP attack packets in *Stacheldraht*

clear violation of the protocol; moreover, additional irregularities appear including not empty *urgent pointer* field, unset *URG*-bit in *tcp_flags* and field *ip_ttl* having a relative small value. In UDP packets, the source port number is initially set to 9999 and decremented by one for every subsequent packet; similarly, the initial destination port is set to one and incremented by one. It is worth pointing out that the checksum of such packets is not computed and may be either zero or a random number. As packets go through the *Protocol Decoder (PD)*, the above simple structural packet irregularities are identified, the application types can be determined, and configurable actions such as blocking can be taken. In addition, the *Behavior Police (BP)* is able to identify flooding attacks generated by *Stacheldraht* via the relationships between source and destination ports of UDP-packets. Therefore, a large amount of attack traffic can be identified quickly and avoid the processing by *Session Correlator (SC)*, *Traffic Distinguisher (TD)*, and *Message Sequencer (MS)*, which is expensive in terms of CPU cycles and memory consumption.

4.2 *Tribe Flood Network 2000 (TFN2K)*

Handlers and agents routinely make up a *Tribe Flood Network 2000 (TFN2K)* in which individual handlers control groups of agents. The key *TFN2K* feature is that communications are unidirectional from handlers to agents. Messages are transported via TCP, UDP and ICMP, encrypted with strong cryptographic algorithms such as CAST and encoded with a base-64 scheme. The encryption key is defined at compile-time and is used as the password to access *TFN2K*. To reduce the probability of detection, *TFN2K* interleaves its message flow with decoy packets as Table 5 shows and by default, handler originating messages have spoofed source IP addresses. When ICMP is employed as the covert communication channel between a handler and an agent, an ICMP-echo reply is used to avoid returned message from agent's TCP/IP stack. On the other hand, if TCP/UDP transport services are used, most of the protocol fields in the generated packets have randomized contents. In particular, the *udp_length* is always set to larger than its actual size by three bytes, the TCP *tcp_off* field is invariably set to zero; both of these abnormalities are likely to create malfunctions in network devices along the communication path. Finally, the UDP/TCP checksums are solely computed on packet headers and payloads without considering the required 12-byte pseudo-headers rendering them corrupt.

Should an intruder successfully pass the password-authentication and obtain access to a *TFN2K* network, she may use the commands of Table 14 to communicate with handlers and launch various attacks including SYN-floods, *Smurf*, and *Targa*. Each command is assigned a unique numeric identifier and may accept parameters as shown in columns *cmd* and *parameters* of Table 14. For instance, the attacker can launch SYN-floods, *Smurf*, and *Targa* attacks with commands 5, 7 and 9 respectively. Based on the command issued by the attacker, the *TFN2K* handler constructs a TCP, UDP, or ICMP message, and subsequently delivers the message to agents after encryption and encoding. Before encryption and encoding, the message is text-based and follows the format convention *+symbolic_id+data*, where *symbolic_id* is a single character representing a specific command, + is a separator and *data* outlines the specific command parameters. For instance, as

cmd (numeric id and parameters	*description*	*msg format before encrypt/encode*
0 (void)	stop ongoing floods	+d+
1 -i spoof-level	set IP spoof level, 0 (32 bits), 1 (24), 2 (16), or 3 (8)	+c+level
2 -i packet-size	Change Packet size	+b+packet-size
3 -i remote-port	Bind root shell to a port	+a+remote-port
4 -i victim1@victim2@...	UDP flood	+e+victim1@victim2@...
5 -i victim1@victim2@... [-p dest-port]	TCP/SYN flood	+g+dest-port; +f+victim1@victim2@...
6 -i victim1@victim2@...	ICMP/PING flood	+h+victim1@victim2@...
7 -i victim@broadcast1@broadcast2@...	ICMP/SMURF flood	+i+victim@broadcast1@broadcast2@...
8 -i victim1@victim2@...	MIX flood (UDP, TCP, ICMP)	+k+victim1@victim2@...
9 -i victim1@victim2@...	Targa3 flood	+j+victim1@victim2@...
10 -i command	execute remote shell command	+l+command

Table 14: *TFN2K* client-to-handler-to-daemon commands

shown in the first message of Table 2, after granted access to the handler at IP address 192.168.5.143, the intruder issues command "tfn -P ICMP -h 192.168.5.142 -c 4 -i 192.168.5.37" in order to launch a UDP flooding attack, where argument "-P ICMP" specifies ICMP as the transport service for the communication between the handler and the daemon, argument "-h 192.168.5.142" indicates the location of the daemon, while option "-c 4" is the command identifier for UDP flooding attack, and option "-i 192.168.5.37" is the parameter to the command specifying the primary victim. Such an attacker-issued command is transferred as message "+e+192.168.5.37", and delivered to the daemon at 192.168.5.142 as the payload of an ICMP echo reply message after encryption and encoding. The same command issued by the attacker but delivered to agents with UDP and TCP transport services are described in Tables 3 and 5. Such well-formed messages will be easily identified if they are transmitted in plain text. However, as Tables 2 and 3 show, encryption and encoding are used to obfuscate messages before delivery. Therefore, our *TFN2K* analyzer predominantly resorts to behavior analysis and protocol anomaly inspection.

At its core, our analyzer mainly exploits the artifact of trailing *A*s[8] at the end of every message as discussed in Algorithm 1. It is also worth noting that the length of the trailing sequence can be readily computed as demonstrated in Algorithm 8, which outlines the function of our *TFN2K* analyzer. First, the analyzer inspects the content and size of the incoming packet P to ensure it is encoded with the *TFN2K* base-64 scheme. Then, it computes the number of trailing *A*s based on the length *blen* of P's payload following the inverse procedure of that in Algorithm 1. The length *clen* of the CAST-encrypted-and-padded cipher text can be determined with the help of length *blen*, which is the payload size of P and also the length of the encoded cipher. In order to figure out the number of padding zeros to the cipher text before encoding,

[8] whose ASCII code is 0x41

Algorithm 8 *TFN2K* Traffic Analyzer

Input: packet P, its session S and stream I
if (*TYPE* of S is already set) **then**
    application type of P as well as of S has been identified; P is directly handed over *Traffic Arbitrator* and exit;
end if
check transport payload of P to ensure that all characters are in [A-Z, a-z, 0-9, +/];
$encode \leftarrow$ payload of P; $blen \leftarrow$ length of $encode$; $q \leftarrow$ (*blen* div 4); $r \leftarrow$ (*blen* mod 4);
S cannot be *TFN2K* if r is not 0, 2, or 3;
$clen \leftarrow (3\ q)$ if (r = 0); $clen \leftarrow (3\ q + r$ - 1) otherwise; $plen \leftarrow (clen$-16); P cannot be *TFN2K* if (($clen < 16$) or ($plen < 4$));
$q \leftarrow$ ($plen$ div 16); $r \leftarrow$ ($plen$ mod 16); $elen \leftarrow plen$ if (r = 0), or $elen \leftarrow$ (16(q+1)) otherwise;
$q \leftarrow$ ($elen$ div 3); $r \leftarrow$ ($elen$ mod 3); $start \leftarrow (4q)$ if (r = 0), or $start \leftarrow (4\ q + r + 1)$ otherwise; Clearly, $start$ is the start point (indexing from 0) for the trailing A sequence if P is created by *TFN2K*.
check content in [0, $start$) of $encode$ for pattern “AAAA”, if found, exit as P cannot be *TFN2K*;
check content in [$start$, $blen$) of $encode$ for any non-A character, if found, S cannot be *TFN2K* and exit;
set *TYPE* of S to *TFN2K* if it is not set yet; otherwise, set *CONFIRMED* of S to *TFN2K*;

we first compute the size of the original plain text *plen* as (*clen*-16), which in turn helps determine the size *elen* of the cipher text without padding. The number of padding zeros to the cipher text can be calculated as (*clen*-*elen*), the latter is used to determine the length of the *A*-sequence. With the help of *elen*, we can find out the starting point for the trailing A sequence, while *blen* can be used to determine the end point of the trailing A sequence as shown in Algorithm 8. Finally, the *TFN2K* analyzer examines the content of the should-be-trailing area (i.e., [$start$, $blen$] in Algorithm 8) to ensure that it exclusively consists of *A*s. In order to reduce false positives, the *TFN2K* analyzer also inspects the content in [0, $start$) of P, which should be the CAST-encrypted cipher text encoded in the base-64 scheme. It is reasonable to expect that CAST encryption algorithm does not produce recognizable patterns in its cipher text such as a sequence of zeros [29, 49]. Therefore, our *TFN2K* analyzer assumes that the CAST-encrypted *TFN2K* cipher texts do not contain a sequence of three consecutive zeros encoded as pattern *AAAA* in the base-64 scheme. To put it simply, *AAAA* cannot appear in the encoded cipher text of *TFN2K* message.

Since raw-sockets are used to transmit packets between handlers and agents, IP, ICMP, TCP and UDP headers have *TFN2K*-assigned values. In this manner, *TFN2K*-generated packets feature a number of unique characteristics as Table 15 depicts. What all packets share regardless of their transport protocol is that their *ip_tos* field is set to zero, *ip_ttl* has values in the range [200, 255], and *ip_id* takes values in the range [1024, 65535]. Also half the times, fields *icmp_seq* and *icmp_id* have zero values and the rest assume random values. Similar observations are drawn for fields *tcp_seq*, *tcp_ack*, and *tcp_win* in TCP packets. Moreover, the TCP-header field of *tcp_off* is set to zero, wrong values appear in the *udp_length* of UDP-header and checksums for all TCP/UDP packets are incorrectly computed. Our *TFN2K* analyzer exploits such packet abnormalities while a packet is being examined in the *Protocol Decoder (PD)* module. Similar protocol anomalies can be observed in pure *TFN2K* flooding attack traffic as well. For instance, checksums for TCP and UDP packets are incorrectly calculated; field *ip_tos* is always zero; the TCP protocol header *tcp_off* is set to zero. In addition, for UDP flooding attacks, the source port number decreases by one and the destination port number increases by one for each subsequent packet, while their sum always remains constant to 65536; this pattern is exploited by the *Behavior Police* module of our *DDoS Container* to discern *TFN2K* UDP flooding attacks.

field	*characteristics*	*checked by module*
ip_tos	always set to zero	*Protocol Decoder*
ip_flags	set to zero, meaning bits "don't fragment" and "more fragment" are unset	*Protocol Decoder*
ip_ttl	random in [200, 255]	*Protocol Decoder*
ip_id	random in [1024, 65535]	*Protocol Decoder*
Packets based on TCP		
tcp_flags	SYN|ACK, SYN, or ACK	
tcp_seq, tcp_ack, tcp_win	half of time 0, others random	*Protocol Decoder*
tcp_off	always set to zero	*Protocol Decoder*
tcp_checksum	incorrectly left out pseudo-header	*Protocol Decoder*
Packets based on UDP		
udp_length	3 bytes larger than true value	*Protocol Decoder*
dst_checksum	incorrect as pseudo-header not included	*Protocol Decoder*
Packets based on ICMP		
icmp_type, icmp_code	zero	*Protocol Decoder*
icmp_id, icmp_seq	half of time zero, others random	*Protocol Decoder*

Table 15: Unique Handler-to-agent TCP/UDP/ICMP-packet characteristics

4.3 *DNS Amplification Attacks*

The Domain Name Service (*DNS*) system provides translation services between domain names and IP addresses using a hierarchical overlay network over the Internet [43, 61, 60]. For flexibility, many *DNS* servers act as open-resolvers and automatically forward DNS queries to other authoritative name-servers on behalf of requesters [61]. Open-resolvers have been recently used to conduct *DNS amplification attacks*. Such attacks proceed into two phases: initially, they harvest a large number of Internet open-resolvers and subsequently, they generate and deliver over-sized UDP-*DNS*-queries. Here, the size of queries is typically larger than 1024 bytes. *DNS* attacks often use the IP-address of a victim as the source address in all *DNS* requests which generate the same number of responses delivered to the victim. In this way, amplification attacks force all resolver-responses to reach and overwhelm a single victim [60]. The problem is further exacerbated with the poor life-cycle management of *DNS* resource records (*RRs*) in many name servers, which include expired host addresses and outdated entries for start of authority (*SOA*).

In developing an analyzer for *DNS amplification attacks*, we assume that our *DDoS Container* operates along with a firewall unit that can be easily configured to detect and block packets with spoofed IP-addresses. Each *DNS* message –query (Q) or reply (R)– contains a header, which has a fixed size of 12 bytes. There are 6 fields in the DNS header, and includes fields *transaction ID*, flags, numbers of questions, numbers of answer *RRs*, numbers of authority *RRs*, and numbers of additional *RRs*. each of which is 2 bytes long. The field *transaction ID* is used to match a *DNS* query and its corresponding reply [61]. The *Q/R* (query/reply) bit of the *flags* field in the *DNS* header of a message makes it straightforward to determine whether a message is either a query or a response.

Our analyzer exploits the fact that an *DNS* amplification attack launched from an external network and targeting machines in an internal network can be easily identified as its replies (Rs) have no corresponding originator in the internal network. By also taking into account that a firewall may block all incoming packets with incorrect destination IPs and outgoing packets with spoofed source IPs, an enterprise can be fully protected against amplification attacks.

Algorithm 9 outlines our analyzer as it deals with interactions between open-resolvers and victims. In the context of our framework, the analyzer is provided with a dedicated bit in the fields *TYPE* and *CONFIRM*. The algorithm differentiates encountered *DNS* sessions into the following categories:

Algorithm 9 *DNS amplification attack* Analyzer for traffic between open-resolvers and victims

```
Input: packet P and its session S
if (field CONFIRM of S is set) then
    P is handed over to Traffic Arbitrator module and exit;
end if
if (P is not a UDP DNS packet) then
    P is handed over to Traffic Arbitrator and exit;
end if
if (P is a DNS query) then
    store the header's transaction ID and set bit in TYPE corresponding to DNS amplification
else
    if (bit in TYPE corresponding to DNS amplification not set) then
        set CONFIRM of S and exit; (case III)
    end if
    if (transaction ID of P != stored transaction ID) then
        drop P and exit; (case II)
    else
        P is a normal DNS reply and S is a legitimate DNS session; (case I).
    end if
end if
```

I: normal sessions where a *DNS*-query is first encountered by the analyzer and then a reply appears with *transaction ID* identical to an already encountered query.

II: abnormal *DNS* sessions where queries and replies are detected by the analyzer but they have different *transaction ID*s. In this rare case, the *DNS*-replies can be considered as outdated messages and responses to previous *DNS* queries.

III: amplification attack sessions that contain *DNS*-replies only.

Evidently in case *I*, the traffic is simply passed over to the *Traffic Arbitrator*. In case *II*, the analyzer may simply discard the incoming packet P as the mismatched *DNS* reply likely originates from old and non-existing session at this time. In case *III*, the analyzer identifies an *DNS* amplification attack, marks the corresponding session as such, and the traffic from this point on is handled by the *Traffic Arbitrator*.

If both attackers and victims reside in networks protected by our *DDoS Container*, the above *DNS* analyzer is expected to fail as such *DNS*-messages are treated as belonging to session category *I*. However internal-network originated and bound incidents are handled by our *Behavior Police (BP)* module. Within this module, we define a *DNS flood template* that helps identify UDP-flooding due to internal amplification attacks. For instance, the template *(prot=UDP)&&(dst_ip=same)&&(src_p=DNS)* along with threshold assigned to a value such as 1000 pps can identify *DNS* attacks with intensity above 1000 pps. Here, the condition *dst_ip=same* groups all traffic from the same destination IP address while *src_p=DNS* indicates the same source port (i.e., 53). Overall, our analyzer in concert with a firewall and the above *BP* template can effectively detect *DNS* amplification attacks.

5 Implementation and Experimental Evaluation

We implemented the proposed *DDoS Container* in C and integrated it as a module in FortiGate-800, a multi-functional device that operates in inline fashion and provides firewall, anti-virus, and IDS/IPS functionalities [30] and whose rated speed is 400 Mbps. We deployed our *DDoS Container* in test-bed environments that follow the network layout of Figure 7 and installed binaries of *DDoS* clients, handlers, and agents in a number of test machines so that various *DDoS* networks are formed. Here, the networks are essentially partitioned in internal and external segments that allow us to better control the traffic observed by FortiGate-

800. For instance, communications between *TFN2K* $handler_1$ and $agent_2$ pass through the *DDoS Container* system, making possible for the latter to manipulate the ensued traffic. All 20 test machines operate either *Windows2000* or *Linux* and are connected to FortiGate-800 via 100 Mbps switches. In order to verify the behavior of our system, we use a *Ethereal* traffic sniffer [20] on a dedicated machine –*Sniffer* in Figure 7– to capture data exchanged among test machines and our *DDoS Container*.

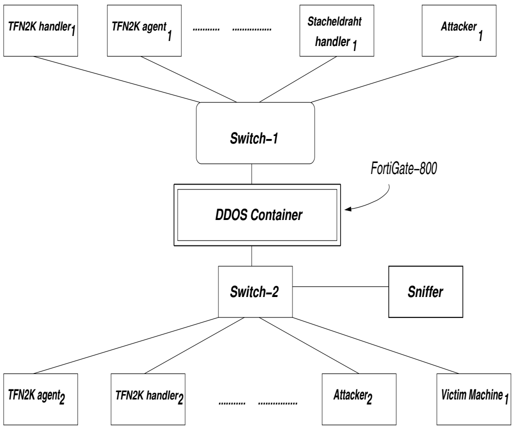

Figure 7: Deployment of our *DDoS Container*

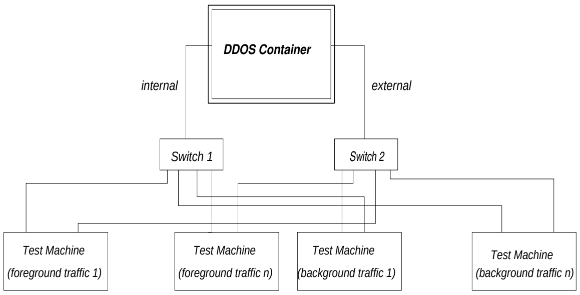

Figure 8: Trace-driven *DDoS Container* test environment

Figure 8 depicts the specific trace-driven testbed that we used for all reported results. By offline creating streams with varying mixes of *DDoS* traffic (as foreground) and normal traffic (as background) and storing them in separate trace files, we are able to inject them to the *DDoS Container* using different testers. By controlling the traffic injection, we are able to establish the behavior of our *DDoS Container*. In what follows, we briefly present the main results of our experimental evaluation.

5.1 Baseline Behavior for the *DDoS Container*

To establish the baseline behavior of our *DDoS Container* and ascertain its capabilities in identifying communications among *DDoS* elements, we use the environment of Figure 7. Due to the hierarchical network layout, it is possible that the *DDoS Container* may not observe all *DDoS* activities depending on its location. In this baseline phase, we mainly focus on the *DDoS Container* identification capabilities and so we configure the system to create alerts for suspicious sessions but forwarding all packets to their destination. This "alert-only" configuration makes it feasible for the sniffer to capture the generated traffic and form traces that can be run in the environment of Figure 8. We conduct experiments with both default and customized *DDoS* codes.

In the course of default deployment, we use the Internet-available *DDoS* sources and compile them in their default settings including TCP/UDP-ports, passwords, and encryption keys. Although we repeat the process for each toolkit, we only outline the testing with *TFN2K* for brevity. We use machines $attacker_1$, *TFN2K*-$handler_2$, *TFN2K*-$agent_2$ and $victim_1$ to construct a *DDoS* network; here, the attacker uses *telnet* to access the handler and the *DDoS Container* observes only the attacker-handler communications. The FortiGate–IPS module has a built-in *telnet* analyzer that we exploit to identify all *TFN2K* attacker-issued commands of Table 14. On the other hand, we use machines $attacker_1$, *TFN2K*-$handler_1$, *TFN2K*-$agent_2$ and $victim_1$

to make handler-to-agent communications visible to the *DDoS Container*. With the help of command line option "-P", the attacker instructs the handler to use different transport services including TCP, UDP and ICMP. Segments of such handler-to-agent traffic captured by the sniffer are shown in Tables 2, 3, and 5. Our *DDoS Container*successfully identifies such *TFN2K* handler/agent communications. Should we deploy the configuration consisting of $attacker_1$, *TFN2K*–$handler_2$, *TFN2K*–$agent_1$ and $victim_1$, we expose to the *DDoS Container* all messages exchanged among *DDoS*–elements involved. Apparently, the environment of Figure 7 allows for the easy deployment of a single attacker controlling multiple handlers, the co-existence of multiple attackers, or even a handler manipulating a group of zombies. In all the above settings, our *DDoS Container* detects all pertinent *TFN2K* sessions. By repeating the same experiments for the *Stacheldraht*, *Mstream*, *Trinoo* and *Kaiten* we reach the same outcome. During the second stage of our baseline experiments, we customize *DDoS* codes by re-designating ports, passwords, encryption keys and salt-strings used. By repeating the aforementioned set of experiments, we show that our *DDoS Container* correctly identifies all *DDoS* sessions as it bases its operation on deep inspection and behavior analysis instead of static port information and specific encrypted patterns.

Next, we turn our attention to *DDoS Container* performance, in particular, we investigate the maximum number of concurrent sessions sustained by our implementation. Provided that *DDoS* user work in an interactive manner and the time gap between two-consecutive commands is often long, overheads for the operation of *DDoS Container* do not appear to be a critical issue. We anticipate however that our *DDoS Container* will be ultimately deployed at the perimeter of networks; in this regard, it may encounter a tens of thousands of concurrent sessions and the performance may be adversely affected if heavy overhead is present. To determine the capabilities of our system, we use the traffic of Table 4 as a template and generate test-cases executed in the environment of Figure 8. The traffic consists of two parts: the first features packets generated by the normal three-way handshake procedure not shown in the Table 4 and packet 1, while the second part contains packets 2–5 and the remaining in the session[9]. We configure *DDoS Container* to forward all packets but generate two alerts for every *Stacheldraht* session. The first alert is generated when an session is tentatively marked as *Stacheldraht* by using the attacker-to-handler stream (i.e., packet 1). The second alert is created when a session is confirmed using the handler-to-attacker stream and packet 2 has been processed. Should no more memory be available for the processing of an arriving session, the *DDoS Container* does not track it and allows the corresponding packets to pass through (i.e., fail open).

Each test carried out consists of n *DDoS* (or foreground) and m normal (or background) sessions. Both n and m take values in [100,000, 700,000] as we anticipate that such choices are representative for the operation of a device at the edge of the network; this selection is also dependent on the total amount of memory available in FortiGate-800 –default 4 GBytes– and the requirements for session representation. In *Stacheldraht* for example, each connection requires at least 41 Bytes according to Table 9. Based on Algorithm 7, certain amount of data in the first attacker-to-handler message has to be stored in order to perform the correlation with the corresponding handler-to-attacker message; this includes the encrypted session password which is often less than 50 Bytes. Finally, by taking into account overheads for the organization and/or maintenance of hash tables, session splay and interval trees needed for TCP reassembly, each session necessitates at least 512 Bytes of overhead. According to Algorithm 4, we may need to store upto 5 KBytes data exchanged in the session before we are able to determine that a stream is of background/non-*DDoS* nature. Taking into account the above, the requirements for main memory in the case of *Stacheldraht* is approx-

[9] not shown in Table 4

imately *M=0.5*n+5.5*m* KBytes; should both concurrent foreground and background sessions be around 650,000 respectively, the memory requirements before we start losing sessions is approximately 4 GBytes. We should point out that the above estimation for memory consumption *M* presents an upper bound as other *DDoS* tools do not need to store much data as is the case with *Stacheldraht*.

We use n_f[10] machines marked as *foreground testers* in Figure 8 to launch foreground traffic; each replays the first half of the Table 4 for n/n_f times, then pauses for a second and resumes by replaying the second part of the trace for the same number of times. At the same time, m_b *background testers* –default 10– feed noise or normal application (FTP) traffic; this trace is split into two parts and each one is injected into the *DDoS Container* m/m_b times with an intermission of one second. The testers modify the IP addresses and port numbers of both source and destination for each new connection to avoid conflicts among different replayed sessions. In each test, we monitor the behavior of *DDoS Container*, record the number of sessions identified as *Stacheldraht* and alerts generated, and compute the ratio of correctly marked *Stacheldraht* sessions. Table 16 shows the outcome of our testing; each row specifies the number n of foreground sessions, while each column indicates the number m of background streams. As the total number of both foreground and background streams increased, our *DDoS Container* produces correct behavior except for the case of (n,m)=(700,000, 700,000). The latter suggests that our prediction for the required memory M in this set of

	m=100,000	m=300,000	m=500,000	m=600,000	m=700,000
n=100,000	100.00	100.00	100.00	100.00	100.00
n=300,000	100.00	100.00	100.00	100.00	100.00
n=500,000	100.00	100.00	100.00	100.00	100.00
n=600,000	100.00	100.00	100.00	100.00	100.00
n=700,000	100.00	100.00	100.00	100.00	99.99

Table 16: *DDoS Container* test results of *Stacheldraht* workloads

experiments was over-estimated.

By repeating the above testing procedure for all attack tools, we establish similar results. For each test case, we adjust the replay rates of all sessions during the traffic injection, so that we can generate traffic with various characteristics, including constant bit rate (CBR), self-similar, and normal-distributed traffic. However, the aggregated traffic is controlled so that the rated speed of the FortiGate is not exceeded, therefore rendering that any packet drop is introduced due to exceeding memory consumption of the *DDoS Container*. Our experiments show that the memory consumption of the *DDoS Container* under various traffic patterns with different characteristics is similar, indicating that the memory consumption is closely related to the number of concurrent sessions.

5.2 Identifying *DDoS* Attacks Using *Snort-Inline*

Conventional security mechanisms such as IDSs/IPSs can identify *DDoS* attacks only with the help of specially-crafted signatures but remain "unaware" of the unique characteristics of *DDoS* attacks. *Snort-Inline*, an open source IDS/IPS, is implemented atop the libpcap packet-capturing library and is mainly deployed in small networks. It can be configured to detect *DDoS* traffic provided that special signatures such as those of Table 17 are crafted.

By examining for the agent-generated pattern "*shell bound to port*" in the ICMP-echo-reply payload,

[10]set to n_f=10

no.	*rule*	*explanation*
1	icmp $HOME_NET any − > $EXTERNAL_NET any (msg:"TFN server response"; icmp_id: 123; icmp_seq: 0; itype: 0; content:"shell bound to port"; sid:238;)	inspect ICMP message with type "echo reply", ip_id = 123, and telltale
2	icmp $EXTERNAL_NET any − > $HOME_NET any (msg:"tfn2k icmp possible communication"; icmp_id: 0; itype: 0; content:"AAAAAAAAAA"; sid:222;)	ICMP type "echo reply", ip_id = 0, and pattern "AAAAAAAAAA" in payload
3	udp $EXTERNAL_NET any − > $HOME_NET 31335 (msg: "Trin00 Daemon to Master message"; content:"l44"; sid:231;)	inspect UDP packet with dst_port 31335 and pattern "l44" in payload
4	tcp $EXTERNAL_NET any − > $HOME_NET 27665 (msg: "Trin00 Attacker to Master default startup password"; flow: established, to_server; content: "betaalmostdone"; sid:234;)	inspect TCP packet with dst_port 27665, telltale "betaalmostdone" in payload
5	udp $EXTERNAL_NET any − > $HOME_NET 6838 (msg: "mstream agent to handler"; content:"newserver"; sid:243;)	inspect UDP packet with dst_port 6838 and pattern "newserver" in payload
6	tcp $HOME_NET 12754 − > $EXTERNAL_NET any (msg:"DDOS mstream handler to client"; flow: to_client, established; content:">"; sid:248;)	inspect TCP packet with src_prot 12754, ">" in payload, and from server side
7	icmp $EXTERNAL_NET any − > $HOME_NET any (msg:"DDOS Stacheldraht client spoofworks"; icmp_id: 1000; itype: 0; ; content: "spoofworks"; sid:227;)	inspect ICMP packet with type "echo reply", ip_id = 1000, and "spoofworks" in payload
8	icmp $HOME_NET any − > $EXTERNAL_NET any (msg:"DDOS Stacheldraht server response"; icmp_id: 667; itype: 0; ; content: "ficken"; sid:226;)	inspect ICMP packet with type "echo reply", ip_id = 667, and "ficken" in payload

Table 17: Rules/signatures used in *Snort-Inline* to detect *DDoS* traffic

Rule 1 attempts to identify *TFN* traffic; at the same time, fields *icmp_id* and *icmp_seq* of the packets should comply with the *TFN* requirement of having values 123 and 0, respectively. However, by applying Rule 1 to the traffic of Table 1, no alarm is generated as the sought pattern cannot be found. Rule 2 detects the control traffic between *TFN2K* handlers and agents by both searching for a sequence of ten trailing "A"s in the ICMP-echo-reply message and ensuring that the *icmp_id* value is zero. Due to encryption/encoding used, the length of *TFN2K*-generated "A"s varies between [1..21] and in this respect, Rule 2 fails to identify some pertinent traffic such as those in Table 2. Moreover, Table 3 TCP/UDP-traffic escapes the *Snort-Inline* detection entirely as far as Rule 2 is concerned. To detect *Trinoo* traffic, Rules 3 and 4 monitor UDP packets at port 31335 for string *"l44"* and inspect the TCP stream at port 27665 for telltale "*betaalmostdone*", respectively. Similarly, Rules 5 and 6 identify *Mstream*-traffic by looking for the pattern "*newserver*" in UDP-payloads arriving at port 6838 and ">" in TCP-payloads originating from port 12754. As Rules 3-6 inspect specific ports, they are certainly vulnerable to dynamic port assignment. Finally, Rules 7 and 8 attempt to capture *Stacheldraht* traffic by inspecting ICMP-echo-reply messages. Rule 7 looks for *"spoofworks"* and field *icmp_id* with value 1000; while Rule 8 searches for *"ficken"* and field *icmp_id* with value 667. When inspecting the traffic of Table 12, *Snort-Inline* identifies only a small fraction of *Stacheldraht* packets; any control-related traffic escapes detection as well.

5.3 *DDoS Container* Accuracy in Classifying Traffic

To compare the *DDoS Container* accuracy in classifying *DDoS* traffic versus other options such as the open-source *Snort-Inline* IDS/IPS, we conduct a wide range of tests with both implementations and establish their false positive/negative rates. While forming an ICMP-based *TFN2K* attack network with $attacker_1$, *TFN2K-handler*$_2$, *TFN2K-agent*$_1$ and $victim_1$ of Figure 7, we use both *DDoS Container* and *Snort-Inline* to detect malicious traffic; the latter predominantly exploits rule 2 of Table 17. We determine false negative rates for both systems by having the attacker issue the command *"tfn -P ICMP -c cmd_id -i parameter"* with flags *-c* and *-i* indicating specific command identifier and corresponding parameter(s). Table 18 shows a number of such commands with column *plain text* indicating the handler-generated instructions on clear text before CAST encryption and base-64 encoding; *plen* indicates the message size for the plain text and *trail* shows the length of the trailing sequence of *A*s. We can see that *Snort-Inline* succeeds in detecting

#	*cmd*	*plain text* message	*plen*	*trail*	*Snort*	*DDoS Container*
1	-c 0	+d+0	4	5	negative	alert
2	-c 2 -i 64	+b+64	5	6	negative	alert
3	-c 3 -i 128	+a+128	6	8	negative	alert
4	-c 3 -i 1024	+a+1024	7	9	negative	alert
5	-c 4 -i 192.168.5.37	+e+192.168.5.37	15	20	alert	alert
6	-c 5 -i 192.168.5.37 -p 10	+g+10: +f+192.168.5.37	22	8	negative	alert
7	-c 6 -i 192.168.5.141	+h+192.168.5.141	16	21	alert	alert
8	-c 7 -i 192.168.5.37@192.168.5.141	+i+192.168.5.37@192.168.5.141	29	17	alert	alert
9	-c 8 -i 10.0.0.1@10.0.0.1	+k+1.0.0.1@10.0.0.1	47	4	negative	alert
10	-c 9 -i 1.0.0.1@1.0.0.2	+j+1.0.0.1@1.0.0.2	18	3	negative	alert
11	-c 10 -i "ls –almost-all -c"	+l+ls –almost-all -c	50	7	negative	alert
12	-c 10 -i "ls –directory -a -k"	+l+ls –directory -a -k	52	9	negative	alert

Table 18: False negatives generated while testing *TFN2K* sessions

sessions that have more than ten trailing *A*s which essentially implies that the *Snort-Inline* might create 9/21 false negatives if the length of the trailing sequence is uniformly distributed. Although *DDoS Container* correctly detects all twelve sessions of Table 18, *Snort-Inline* fails to generate alerts in nine instances.

We subsequently examine the generation of false positives by both tools and through six tests outlined in Table 19. Initially, we create a trace by generating traffic using the command *ping -p 41414141 192.168.5.141* where the *-p* flag forces the packing of ICMP echo request payload with pattern "41414141" or alphanumeric string "AAAA". In response, node 192.168.5.141 creates an ICMP echo reply whose sample is shown as the first case in Table 19. With the help of Figure 8 testbed, we inject the *ping*-trace without any modification to *DDoS Container*/*Snort-Inline* and both generate no alert. In *DDoS Container*, this is due to the fact that the payload of ICMP echo replies are not base-64 encoded and in *Snort-Inline* because the field *icmp_id* of the reply is non-zero and fails to satisfy rule 2. In the second test of Table 19, command

#	*cmd*	*IP payload*	*description*	*Snort*	*DDoS Container*
1	ICMP echo reply	ICMP header: |00 00 D2 F8 56 16 00 00| ICMP payload: |78 B3 27 42 16 DD 02 00 41 41 41 41 41 41 41 41 41 41 41 41 ...|	icmp_type: 0; icmp_code: 0; icmp_id: 0x5616; icmp_seq: 0; data: 8 binary bytes + "AA...";	no	no
2	replace "icmp_id" with 0	ICMP header: |00 00 21 0F 00 00 00 00| ICMP payload: |78 B3 27 42 16 DD 02 00 41 41 41 41 41 41 41 41 41 41 41 ...|	icmp_type: 0x0; icmp_code: 0; icmp_id: 0; icmp_seq: 0; data: "AAAAAAAAAA ...";	positive	no
3	replace first 8 bytes with A	ICMP header: |00 00 00 00 00 00 00 00| ICMP payload: |41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ...|	type: 0; code: 0; icmp_id: 0; icmp_seq: 0; data: "AAAAAAAAAA ...";	positive	no
4	replace first 8 bytes with base-64 code	ICMP header: |00 00 29 0F 00 00 00 00| ICMP payload: |42 53 47 41 56 6A 62 59 41 41 41 41 41 41 41 41 41 41 41 41 ...|	type: 0; code: 0; icmp_id: 0; icmp_seq: 0; data: "BSGAVjbYAAA...";	positive	no
5	replace first 15 bytes with base-64 code	ICMP header: |00 00 29 0F 00 00 00 00| ICMP payload: |67 47 79 34 31 49 2B 69 69 5A 74 36 4E 73 52 41 41 41 41 41 ...|	type: 0; code: 0; icmp_id: 0; icmp_seq: 0; data: "gGy41I+iiZt6NsRAAA...";	positive	no
6	replace entire payload with base-64 code but put ten As randomly	ICMP header: |00 00 29 0F 00 00 00 00| ICMP payload: |54 47 7A 4D 38 6D 58 53 46 4B 4F 38 4A 7A 44 70 56 52 49 66 ...|	type: 0; code: 0; icmp_id: 0; icmp_seq: 0; data: "TGZM8mXSFKO8Jz ...";	positive	no

Table 19: Test cases for the evaluation of false positives

icmp_field icmp_id 0 is applied to all ICMP messages before replayed by the tester, forcing the *icmp_id* filed of the ICMP header to become zero. Here, *Snort-Inline* mistakenly identifies the ICMP echo reply as *TFN2K* traffic by matching rule 2, while *DDoS Container* raises no alert as the message is not base-64 encoded.

In test 3, we change the first eight bytes of the ICMP payload with a sequence of A while in case 4, we change the first eight bytes of the ICMP payload to random characters in the range [A-Z, a-z, 0-9, +/], the legitimate base-64 code in *TFN2K*. Again, *Snort-Inline* flags both cases as *TFN2K* traffic because sequences of more than ten As appear in the payload. In case 5, we replace the first 15 bytes with base-64 code while in case 6 we replace the entire payload with base-64 code. Although *DDoS Container* considers the above two cases normal traffic, *Snort-Inline* generates false positives. Obviously, in the test cases we perform, *Snort-Inline* has a false positive rate of 5/6. While carrying out tests with the entire range of *DDoS* attack tools investigated in this paper, we establish that *DDoS Container* creates neither false positives nor negatives in contrast to *Snort-Inline*.

5.4 Sensitivity to Diverse *DDoS* Flooding Attacks

In order to inundate a network with heavy traffic, *DDoS* tools often create floods by using low-level interfaces such as raw sockets; these interfaces bypass protocol restrictions and require tools to craft protocol headers for the created IP,TCP,UDP and ICMP packets. The anticipated high packet rate and expected voluminous traffic necessitate that tools use simplified techniques in creating flooding attack packets. Figure 9 shows the assignment of source and destination ports for packets in a *Stacheldraht* UDP-flood attack. Both source and destination ports of packets take values in range [0, 10,000] and for the first attack packet, its corresponding source and destination ports are set to 9998 and 2. In subsequent packets, the source and destination ports are respectively decremented and incremented by one yielding their sum invariant to 10,000, an obvious signature. *TFN2K* UDP-flood packets demonstrate similar behavior as Figure 10 depicts. Source

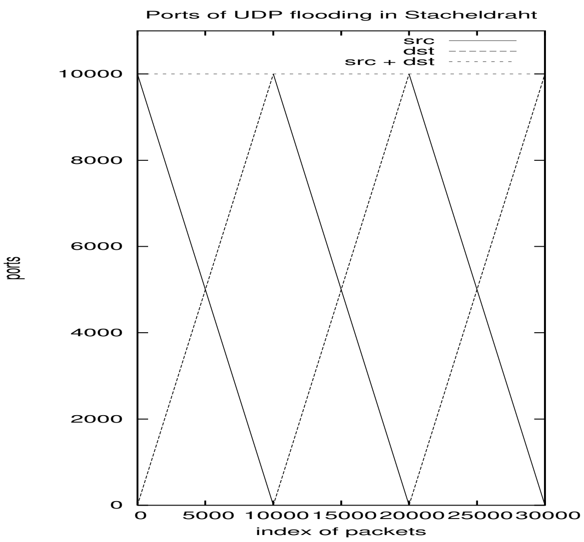

Figure 9: Source and destination ports in a *Stacheldraht* UDP-flood attack

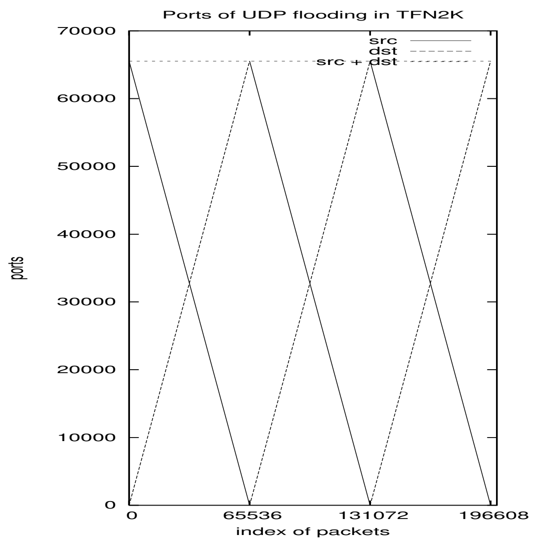

Figure 10: Source and destination ports in a *TFN2K* UDP-flood attack

and destination ports are initialized to 65,534 and 2 respectively for the first packet and their values keep changing in single unit steps while their numeric sum remains constant to 65,536. Table 20 shows a segment of *Mstream*-generated TCP-flood packets with unique characteristics; the packets have fixed size of 54 bytes from which 14 are for the Ethernet header, 20 for the IP header, and 20 for TCP header. Moreover, the *ip_tos* and *ip_flags* fields remain constant to values 8 and 0 respectively and *ip_id* is stepwise incremented as discussed in [13]. In the TCP-header, *tcp_flag* and *tcp_win* are set to *ACK*, and 16,384 while *tcp_sport* and *tcp_seq* are incremented by one when represented in host order instead of network order.

#	*src IP*	*src port*	*dst port*	*ip_id*	*tcp_flags*	*tcp_seq*	*tcp_ack*	*description*
directions of all packets: daemon (D): 192.168.5.141 → victim (V): 192.168.5.37;								
1	122.141.239.55	0x0AE7 (2791)	24035	0xAD41	ACK	0xBD6B0B00	0	pkt1: random ip_id, src_port, *dst_port, and tcp_seq*
2	214.0.67.96	0x0BE7 (3047)	42903	0xAE41	ACK	0xBE6B0B00	0	incremented ip_id in host-order; i.e., pkt1:0xAD41 → pkt2:0xAE41
3	176.51.61.100	0x0CE7 (3303)	64241	0xAF41	ACK	0xBF6B0B00	0	incremented src_port in host-order; i.e., pkt2:0x0BE7 → pkt3:0x0CE7
4	21.203.199.59	0x0DE7 (3559)	19041	0xB041	ACK	0xC06B0B00	0	incremented tcp_seq in host-order; i.e., pkt3:0xBF6B0B00 → pkt4:0xC06B0B00
5	157.170.14.59	0x0EE7 (3815)	29847	0xB141	ACK	0xC16B0B00	0	random source IPs for all packets
6	153.134.240.102	0x0FE7 (4071)	9172	0xB241	ACK	0xC26B0B00	0	random dst ports for all pkts
7	164.181.244.13	0x10E7 (4327)	44259	0xB341	ACK	0xC36B0B00	0	ACK in tcp_flag is set for all pkts
8	152.219.249.44	0x11E7 (4583)	42114	0xB441	ACK	0xC46B0B00	0	however, tcp_ack is 0 for all pkts
9	231.222.111.123	0x12E7 (4839)	40428	0xB541	ACK	0xC56B0B00	0	other peculiarities: fixed ip_flags (zero);
10	4.191.45.57	0x13E7 (5095)	23360	0xB641	ACK	0xC66B0B00	0	fixed packet size (40 bytes, excluding the header);
11	98.92.3.96	0x14E7 (5351)	7174	0xB741	ACK	0xC76B0B00	0	fixed tcp_win (16384 bytes);
12	173.81.218.80	0x15E7 (5607)	59842	0xB841	ACK	0xC86B0B00	0	fixed ip_flags (zero);
13	159.98.164.20	0x16E7 (5863)	52641	0xB941	ACK	0xC96B0B00	0	fixed ip_tos (8) and ip_ttl (255)

Table 20: *Mstream*-generated TCP-flood

Here, we ascertain the effectiveness of our *DDoS Container*, by replaying various types of traffic with known foreground characteristics. For brevity, we use the traffic of Figures 9 and 10 as well as that of Table 20 to feed the testbed of Figure 8. We inject attack mixed with normal traffic into our *DDoS Container* and vary the replay speed so that the attack intensity is adjusted. We form diverse types of workloads by mixing different foreground traffic and background traffic; the former consists of *Stacheldraht* UDP-flood, *TFN2K* UDP-flood, or *Mstream* TCP-flood attacks while the latter is FTP traffic. Various traffic templates can be defined to help assess the effectiveness of our *DDoS Container* in detecting flooding attacks; a number of such templates are shown in Table 21. A threshold expressed in packets-per-second (pps) and shown as *thrd* complements the definition of each template and indicates the intensity of the traffic above which the *DDoS Container* should generate an alert.

Templates 1–6 designate traffic patterns for *Stacheldraht* UDP floods. As the condition *dst_ip=same* indicates, template 1 clusters all packets with same destination IP; this template mimics the way the vast majority of IDSs/IPSs operate using pure destination-based patterns to detect floods. The designation *(dst_ip=same)&&(dst_port inc)* of template 2 identifies a stream in which the destination IPs remain the same but the destination ports are numerically increasing; template 3 outlines a similar pattern but inspects for decreasing source ports. Templates 4 and 5 identify traffic whose packets comply with the condition *src_port+dst_port=10,000* and show the same destination address, while template 6 exclusively uses the designation *src_port+dst_port=10,000*. As templates 1–5 use the destination address to cluster traffic, they mostly reflect the way traditional flood detection methods work. In contrast, template 6 is destination-address-free and we expect it to be more robust in dealing with floods. Templates 7–12 are formed to detect *Mstream* TCP-flood attack packets. We include template 7 to establish a comparison on detection accuracy between our *DDoS Container* and pure destination-based detection methods. On the other hand, template 12 groups all packets together that have acknowledgment number of zero and their TCP-window size is 16384.

By mixing flooding attack traffic generated by *Stacheldraht*, *TFN2K*, or *Mstream* and some attack-free

#	*template*	*thrd*	*C1*	*C2*	*C3*	*C4*	*C5*	*C6*	*C7*	*C8*	*C9*	*C10*
traffic created by *Stacheldraht* or *TFN2K* UDP flooding attacks												
1	(dst_ip = same)	100	neg	alert	alert	alert	neg	alert	alert	alert	no	pos
2	(dst_ip = same) && (dst_p inc)	100	neg	alert	neg	neg	neg	neg	neg	neg	no	no
3	(dst_ip = same) && (src_p dec)	100	neg	alert	neg	neg	neg	neg	neg	neg	no	no
4	(dst_ip = same) && (src_p + dst_p = 10,000) (change 10,000 to 65,536 for *TFN2K*)	100	neg	alert	neg	neg	neg	alert	neg	neg	no	no
5	(dst_ip = same) && (src_p + dst_p = 10,000)	2	alert	alert	alert	alert	alert	alert	alert	alert	no	no
6	(src_p + dst_p = 10,000)	2	alert	alert	alert	alert	alert	alert	alert	alert	no	no
traffic created by *Mstream* TCP flooding attacks												
7	(dst_ip = same)	100	neg	alert	alert	alert	neg	alert	alert	alert	no	pos
8	(dst_ip = same) && (ip_id inc)	100	neg	alert	neg	neg	neg	neg	neg	neg	no	no
9	(dst_ip = same) && (tcp_ack = 0)	100	neg	alert	neg	neg	neg	alert	neg	neg	no	no
10	(dst_ip = same) && (tcp_ack = 0)	2	alert	alert	alert	alert	alert	alert	alert	alert	no	no
11	(dst_ip = same) && (tcp_win = 16384)	2	alert	alert	alert	alert	alert	alert	alert	alert	no	no
12	(tcp_ack = 0) && (tcp_win = 16384)	2	alert	alert	alert	alert	alert	alert	alert	alert	no	no

Table 21: Sensitivity results for *Stacheldraht* UDP, *TFN2K* UDP, and *Mstream* TCP flood attack workloads

FTP traffic, each with different intensities, we generate a number of scenarios indicated as cases *C1* to *C10* in Table 21. In all cases, both foreground attack and background traffic have as their destination the primary victim's IP address. In cases *C1* and *C2*, we inject pure attack traffic with rates of 2 and 100 pps respectively. In *C3*, foreground traffic of 2 pps is mixed with 98 pps attack-free traffic; in *C4*, the foreground and background traffic streams have intensity rates of 98 pps and 2 pps respectively. Cases *C5* and *C6* are similar to *C1* and *C2* with the only difference that while replaying the packets the replay-order of some consecutive foreground packets is swapped; in this spirit, *C7* and *C8* are similar to *C3* and *C4* with different order for some of the packets. Cases *C9* and *C10* inject only background traffic with intensity rates of 2 pps and 100 pps. In Table 21, "no" and "alert" indicate that our *DDoS Container* correctly classifies the injected traffic as either legitimate or malicious while "pos" and "neg" show false positives and negatives.

Table 21 outlines the overall behavior of our *DDoS Container* under the aforementioned diverse traffic settings. The key observation is that when *thrd* is set to 2 pps requirement, the *DDoS Container* under templates 5, 6, 10, 11, and 12 produces the correct results as it either alerts after observing two malicious packets or correctly identifies background traffic shown with "no". Destination-based templates 1 and 7 create false positives in pure background traffic of *C10* as they only inspect destination IP addresses of the incoming packets and the injected background traffic shares the same destination address, therefore producing false positive. On the other hand, templates 2, 3, and 8 miss the flooding attacks in *C6* simply because the swapping replay orders of attack packets destroy the monotonic increment or decrement relationships existing in source ports, destination ports, or IP identifiers (i.e., *ip_id*), causing the observed attack intensities of these templates to be lower than the real attacks. The setting of threshold value is critical as templates 9 and 10 show; when *thrd* is set to 2 pps, the *DDoS Container* is successful in accurately detecting all attack traffic streams while it fails in part to accomplish this when the threshold is set at 100 pps. The higher threshold simply misses attacks with lower intensities. In this regard, it is desired to set the threshold to the lowest possible values. However, decreasing thresholds indiscriminately may generate false positives. For example, if we change the threshold in template 1 from 100 to 2 pps, the *DDoS Container* creates a false positive for *C9*.

Having lower values for the *thrd* without creating false positives also implies that there is better utilization of computing resources. For instance templates 5 and 6 have the same detection accuracy for all tests, while template 6 entails less constraints and thus, requires less memory. Since templates 6 and 12 demonstrate not only superb robustness in detecting flooding attacks but also low memory consumption, they are used in the FortiGate-800 device when deployed in production.

5.5 Discussion on the *DDoS Container* Performance

A thorough testing of FortiGate-800 equipped with our *DDoS Container* has been recently conducted by *NSS*, an independent IPS testing organization [27]. At its rated speed of 400Mbps, the FortiGate-800 detects and blocks all attacks under various test-load conditions. Basic latency figures were well within acceptable limits for all traffic loads and with all packet sizes; they ranged from 249μs for traffic of 100Mbps consisting of 256 byte packets to 280μs with 400Mbps with 1000 byte packets. With 40Mbps of *SYN* flood-traffic generated by the *TFN2K DDoS* tool, FortiGate shows latency of 188μs with 256 byte packets and 216μs with 1000 byte packets [27]. The HTTP response time, defined as the time interval between request transmission and reply arrival, for Web page access increased only slightly during *SYN* flood tests from 214μs under normal load to 219μs with the SYN flood. Even under eight hours of extended attacks comprising of millions of exploits mixed with genuine traffic, FortiGate-800 continued to block 100% of attack traffic while allowing all legitimate traffic pass through. Moreover, our *DDoS Container* was able to correctly identify all "false negative" and "false positive" test cases and demonstrated excellent resistance to known evasion techniques including IP fragmentation and TCP segmentation. Tests by *ICSA-Laboratories* also offered similar observations while testing FortiGate equipped with our *DDoS Container* module [33]. Overall, our own experiments and those of independent testers reveal the high detection/prevention accuracy of our *DDoS Container*; the latter also impacts in a minimal manner both network latency and system throughput.

6 Related Work

DDoS attacks have been long recognized as a major threat to the Internet [24, 44, 22, 18, 19], and [42] helped establish that most sites suffer numerous daily *DDoS*attacks while occasionally experiencing intensive traffic flooding of up to 500,000 pps. *DDoS* tools including *Trinoo*, *TFN* and *Shaft* have been dissected and analyzed to help create counter-measures [40]. *DNS amplification attacks* exploit the "open-resolvers" in the *DNS* system and bombard with over-sized UDP-*DNS*-queries targeted sites [60]. In general, defense mechanisms can be classified as preventive, reactive, and tolerant. Preventive mechanisms attempt to eliminate the conditions necessary for the formation of *DDoS* attacks in their various stages, such as vulnerability identification, site penetration, code implantation, and attack launching [21, 58]. Reactive mechanisms continually monitor the behavior of programs and/or network activities, trying to identify possible attacks and then generate alerts (e.g., in IDSs) or eliminate them (e.g., in IPSs) [26, 22]. In order for legitimate traffic to be handled even in light of an ongoing *DDoS* attack, tolerance mechanisms featuring resource redundancy, bandwidth-rate limitation, and dynamic system re-configuration have been proposed [3, 54].

Although it is critical that the origin of an attack be identified for accountability d and forensic analysis purposes [40, 56, 64], such an identification is not always feasible due to address spoofing [21]. Tracing systems including *ICMP Traceback*, *IP Traceback*, and *CenterTrack* are designed to address this issue but their success remains limited as they often lead to zombie processes instead of the real instigators of *DDoS* attacks [6, 57, 48, 53, 18]. Similarly, the *Sleepy Watermark Tracing (SWT)* approach uses watermarks to uniquely identify connections [62]. *SWT* could be used with routers so that the latter inject pertinent information (i.e., watermarks) to involved network–applications. By correlating incoming/outgoing packets, *SWT* could help accurately determine a path-flow; evidently, this scheme is only feasible should applications be watermark-aware, all routers are trustworthy, and there is no link-to-link encryption.

By monitoring traffic, utilities including *Cisco IOS QoS*, *NetFlow*, *Cflowd*, *FlowScan*, *NetDetector* and *RRDtool* help both detect and visualize abnormal behavior but more importantly provide early-warning to potential *DDoS* attacks [58, 19, 40]. Routers with functionality of ingress/egress filtering ensure that the sources/destinations of data streams comply with adopted policies [21]. More specifically, ingress filtering examines every incoming packet to a network for the validity of its IP source-address; similarly, egress filtering checks all outbound packets to ensure their legitimate addresses [40]. The *unicast reverse path forwarding* (*uRPF*) mechanism in some routers ascertains the validity of a packet if the latter arrives through one of the "best" paths available [58, 40]; although useful, *uRPF* can only mitigate the intensity of a *DDoS* attack. The establishment of demilitarized zones (*DMZ*) [14], the use of proxies to manage TCP-based connections [50], as well as the deployment of firewalls with port or service based traffic filtering [58] may lessen the effectiveness of *DDoS* attacks; unfortunately, such measures are ineffective toward attacks launched internally and their "all-or-nothing" policy may render both legal and useful facilities such as *ping* and *traceroute* unavailable if *ICMP*-messages are not allowed to enter/leave such a guarded-network.

Resource-intensive TCP SYN-flooding and packet fragmentation attacks are often dealt with "client puzzle" protocols; for each client request, servers pose "puzzles" that are time-dependent and feature information unique to servers under heavy traffic [12, 63, 40]. A server allocates resources for a connection only if the initiator correctly solves the puzzle; forcing the attacker to commit significant resources to sustain the intensity of an attack [34, 39, 19, 2]. Should filtering be impossible, network topology reconfiguration including "back-holing" of victims may reduce *DDoS* damages [66]. Rate-limiting mechanisms set thresholds for bandwidth consumption for various types of traffic, especially those identified as malicious [37, 41]. By removing traffic ambiguities, protocol normalization or scrubbing techniques also help mitigate the effectiveness of attacks [38]. Auditing tools help discover *DDoS* agents and/or handlers by identifying changes in file systems and critical system configurations [59] or locating unique patterns in programs, especially binaries [11]. With the help of such auditing strategies, host-based tools such as *Tripwire* may detect malicious *DDoS* codes, while network-based auditing tools such as *ddos_scan* can detect the existence of handler-agent communications by searching for specific patterns in ongoing network traffic [40]. Unfortunately, both host and network-based tools become ineffective when *DDoS* attacks utilize techniques such as dynamic port allocation, message encryption, and information compression.

Reactive mechanisms to *DDoS* attacks mostly entail pattern matching and behavior anomaly analysis. Patterns of known attacks are often stored in a signature database used to identify *DDoS* activities [47]. When traffic at a site deviates from what is deemed as "normal", it is flagged and counter-measures are taken [66, 37, 41]. In this regard, there is a wide range of tools which successfully address mostly individual aspects of *DDoS* attacks. For example, the *CaptIO* through the use of rules can detect ICMP-floods and subsequently limits the bandwidth consumption of such traffic types; it fails however to identify either spoofed or multi-source attacks. Similarly, the *Top Layer AppSwitch 3500* can counter attacks such as *land*, *smurf*, *fraggle*, and *UDP*-bombs but is unable to handle ICMP and SYN-floods coming off random source addresses [27].

By applying temporal quantization and Granger causality test to the MIB databases from multiple domains, precursors to *DDoS* attacks can be extracted, which may indicate imminent attacks [7, 8]. Unfortunately, in order to conduct causality analysis, MIB databases in both attacker and primary victim machines should be accessible [7] which may be of limited value in pragmatic settings. In addition, information on MIB variables from different domains should be exchanged in real-time [7, 8]; this may not be feasible especially when an intense *DDoS* attack is under way. Furthermore, the MIB variables used including *TCPInSegs*, *UDPOutDatagrams*, and *ICMPInMsgs*, are of coarse granularity making it difficult to distinguish among different *DDoS* attacks. Finally, this causality analysis is based on abnormal traffic behavior such as flooding; therefore, it is applicable to communications between zombies and primary victims only and is ineffective for messages exchanged between attackers and handlers as well as handlers and agents. Our work in this paper builds on the abovementioned efforts and our main objective is to provide not only a pragmatic and comprehensive but also an extensible framework capable of effectively detecting/preventing malicious traffic among attackers, handlers, zombies, and primary victims in a wide range of contemporary *DDoS* attacks.

7 Conclusions and Future Work

By penetrating into a large number of machines through security flaws and vulnerabilities and stealthily installing malicious pieces of code, a distributed denial of service (*DDoS*) attack constructs a hierarchical network and launches coordinated assaults. By exhausting the network bandwidth, processing capabilities and other resources of victims, *DDoS* render services unavailable to legitimate users. As *DDoS* toolkits use multiple mechanisms, it is in general very challenging to identify and/or prevent such attacks. Although trace methods and ingress/egress filtering techniques are used to locate agents and/or zombies in intermediate network nodes, they are complex to implement and difficult to deploy as they frequently call for global cooperation. Elements of hierarchical *DDoS* attack networks use dynamic TCP/UDP ports and source address spoofing to hide attackers and thwart their tracing. Moreover, one-way communication channels, encrypted messages, and the use of evasive techniques render conventional IDSs/IPSs ineffective as the latter typically resort to specific pattern matching and fixed-port traffic identification.

In this paper, we propose a comprehensive framework, the *DDoS Container*, whose main objective is to overcome the deficiencies of existing approaches. The *DDoS Container* uses network-based detection methods and operates in inline fashion to inspect and manipulate passing traffic in real-time. By tracking connections established by both *DDoS* attacks and normal applications, our framework maintains state information for each session, conducts stateful inspection, and correlates data among sessions. *DDoS Container* performs stream re-assembly and dissects the resulting aggregations against protocols followed by known *DDoS* systems facilitating the identification of such malicious activities. The use of deep inspection and behavior analysis enhance *DDoS Container*'s detection accuracy when it comes to encrypted *DDoS* traffic. Our framework can take a number of steps in handling detected *DDoS* traffic including alerting, packet blocking and proactive session termination. Experimentation with the prototype of our *DDoS Container* demonstrates its effectiveness in a large number of settings and establishes its efficiency.

We intend to follow up this work by pursuing three objectives: (i) maintain the currency of our *DDoS Container* framework by incorporating analyzers for emerging and new *DDoS* strains, (ii) provide mechanisms to exchange information among various *DDoS Container*s deployed in different locations so that event correlation in targeted network regions can be conducted; here, the goal is to detect *DDoS* attacks with victims spanning multiple domains or launched with very light intensity rates, and (iii) explore the integration of our *DDoS Container* with other security systems including firewalls, anti-virus, host-based IDSs/IPSs, and anti-malware programs to more effectively combat aggregate malicious activities resulting from the mixture of *DDoS* and popular worms.

Acknowledgments: We are very grateful to the reviewers for their meticulous comments that helped us improve the presentation of our work. We are also thankful to Peter Wei of Fortinet, Inc. for discussions on the framework presented in this manuscript and Qinghong Yi, Gary Duan, Ping Wu, Fushen Chen, Joe Zhu and Hong Huang for helping with parts of our implementation and testing effort.

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] T. Aura, P. Nikander, and J. Leiwo. DOS-Resistant Authentication with Client Puzzles. *Springer-Verlag, Lecture Notes in Computer Science*, 2133, 2001.

[3] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the 1999 USENIX/ACM Symposium on Operating System Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[4] M. Blaze, J. Ioannidis, and A.D. Keromytis. Toward Understanding the Limits of DDoS Defenses. In *Proceedings of the Tenth International Workshop on Security Protocols*, Cambridge, United Kingdom, April 2002.

[5] D. Brumley. Remote Intrusion Detection (RID). *http://www.stanford.edu/*, 2000.

[6] H. Burch and B. Cheswick. Tracing Anonymous Packets to Their Approximate Source. In *Proceedings of the 2000 USENIX LISA Conference*, pages 319–327, New Orleans, LA, December 2000.

[7] J. B. D. Cabrera, L. Lewis, X. Qin, W. Lee, and R. K. Mehra. Proactive Intrusion Detection and Distributed Denial of Service Attacks - A Case Study in Security Management. *Journal of Network and Systems Management*, 10(2):225–254, June 2002.

[8] J. B. D. Cabrera, L. Lewis, X. Qin, W. Lee, R. Prasanth, B. Ravichandran, and R. Mehra. Proactive Detection of Distributed Denial of Service Attacks Using MIB Traffic Variables - A Feasibility Study. In *In Proceedings of The Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM 2001)*, pages 609–622, Seattle, WA, May 2001.

[9] CERT Coordination Center. Smurf Attack. *http://www.cert.org/ advisories/CA-1998-01.html*, 1998.

[10] CERT Coordination Center. Trends in Denial of Service Attack Technology. *http://www.cert.org/ archive/pdf/DoS trends.pdf*, October 2001.

[11] National Infrastructure Protection Center. Advisory 01-014: New Scanning Activity (with W32-Leave.worm) Exploiting SubSeven Victims. *http://www.nipc.gov/ warnings/advisories/ 2001/01-014.htm*, June 2001.

[12] Y. W. Chen. Study on the Prevention of SYN Flooding by Using Traffic Policing. In *Proceedings of the Network Operations and Management Symposium, 2000 (NOMS 2000)*, pages 593–604, Honolulu, HI, 2000. IEEE/IFIP.

[13] Z. Chen, Z. Chen, and A. Delis. Analyzers for *DDoS* Attack Tools. Technical report, Athens, Greece, December 2005. Department of Informatics and Telecommunications, Univ. of Athens, http://www.di.uoa.gr/∼ad/analyzers.pdf.

[14] W.R. Cheswick, S.M. Bellovin, and A.D. Rubin. *Firewalls and Internet Security*. Addison-Wesley, Professional Computing Series, Boston, MA, second edition, 2003.

[15] D. E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[16] ComputerWorld. Microsoft Admits Defense Against Attacks Was Inadequate. *http://www.computerworld.com /software-topics/os/story /0,10801,57054,00.html*, Jan. 2001.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1997.

[18] D. Dean, M. Franklin, and A. Stubblefield. An Algebraic Approach to IP Traceback. In *Proceedings of the 2001 Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[19] C. Douligeris and A. Mitrokotsa. DDoS Attacks and Defense Mechanisms: Classification and State-of-the-Art. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 44(5):643–666, April 2004.

[20] Ethereal. Ethereal: Powerful Multi-Platform Analysis. *http://www.ethereal.com*, May 2005.

[21] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing. *Internet Engineering Task Force*, May 2000.

[22] M. Fullmer and S. Romig. The OSU Flowtools Package and Cisco Netflow Logs. In *Proceedings of the 2000 USENIX LISA Conference*, New Orleans, LA, December 2000.

[23] X. Geng and A. B. Whinston. Defeating Distributed Denial of Service Attacks. *IT Professional*, 2(4):36–41, July 2000.

[24] V.D. Gligor. A Note on the Denial of Service Problem. In *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, Oakland, CA, December 1983.

[25] V.D. Gligor. Guaranteeing Access in Spite of Distributed Service-Flooding Attacks. In *Proceedings of the Security Protocols Workshop*, Sidney Sussex College, Cambridge, UK, April 2003. Springer-Verlag.

[26] J. Green, D. Marchette, S. Northcutt, and B. Ralph. Analysis Techniques for Detecting Coordinated Attacks and Probes. In *Proceedings of USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 1999.

[27] NSS Group. Intrusion Prevention System (IPS) Group Test. http://www.nss.co.uk/ips/edition2/fortinet/, 2005.

[28] K. Hafner and J. Markoff. *Cyberpunk: Outlaws and Hackers on the Computer Frontier*. Simon and Scuster, New York, NY, 1991.

[29] H. M. Heys and S. E. Tavares. On the Security of the CAST Encryption Algorithm. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 332–335, Halifax, Nova Scotia, Canada, Sep. 1994.

[30] Fortinet Inc. Intrusion Prevention System. *Web Site*, May 2005.

[31] Computer Security Institute and Federal Bureau of Investigation. 2000 CSI/FBI Computer Crime and Security Survey. *Computer Security Institute publication*, March 2000.

[32] F. Kargl, J. Maier, and M. Weber. Protecting Web Servers from Distributed Denial of Service Attacks. In *Proceedings of 10th International World Wide Web Conference*, Hong-Kong, China, May 2001.

[33] ICSA Lab. Intrusion Prevention System (IPS) Test. http://www.icsalabs.com/, 2005.

[34] J. Leiwo, P. Nikander, and T. Aura. Towards Network Denial of Service Resistant Protocols. In *Proceedings of the 15th International Information Security Conference*, New York, NY, August 2000.

[35] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang. SAVE: Source Address Validity Enforcement Protocol. In *Proceedings of the IEEE INFOCOM International Conference*, New York, NY, June 2002.

[36] R. Love. *Linux Kernel Development*. Developer's Library Sams Publishing/Novel, second edition, 2005.

[37] R. Mahajan, S. Bellovin, S. Floyd, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM Computer Communications Review*, 32(3), July 2002.

[38] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and Application Protocol Scrubbing. In *Proceedings of the INFOCOM International Conference (3)*, pages 1381–1390, Tel-Aviv, Israel, March 2000.

[39] C. Meadows. A Formal Framework and Evaluation Method for Network Denial of Service. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, Mordano, Italy, June 1999.

[40] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall, ISBN: 0-13-147573-i, 2005.

[41] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the Source. In *Proceedings of the 10th IEEE International Conference on Network Protocols*, Paris, France, November 2002.

[42] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the 2001 USENIX Security Symposium*, Washington, D.C., Aug. 2001.

[43] R. Naraine. Massive DDoS Attack Hit DNS Root Servers. *http://www.esecurityplanet.com/ trends/ article/ 0,,10751 1486981,00.html*, October 2002.

[44] R. Needham. Denial of Service: An Example. *Communications of the ACM*, 37(11):42–47, November 1994.

[45] Fox News. Powerful Attack Cripples Internet. *http://www.foxnews.com/story /0,2933,66438,00.html*, April 2003.

[46] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *Proceedings of ACM SIGCOMM Conference*, San Diego, CA, August 2001.

[47] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *USENIX 13-th Systems Administration Conference – LISA'99*, Seattle, WA, 1999.

[48] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIG-COMM Conference*, pages 295–306, Stockholm, Sweden, August 2000.

[49] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C (2nd Edition)*. John Wiley & Sons, Inc., New York, 1996.

[50] C. Schuba, I. Krsul, M. Kuhn, G. Spafford, A. Sundaram, and D. Zamboni. Analysis of a Denial of Service Attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.

[51] Packet Storm Security. Wet Site. *http://packetstormsecurity.com*, 2005.

[52] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[53] D. Song and A. Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of the 2001 IEEE INFOCOM Conference*, Anchorage, AK, April 2001.

[54] O. Spatscheck and L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the 1999 USENIX/ACM Symposium on Operating System Design and Implementation*, pages 59–72, February 1999.

[55] S. M. Specht and R. B. Lee. Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures. In *International Workshop on Security in Parallel and Distributed Systems*, pages 543–550, San Francisco , CA, September 2004.

[56] L. Spitzner. *Honeypots: Tracking Hackers*. Addison Wesley, ISBN: 0321108957, 2002.

[57] R. Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *Proceedings of the 2000 USENIX Security Symposium*, pages 199–212, Denver, CO, July 2000.

[58] Cisco Systems. Unicast Reverse Path Forwarding. *Cisco IOS Documentation*, May 1999.

[59] Tripwire. Tripwire for Servers. *http://www.tripwire.com/products/servers/*.

[60] R. Vaughn and G. Evron. DNS Amplification Attacks. *http://www.isotf.org/news/DNS-Amplification-Attacks.pdf*, March 2006.

[61] P. Vixie. Extension Mechanisms for DNS (EDNS0). *Internet Engineering Task Force*, August 1999.

[62] X. Wang, D. S. Reeves, S. F. Wu, and J. Yuill. Sleepy Watermark Tracing: An Active Network-Based Intrusion Response Framework. In *Proceedings of the IFIP TC11 Sixteenth Annual Working Conference on Information Security: Trusted Information: The New Decade Challenge*, pages 369–384, 2001.

[63] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New Client Puzzle Outsourcing Techniques fro DoS Resistance. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 246–256, Washington, DC, October 2004.

[64] N. Weiler. Honeypots for Distributed Denial of Service. In *Proceedings of Eleventh IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2002*, pages 109–114, 2002.

[65] Wired.com. Yahoo on Trail of Site Hackers. *http://www.wired.com /news/business /0,1367,34221,00.html*, May 2003.

[66] J. Yan, S. Early, and R. Anderson. The XenoService – A Distributed Defeat for Distributed Denial of Service. In *Proceedings of the 3rd Information Survivability Workshop (ISW'00)*, Boston, USA, October 2000.