Catching the Middlebox: a Technique for the Detection of Intermediate Network Devices.

Luis MartinGarcia
Department of Telematic Systems Engineering.
Telecommunication Engineering School.
Universidad Politécnica de Madrid.
{luis.mgarcia@alumnos.upm.es}

***Abstract*—In spite of being one of the master guidelines in the design and the architecture of the Internet, the importance of the end-to-end argument has diminished over the years. Nowadays, most network configurations include some type of middlebox that manages or alters the traffic exchanged by the ends. Examples include firewalls, NATs, application layer gateways or load balancers. Although these devices often provide important functions that are essential to guarantee the security, efficiency or scalability of the network, their use may imply the violation of the end-to-end principle and introduce severe problems to some applications and network services. The detection of middleboxes presents unique difficulties, mainly because they are designed to be transparent to end nodes. However, the problem is not at all unapproachable. This paper presents a novel technique that allows the detection of intermediate devices between two end nodes, based on the differences found between the packets that were originally created by the sender, and the packets that were received by the other end.**

I. Introduction

The end-to-end principle has been one of the master guidelines in the architecture of the Internet. However, its importance has diminished over the years. Nowadays, in most network configurations, it is very common to find some type of intermediate device that manages or alters the traffic exchanged by the ends. Examples include firewalls, NATs, application-layer gateways or load balancers. Such devices, also known as *middleboxes*, typically provide important functions that are essential to guarantee the security, efficiency or scalability of a network, but their presence often violates the end-to-end principle and introduces severe problems to some applications and network services that were designed under the assumption that network traffic flows virtually unaltered from one end to the other. As a consequence, the correct operation of services designed according to the Client/Server model, or the relatively new peer-to-peer scheme, may require applications to have certain knowledge of the underlying transport and network layers, which constitutes another violation of an important principle, the independence of protocol layers.

Due to their transparent mode of operation, the presence of middleboxes is not easy to detect, particularly at the application layer. This may require users to be aware of the existence of intermediate devices and make the appropriate configuration adjustments in order to access or provide a particular network service.

Although the detection of middleboxes has a wide variety of applications, from network reconnaissance to the improvement of the user experience in the access to network services, there is a significant lack of research on the area. Nevertheless, the problem is not at all unapproachable. This paper presents a novel technique for the detection of intermediate devices, through the analysis of the differences between the network packets generated on one end of a communication, and the packets that were actually received at the other end.

The remainder of this paper is organized as follows. Section II discusses the concept of middlebox, describing some of their types, features and characteristics. Section III presents our contribution to the field, introducing a novel technique for the detection of middleboxes and the tools and methodology that we used to produce a working implementation of it. Section IV reports the results of our experiments with the implementation. Finally, section V presents our conclusions, and discusses open questions and future work.

II. Middleboxes

This section discusses the concept of middlebox, describing the most popular types and their possible features and characteristics.

A. Middlebox modeling

Given the wide variety of intermediate devices and the way they process and alter the packets that flow through them, it seems convenient to have a way to model them and express their characteristics in a formal manner. Some authors propose models to represent specific types of devices, like firewalls [11], while others develop generic models to represent network communication mechanisms [10]. However, it is [7], who made the best contribution to the field, presenting a specific model for the representation of intermediate devices, using a simple

and precise approach. In particular, the proposed model is composed by six elements, that are described by its authors as follows.

1) Interfaces and zones: a middlebox is composed by one or more physical network interfaces, each of which belongs to one or more logical network zones. A zone represents a packet entry and exit point from the perspective of middlebox functionality. A middlebox may process packets differently based on their ingress and egress zones.

2) Input preconditions: specify the types of packets that are accepted by a middlebox for processing, and are represented using a clause of the form $I(P, p)$, which is true if the headers and contents of packet p match certain pattern P.

3) State data: refers to all the information that a middlebox maintains about the flows and sessions that it processes.

4) Processing rules: model the core functionality of a middlebox. A processing rule specifies the action taken by a middlebox when a particular condition becomes true.

5) Auxiliary traffic: in addition to its core functionality of transforming and forwarding packets, a middlebox can generate additional traffic, either independently or when triggered by a received packet.

6) Interest and State Fields: the interest fields of a middlebox identify the packet fields of interest; in other words, the protocol fields that it analyzes or modifies. The state fields identify the subset of the interest fields used by the middlebox in storing and retrieving state. Although these fields can be deduced from the processing rules, they are explicitly presented in the model because they can highlight succinctly unexpected aspects of middlebox processing.

B. Features and characteristics

In addition to the previous model, it is possible to classify middleboxes into distinct groups based on certain characteristics. [6] proposes a set of eight variables that can identify a given intermediate device. Such variables are described as follows.

1) Protocol layer: specifies one or more protocol layers at which a middlebox operates.

2) Functionality (Explicit vs. Implicit): specifies whether the functionality provided by a middlebox is an explicit design feature of the protocols (such an SMTP relay) or an unforeseen add-on, possibly designed to operate transparently (like a NAT device).

3) Instances (Single hop vs. multi-hop): specifies how many instances of a middlebox can co-exist in the path between two end nodes. Typical values are 1, 2, 2n, or infinite.

4) Position (In-line vs. Call-out): specifies the position of a middlebox in the network. Middleboxes may be place in-line, on the data path, or may be located out of it, requiring an explicit call-out triggered by some event.

5) Goals (Operation vs. Optimization): specifies whether the middlebox performs an essential function, without which end nodes can not communicate as desired, or only an optimization.

6) Alteration capabilities: specifies whether the middlebox performs forwarding functions that leave the packets virtually unaltered, or functions that alter the packets in a non-trivial way or create side effects for the end hosts. Examples of the former include switches or routers. Examples of the latter include firewalls, NATs, or proxies.

7) State management (Hard vs. Soft State): specifies whether, upon a sudden lost of state information, sessions continue to run, either normally or in some kind of degraded mode (soft state), or fail and need to be re-established from scratch (hard state).

8) Failure Handling (Fail-over vs. Restart): specifies whether, in the event of a hard state middlebox failure, the session is redirected to an alternative box that has a copy of the state information, or it is forced to abort and restart.

C. Types

From the model and characteristics introduced in the previous sections, it seems clear that there is a wide variety of intermediate devices, to the extend of their purpose, as well as their impact on the networks where they are deployed. This section presents a list of the most common types of middleboxes [6].

1) Network Address Translators (NATs): a NAT is a device that alters IP datagrams, modifying their source and destination address. This is often done to facilitate communication between hosts that use private, non routable, IP address spaces, and hosts with public IP addresses.

NAT devices are not compatible with application layer protocols that have dependencies with underlying IP addresses. Examples include FTP or SIP. For this reason, NATs are often combined with application layer gateways, which are capable of making the necessary changes to enable communications.

There is an special type of NAT systems, called NAP-PT (NAT with Protocol Translator) [15], which transforms IPv6 into IPv4 datagrams and vice versa. However, its utilization has been deprecated by the IETF, so they may not enjoy a wide deployment [5].

2) Firewalls: a firewall is a system that is located between two or more network segments and has the ability to analyze the traffic that reaches its interfaces, denying or authorizing its entry, according to a pre-established security policy. Firewalls are probably one of the most common middleboxes in IP networks.

In general, the traffic that traverses the firewall does not suffer special alterations. However, when a firewall drops a packet, it does not inform the original sender of such event. This does not only cause connectivity problems, but also makes it very difficult for the sender to diagnose the problem.

Firewalls often operate at the network and transport layer. Nevertheless, some specialized firewalls may also operate at the application layer, making decisions based on operation types or compliance to the standards [12].

3) SOCKS Gateways: a SOCKS gateway is a device that acts as an intermediary between a client and a server, typically in scenarios where a firewall blocks direct communication between the two parties. These gateways use the SOCKS protocol [9], which operates at the application layer (OSI session layer), and are accessible through TCP port 1080.

4) Tunnel Endpoints: they are the devices that create or manage communication tunnels. They offer data encapsulation and transport services between two points in a network. Tunnel endpoints alter the packets that traverse the tunnel, first adding and later removing some protocol headers. Although packets that enter one end of the tunnel leave the other end unaltered, the presence of the tunnel may affect the end-to-end principle, as the transmitted packets could have experienced a different per-hop treatment (QoS, routing, etc.), if the communication had not been tunneled.

5) Traffic Handlers: also known as packet classifiers, markers or schedulers, are devices that classify, schedule or tag the packets that traverse them, with the intent of adapting network traffic to a specific policy. In particular, these devices may tag packets to provide differentiated services, alter their order or time sequences, or drop a number of them based on different parameters and measurements. Although their presence affects the end-to-end principle, they do not introduce significant changes to the best effort nature of the Internet.

6) Load balancers: a load balancer is a device that redirects traffic destined to a particular network service, to the appropriate physical or logical server, based on the load conditions of the set of servers that provide the same service. Load balancers can operate at the IP level, rewriting destination addresses, or at the application layer, making the appropriate changes to application data or providing the necessary redirection mechanisms.

7) Application Layer Gateways: an application layer gateway (ALG) is a device that is able to process and modify application layer data found in network packets that traverse it. Typically, their purpose is to adapt application layer protocols to changes in other layers like those performed by NAT devices. Other uses include performing translations between different application protocols or different versions of a protocol, generating usage statistics or keeping event logs.

8) Transcoders: a transcoder is a device that performs application layer data conversions. They are mainly used in communications where the sender is unable to provide data in a suitable format for the receiver. Examples of transcoder use include conversion of voice data between VoIP and cellular voice, bitrate video conversion or image scaling.

9) Proxies: a proxy is a device that simultaneously plays the role of a server and a client. They act as clients of a network service, making requests on behalf of the real client, and act as servers for such client, forwarding the data provided by the real server. A proxy may be used explicitly (clients are aware of its presence and choose to access network services through it), or implicitly (clients ignore its existence and the proxy intercepts communications transparently). In both cases, the traditional client/server network flow, is divided in two sub-flows, one between the client and the proxy, and the other between the proxy and the server.

10) Caching Proxies: a caching proxy is a device that monitors client/server application layer sessions and stores (caches) server responses in order to replay them if the client issues identical requests in the future. Its purpose is to improve response times and to prevent redundant communications.

11) Performance Enhancing Proxies (PEPs): a PEP is a device that is intended to improve end-to-end performance of some network protocol. Typically PEP devices work in pairs (like tunnel endpoints), breaking end-to-end connections into multiple parts, using different parameters or even different protocols, for each segment of the communication. PEPs are very popular in TCP/IP networks with satellite links, as TCP does not perform well on links with large bandwidth-delay products.

12) Redirecters: a redirecter is a device that intercepts communications initiated by a client and redirects them to another server that, using the same protocols, provides a different service than the one expected by the client. Redirecters are often used in networks that, in order to be accessed, require users to pay a fee, accept some legal conditions or provide authentication details. Perhaps the most common case are HTTP redirecters placed in airports, hotels or universities, that do not let users access the Internet until they have completed certain steps.

13) Intrusion Detection Systems (IDS): an IDS is a device that monitors network traffic in order to detect signs of an attack or a violation of a per-established security policy. Due to their passive nature, traditional IDS devices are not considered middleboxes. However, there is a special type of IDS called *in-line IDS* (or Intrusion Prevention System), that is placed at some intermediate point in a communication path, and have the ability to block traffic when a particular event is detected. Although, in practical terms, in-line IDS devices could be considered application layer firewalls, they have been explicitly included in the list, due to their popularity in the security field.

III. Middlebox detection

The detection of intermediate devices has not been studied thoroughly, even though it is of great importance for many applications. Some authors have unsuccessfully tried to define certain requirements for their discovery [16]. Others have attempted to introduce mechanisms to allow middleboxes to explicitly signal their presence upon request, either through the standardization of new TCP options [17], or using new protocols to interact with such devices [14]. However all proposed techniques require intermediate devices to be aware of the semantics of the protocol or TCP option being used to detect them, and more importantly, their willingness to disclose their presence.

In this section, we present a novel technique for the detection of middleboxes that does not require their explicit cooperation. We also analyze the problems and challenges that we encountered in the process of its implementation.

A. Conventions and Assumptions

The proposed technique is based on the following assumptions and conventions:

- The protocol for the detection of middleboxes is carried out by two parties: an entity called an *echo client*, denoted by E_c, and another entity called *echo server*, denoted by E_s.
- E_c has the ability to generate and transmit arbitrary network packets.
- E_s has the ability to capture any network packets that arrive to its network interfaces.
- E_c and E_s share some secret K, which has been agreed via some out-of-band mechanism.
- There is a working communication path between E_c and E_s, and both can successfully establish a TCP connection, initiated by the former, over some port p.
- If they exist, middleboxes are located in the communication path between E_c and E_s, and perform some kind of alteration to the traffic that traverses them.

B. Detection Algorithm Overview

The following steps describe the general idea of the proposed algorithm.

1) E_c establishes a TCP connection with E_s over the p port.
2) E_c and E_s agree to establish an application layer session.
3) E_c informs E_s of the type of packets that is about to send.
4) E_s gets ready to capture packets of the requested type, and tells the client that it may proceed with the transmission.
5) E_c starts sending packets.
6) E_s captures the packets as they reach its network interfaces and provides a copy to E_c through the TCP channel established in step 1.
7) E_c receives the copy of the packets returned by the server and compares them with the packets that were sent originally. Any non-trivial difference found in the packets, will reveal the existence of a middlebox at some point in the communication path between E_c and E_s.
8) When the client considers that enough packets have been sent, the connection is closed.

C. The Nping Echo Protocol

In order to implement the basic idea that was outlined in the previous section, it is necessary to design a proper protocol. This section provides a detailed description of such protocol, that we named Nping Echo Protocol (NEP).

The protocol if formed by seven different types of messages, described in detail below. All messages have a common header $H_0 = \{v, t, l, s, T\}$, where v denotes the protocol version number (currently $v = 1$), t indicates the type of message that follows the header, l is the length of the message, s is a sequence number and T is the current time at the sender. The following list briefly describes all message types.

- Type NEP_HANDSHAKE_SERVER, which we will denote by H_s. It is the first message in the three-way handshake that client and server carry out in order to establish a NEP session. It is sent by the server and its purpose is to inform the client of the version of the protocol supported by the server, and to provide a timestamp and a random nonce for security reasons.
- Type NEP_HANDSHAKE_CLIENT, H_c. Its purpose is to indicate agreement on the protocol version, to confirm the random nonce in H_s and to provide another nonce value to be confirmed by the server in its next message.
- Type NEP_HANDSHAKE_FINAL, H_f. Its purpose is to confirm the random nonce in H_c and indicate the successful establishment of the session.

- Type NEP_PACKET_SPEC, H_p. Its purpose is to inform the server of the characteristics of the packets that the client intends to transmit.
- Type NEP_READY, H_r. Its purpose is to indicate that the server is ready to receive network packets from the client.
- Type NEP_ECHO, H_e. Its purpose is to provide the client with a copy of the state of one of his packets when it reached the server.
- Type NEP_ERROR, H_x. Its purpose is to indicate that, due to some error, the session needs to be aborted.
- Type NEP_BYE, H_b. Its purpose is to signal the successful termination of the current session.

The middlebox detection process is conceptually divided into four different phases.

1) Phase 1, Side Channel Establishment Handshake: where client and server agree to establish an echo session. It involves the following steps:

1) E_c establishes a TCP connection with E_s over port p.
2) E_s sends $H_s = \{H_0, n_s, M_s\}$ to E_c, where n_s is a 256-bit random number, and M_s is a message authentication code for H_s.
3) E_c verifies that M_s is correct and sends $H_c = \{H_0, n_s, n_c, M_c\}$ to E_s, where n_s is the same random number included in H_s, n_c is another 256-bit random number generated by E_c, and M_c is a message authentication code for H_c.
4) E_s verifies that M_c is correct and that the received n_s matches the n_s in H_s. If the verification succeeds, E_s sends $H_f = \{H_0, n_c, M_f\}$ to E_c, where n_s is the random number included in H_c and M_c is a message authentication code for H_f.
5) If E_c determines that M_f is correct and that the received n_c matches the n_c in H_c, the session is considered successfully established.

2) Phase 2, Parameter Exchange: where the client informs the server of the packets that it intends to send. It involves the following steps:

1) E_c sends $H_p = \{H_0, s, c, M_p\}$ to E_s, where s is the number of network packets that E_s is planning to send to E_s, and c is a list of characteristics (such as upper level protocol, port numbers, IP identifiers, etc) that describe such packets, and M_p is a message authentication code for H_p.
2) E_s verifies that M_p is correct, gets ready to capture packets from the wire that match the characteristics in c, and sends $H_r = \{H_0, M_r\}$ to E_s, to indicate its readiness.

3) Phase 3, Packet Transmission: where the client transmits the packets and the server returns a copy of what it received. It involves the following steps:

1) E_c verifies the M_r in H_r, generates a set of packets P with the characteristics that it previously announced, and sends each packet p in P, one by one, to E_s, not over the side channel, but through standard packet transmission mechanisms.
2) For each packet p' that E_s captures from the wire, it determines if $p' \epsilon P$, based on the characteristics of p' and the characteristics listed in c.
3) If $p' \epsilon P$, E_s sends a message $H_e = \{H_0, l, p', M_e\}$ to E_s, where l is a number that identifies the link-layer type in p', and M_e is a message authentication code for H_e.
4) When E_c receives H_e, validates M_e, and stores p' for later processing.
5) When client or server find it appropriate, the session is closed sending a message $H_b = \{H_0, M_b\}$ to the other end.

4) Phase 4, Middlebox Detection: where the client processes a received H_e message to detect the presence of middleboxes in the path between him and the server. It involves the following steps:

1) Extract p' from the received H_e message.
2) Compare the value of every field in p' with the original packet p.
3) Any non-trivial difference between p' and p indicates the presence of a middlebox in the path.

Because the client has access to both versions of the packet (the original packet before transmission, p, and the version of the packet that was received by the server, p'), it can compare them and spot any alterations made in transit. Of course, not all differences indicate the presence of middleboxes, as IP packets are expected to present trivial alterations in transmissions that involve multiple hops: for every hop, the TTL is decremented by one unit and the checksum is recomputed. However, the variation of the TTL itself already offers the client some information: the number of routers the packet traversed until it reached the server.

Any additional differences found between p' and p will evidence the presence of an intermediate device in the path. In order to determine what type of device has altered p, the client needs to have a database of middlebox types and characteristics. In particular we suggest a database of tuples $m_i = \{T, L, F\}$, where T is the type of device (NAT, ALG, proxy, etc.), L is the list of layers at which type T operates, and F is the list of "fields of interest" of the device. Based on the fields that changed their value in transit, and the layer those fields belong to, the client should be able to determine the type T of the device that modified the packet in transit.

D. Security Problems and Implementation Challenges

Conceptually the operation of the protocol is simple, but in practice, its implementation involves several problems and challenges that must be taken into account. This section discusses some of those problems.

1) Packet identification at the server side: one of the main challenges that the echo server must face is the to identify, among the set of packets that are captured from the wire, which of them were generated by the client. Any host connected to the Internet is exposed to a continuous noise of unsolicited packets that arrive to their network interfaces. Such traffic may be caused by port scans [1], [2], or by misconfigured routers an end systems [3], [18]. It seems clear that no echo server connected to the Internet can expect to receive traffic only from an echo client, and therefore, it must be capable to distinguish legitimate packets from any noise.

To solve this problem, we included the H_p message in the protocol. Such message is sent by the client and its purpose is to provide the server with a list of characteristics of the network packets that the client is about to send. Based on such characteristics, the server can discard packets that do not match them. However, the server must tolerate a certain degree of variability, as the presence of middleboxes in the communication path can cause alterations of the packets in transit, and therefore, not all characteristics may survive the transmission. In particular, we propose to classify captured packets into two groups, noise and legitimate, through an scoring algorithm. Such algorithm may be described as follows:

Let p be a network packet, $F = \{f_1, f_2, ..., f_n\}$ the set of the n fields that form p, $L = \{l_1, l_2, ..., l_n\}$ the set of lengths of the fields in F expressed in octets, and $V = \{v_1, v_2, ..., v_n\}$, the set of specific values that the fields in F take for a given p.

1) E_c and E_s perform the three-way handshake that establishes a NEP session.
2) E_c sends $H_p = \{H_0,\ s,\ c,\ M_p\}$ to H_s where $c = \{F, L, V\}$
3) E_s indicates that is ready to receive packet p, through an H_r message sent to E_c.
4) E_c builds a packet p formed by n fields with the values in V and sends p to E_s.
5) In transit, p traverses one or more intermediate devices that alter the value of one or more fields.
6) E_s captures a packet p', formed by fields with values V'.
7) E_s computes a score for p' as follows: $s(p') = \sum_{i=1}^{n} 1 \ \forall\ v_i = v'_i$
8) If $s(p')$ exceeds some threshold t_p, E_s determines that packet p' has been generated by E_c, and sends a copy of p' to E_c, encapsulated in an H_e message.

The operation performed by E_s in step 7 is the score of a given packet based on its similarity with the characteristics provided by the client in the H_p message. In particular, it reflects the number of fields that are equal, which certainly offers information about their similarity. However, not all matches should contribute equally to the score, as the probability of a random value match varies inversely with the length of the field. Statistically, a field of length n octets will match in one out of 2^{8n} packets. For this reason, the scoring operation needs to be modified, so it takes lengths into account. A possible approach could be:

$$s(p) = \sum_{i=1}^{n} 2^{l_i \cdot 8} \ \forall\ v_i = v'_i$$

In other words, the sum of the inverse of the probabilities of a random match. One major drawback of this approach is that the score value would vary a lot, which makes it difficult to choose the threshold value t_p that a packet must score in order to be considered legitimate. Another possible solution would be to compute $s(p)$ as follows:

$$s(p) = \sum_{i=1}^{n} l_i \ \forall\ v_i = v'_i$$

In this case, the contribution of a field to the score varies linearly with its length, what reduces the supremacy of long fields. We establish one exception to this rule: matches of application layer data, for which we propose an upper bound of 4. In other words, when the list of characteristics provided by E_c in H_p contains information about a payload above the transport layer, the maximum contribution of any positive match will be limited to the contribution of an equivalent 4-octet field. This prevents very common payloads like *"GET / HTTP/1.1\r\nHost:"* from causing the score to exceed the t_p threshold even when no other fields matched.

Nevertheless, it does not make sense to consider all fields with the same length equal, as it is very common for network protocols to have fields with fixed or easily predictable values. Examples include header lengths, flags or protocol identifiers. Consequently, $s(p)$ needs to be modified so it takes into account that fields that take random values or values that are difficult to predict by a third party without access to the traffic sent by E_c, are more significant that others. We propose the addition of a new element to the formula, a weighting factor that adjusts the importance of each particular field. We therefore define a new set of weighting factors $W = \{w_1, w_2, ..., w_n\}$, where w_i is the weight for field f_i in F. With this modification, $s(p)$ would be computed as follows:

$$s(p) = \sum_{i=1}^{n} l_i \cdot w_i \ \forall\ v_i = v'_i$$

This approach offers a great flexibility for the implementation, something that is essential, due to the wide variety of protocols and header fields. The value taken by each w_i in W depends on the syntax and semantics of each protocol field. In table I we summarize the weights that we used in our implementation. However, we do not claim that our selection is optimal, leaving that as a future line of work.

There is one last issue with this scheme. The introduc-

Protocol	Field	Weight
IPv4	TOS	1.0
IPv4	Protocol	0.9
IPv4	Identifier	2.5
IPv4	Fragment Offset	1.0
IPv6	Traffic Class	1.0
IPv6	Flow Label	2.5
IPv6	Next Header	0.9
TCP	Source Port	1.5
TCP	Destination Port	1.0
TCP	Sequence	2.0
TCP	Acknowledgement	1.0
TCP	Flags	1.0
TCP	Window	1.0
TCP	Urgent Pointer	1.0
ICMP	Type	1.0
ICMP	Code	1.0
UDP	Source Port	1.5
UDP	Destination Port	1.0
Other	Payload	1.0

Table I
SUMMARY OF WEIGHTING FACTORS

tion of weighting factors assumes that the contribution of a given matched field to the score is always the same, independently of the value that produced the match. However, some network protocols choose special field values to indicate that some functionality is not being used. One good example of it is the Acknowledgement field in TCP, that takes a value of zero whenever the ACK flag is not set. In that case, matches of the value zero should not influence the score as much as other values that are less common and more difficult to predict. For this reason, we add one last element to $s(p)$, as follows:

$$s(p) = \sum_{i=1}^{n} l_i \cdot w_i \cdot z(f_i, v_i') \ \forall \ v_i = v_i' \text{ , where}$$

$$z(f_i, v_i') = \begin{cases} w_z & if\ v_i' is\ a\ default\ value\ of\ f_i \\ 1 & otherwise \end{cases},$$

and w_z is the special case weighting factor for which $0 \leqslant w_z < 1$.

2) Multi-session support: another problem that an echo server must face is the provision of the service to multiple, simultaneous clients. This introduces some challenges when the server has to determine which packets belong to which particular client. The most obvious solution would be to select those packets whose source IP address matches the address observed from the client's side channel establishment. However, such condition is not enough to guarantee the accuracy of the identification, as the side channel itself involves certain TCP traffic exchanged between client and server that must be ignored. In addition, one client may decide to establish multiple session in parallel, what would result in many packets with the same source and destination IP address but that belong to different sessions. Same applies to different clients that are behind a single NAT device.

It could also be the case that some client decides to use the echo service to determine if a packet with an spoofed IP address can reach the server. In this case, the source IP address observed by the server would not match the client's. In the same way, packets should not require to be addressed to the server's IP address, as the server could be run inside some intermediate device, like a router, placed along the path.

It seems clear that while the IP address used by the client to establish the side channel can be a useful piece of information for the server in some cases, it must not be relied upon, as there are some scenarios in which such information can not be used reliably.

Our implementation does not take IP addresses into account because a minor modification to the scoring algorithm presented in the previous section lets servers handle simultaneous echo sessions and match captured packets with the appropriate client in an effective manner. Let $U = \{u_1, u_2, ..., u_k\}$ be a list of k clients that have an active echo session with the server (sorted oldest first), and $C = \{c_1, c_2, ..., c_k\}$ the set of characteristics, $c_j = \{F_j, L_j, V_j\}$, provided by each client in the H_p message, the process is as follows:

1) E_s captures a packet p', formed by fields with values V'.
2) For each c_j in C, E_s computes a score for p' as follows: $s(p', c_j) = \sum_{i=1}^{n} l_i \cdot w_i \cdot z(f_i, v_i') \ \forall \ v_i = v_i'$ where $v_i \epsilon V_j$
3) E_s selects the client with the highest score for packet p, $s_m = s(p', c_m)$.
4) If s_m exceeds some threshold t_p, E_s determines that packet p' has been generated by the user u_m, and sends a copy of p', encapsulated in an H_e message, through the side channel established with s_m.

Although the algorithm does not guarantee a total accuracy, we believe that, providing clients select some of their packet characteristics randomly, the probability of misidentifying packets is reasonably low. Nevertheless, a malicious client with the ability to guess all packet characteristics provided by another client, could include the same c in its H_p message and therefore, obtain the same score for each packet. To alleviate this problem we propose that the server resolves ties by awarding the packet to the client that connected first.

3) Significant protocol layers: another aspect to consider in the design of the protocol is which network layers are significant to the process. It seems reasonable to take the network and transport layers into account, as they play a key role in today's networks, their protocol headers are delivered end-to-end, and there is a wide variety of middleboxes that operate at that level.

Link layer headers, on the other hand, are only propagated in a point-to-point fashion, so unless client and server are connected to the same subnet, it does not make much sense for the client to provide details about the link layer parameters that it intends to use. Nevertheless, it could be interesting if echo servers included link layer headers in H_e packets. A possible usage scenario would be a server that is located in a subnet with more than one router. If the client has access to the link layer source address, it could be able to determine if packets reach the server forwarded by different devices. Same applies to devices performing load balancing at the link layer [4]. It should be noted that in these cases, the server would have to tell the client explicitly which link-layer protocol is in operation, so the client can interpret the data correctly, and determine the offset where the network layer header starts. That is the reason why, in our implementation, message H_e includes a link layer identifier.

The application layer is also a good candidate to be considered for the protocol. If the server also included a packet's application data in H_e messages, clients would be able to detect application layer gateways or any other type of middlebox that alters data at that level. Although the implementation is straightforward for connectionless protocols like UDP or ICMP, it presents some challenges when it comes to connection-oriented protocols like TCP. In order for a given client to transmit data over TCP, it must first establish a connection, through the standard TCP three-way handshake. Therefore, the client needs to be able to generate custom TCP packets for that matter. Let E_s be a host that offers an echo service and also some other network service through port n, a client E_c would follow these steps:

1) Generate a TCP packet with the SYN flag set and a target port number n and send it to E_s.
2) Start capturing packets that arrive to its network interfaces.
3) Capture the TCP packet with the SYN and ACK flags set that E_s sends in response.
4) Generate and send a TCP packet with the ACK flag set, and the appropriate sequence and acknowledgment numbers.
5) Transmit any application layer data in additional TCP packets, with the appropriate parameters for the connection.

This has a significant impact on the complexity of a client's implementation, as it requires echo clients to emulate TCP stacks, at least partially, keeping track of sequence and acknowledgment numbers and handling packet losses and retransmissions. However, a client can not simply invoke system calls such as *connect()*, as it needs to know the value of the different header fields at the network and transport layers, in order to produce meaningful H_p messages. It is true that we could relax the restriction that we imposed to application layer matches in section III-D1 so payloads above the transport layer contribute to the score proportionally to their length. This would allow clients to establish TCP connections using standard system calls and produce H_p messages that only contain information about the application layer. Such H_p messages would contain enough information for the scoring algorithm, providing the transmitted payloads contain enough entropy to avoid collisions with other payloads. Nevertheless, our current implementation does not relax the described restriction, nor does yet establish full TCP connections. For this reason, changes in application layer data may only be observed when non-connection oriented transport protocols, such as UDP, are used.

4) Security: the fact that the echo server captures and retransmits packets that reach its network interfaces, makes it an attractive target for attackers that want to access the server's traffic. For this reason, it is important to take security into account in all phases of the protocol. In this section, we will discuss the potential security problems, and the measures we have taken to mitigate them. For this matter, we define two different attacker models, to reflect what we believe are two common uses of the protocol. They differ only in whether the attacker knows the secret K. We assume that the attacker always has control of the network, but may not break encryption or forge message authentication codes.

- **Model 1: trusted clients/private server.**
  - The server and all legitimate clients know a secret K and are honest. No other party knows K.
- **Model 2: untrusted clients/public server.**
  - Secret K is made public, so anyone may use the server. Clients are not assumed to be honest.

We know define the expected security properties of the protocol. In general, the protocol seeks to ensure confidentiality, integrity, and authentication. Nevertheless, Model 1 has more stringent security properties than Model 2. For Model 1, we expect the following security properties to hold:

- **Property 1A:** an attacker cannot make use of the echo service.
- **Property 1B:** an attacker cannot convince a client that it is a legitimate server.
- **Property 1C:** an attacker cannot modify traffic without detection.
- **Property 1D:** once a client and a server have established a session, an attacker cannot access the information exchanged during that session.
- **Property 1E:** when a connection between a legitimate client and a server is ended, it is ended from the point of view of both endpoints (mutual termination). In particular, an attacker cannot keep one end of a session alive.

In Model 2, the attacker knows K, so ensuring confidentiality, integrity, and authentication is impossible. But a malicious client should not be able to deny service to other clients or to gain access to more information than any honest client. For Model 2, these are the expected security properties:

- **Property 2A:** a malicious client cannot interfere with other client sessions.
- **Property 2B:** a malicious client can only see captured packets that correspond to its own, not those of other clients, and especially not any traffic unrelated to the echo protocol.

The first three properties for Model 1 are satisfied by the message authentication code that is appended to every message of the protocol. As the attacker does not know K, it is impossible for him to produce valid messages. In particular, the attacker cannot produce valid H_c messages, so it is becomes impossible to establish sessions with a server (property 1A); he cannot produce valid H_s or H_f messages so he is not able to act as a legitimate server (property 1B); and he cannot alter legitimate messages without being detected because he is not able to recompute message authentication codes (property 1C).

Replay attacks are ineffective. The presence of nonces in the three-way handshake guarantees the freshness of H_c and H_f. In addition, every message contains a timestamp and a sequence number, which allows the receiving party to verify that they belong to the current session. Furthermore, all cryptographic keys are influenced by the nonces (see Section III-D5), so it is highly unlikely that two different sessions use the same keys, which makes it virtually impossible for an attacker to replay any message.

The fourth property is satisfied by the use of encryption for all messages, except for the first three (which correspond to the three-way handshake session establishment). In particular, our implementation encrypts messages after the appropriate message authentication code has been computed. Such authentication code is excluded from the encryption, and is transmitted in clear text.

Property 1E is satisfied by the use of the special messages E_x and E_b. The former is produced and sent by one of the parties to indicate that there was some error that caused the session to terminate. The latter is sent to indicate that the sender wishes to end the session.

In model 2, security properties are harder to satisfy. Property 2A is impossible to meet if the attacker has the ability to intercept a client's traffic. This is a problem for virtually all application layer protocols, as an attacker may easily tear down existing transport layer sessions. Even if protocols like IPSec are in operation, malicious users can choose to block traffic in any direction, what results in a denial of service. Conscious of this limitation, we relax our initial assumption to state that the attacker may have access to the traffic produced by the client or the server, but does not have the ability to intercept it or supplant their identity at the network and transport layers. In other words, the attacker may be able to sniff the traffic exchanged by legitimate clients and servers but may not inject traffic in the network with IP addresses for which he is not the legitimate holder.

In this new scenario, Property 2A is satisfied by the server, as it keeps separate state information for each client, such as nonces, timestamps and sequence numbers. Even though the attacker has access to such information and could produce valid message authentication codes, he cannot inject messages into existing sessions, as he is not able to transmit data on behalf of other clients.

Property 2B has important implications. Our main concern is to prevent malicious clients from accessing other traffic than their own. Once the server has determined that a particular captured packet belongs to a given client, such packet is echoed and never processed again. For this reason, if an attacker manages to convince the server that certain packets are his, such packets will be echoed to the attacker, and not to the legitimate client, what would cause a denial of service for the latter.

Because the attacker has access to any client's H_p message, he can easily establish a session with the server and supply the same list of packet characteristics. This would cause the attacker and the legitimate client to obtain the exact same score for every packet. As we mentioned above, the server resolves ties by awarding the packet to the client that connected first. One could think that this solution prevents attackers from stealing a client's packets. However, the time at which the client and the attacker established a connection with the server is not a valid piece of information to base decisions on. Doing so would allow the following attack.

1) An attacker, E_a, establishes a TCP connection with E_s.
2) E_a and E_s establish an echo session exchanging messages H_s, H_c, and H_f.
3) E_a waits until the victim, E_c, establishes a TCP connection and an echo session with E_s and sends an H_p message.
4) E_a captures the H_p sent by E_c.
5) E_a extracts the list of characteristics c from H_p, generates its own H_p message with the same c, and sends it to E_s.

At this point, the server holds information about two connected clients, E_a and E_c, both with the same c. When E_c starts transmitting network packets, the server will capture them and apply the score operation to each one. Because E_a and E_c have the same c, they will obtain the same score for every packet, but since E_a connected

first, packets will be echoed to the attacker and not to the legitimate client.

To mitigate the attack we suggest making the server record the time at which H_p messages are received and resolve ties based on such time, awarding packets to the client whose H_p arrived first. This does not solve the problem completely, as the attacker might know a fastest network path to reach the server, and could be able to capture the victim's H_p message and make his own H_p arrive to E_s first. However, we believe the solution offers a reasonable compromise between ease of implementation and security.

We are also concerned about a particularly dangerous situation: an attacker that manages to receive H_e packets that contain traffic that is unrelated to the echo protocol. In other words, an attacker that is able to convince the server that any packet that reaches its network interfaces has been generated by him. This has obvious security implications as it would allow the attacker to sniff the server's traffic remotely, from any network location. The attack could be easily carried out if no restrictions are placed in H_p messages. For example, an attacker could send a list of characteristics like $\{IP_{protocol} = 6, IP_{protocol} = 6, ..., IP_{protocol} = 6\}$, which means, "layer above IP equals TCP". This would increase the attacker's score multiple times for any TCP packet that reaches the server. If enough duplicate tests are provided, the score will exceed the threshold value t_p, causing the server to send a copy of every TCP packet to the attacker. Another possibility would be to provide $\{ICMP_{type} = 0, ICMP_{type} = 1, ..., ICMP_{type} = 255\}$, what would increase the attacker's score for any ICMP message, regardless of its "Type" field.

To solve this problem, we suggest prohibiting multiple tests with the same left-hand side. Additionally, servers should verify that the characteristics provided by clients are reasonable. Examples include, verifying that IPv4 or IPv6 characteristics are specified, but not both at the same time, or verifying that there are characteristics for only one transport layer protocol, not many.

5) Cryptographic keys: our implementation of the protocol uses a set of five cryptographic keys per client session. All keys are derived from the K secret that E_c and E_s share, the random nonces exchanged during the three-way handshake (n_c and n_s), and a unique type identifier for each key. There is one encryption key and one message authentication key for each direction ($E_c \rightarrow E_s$ and $E_s \rightarrow E_c$). Additionally, there is an special key used for the authentication of message H_s, that is generated and used temporarily due to the absence of the client-side generated nonce at the time message H_s is created.

The key derivation is performed through a slight variation of the PBKDF1 algorithm [8], which uses the SHA-256 hash function. A pseudo-code representation is presented in Alg. 1. Note that $N = \{n_s, n_c\}$, except for the authentication of H_s, where $N = \{n_s, 0\}$.

Algorithm 1 Key derivation Process

```
h=SHA256(K + N + Key_Type_Id)
do(1000 times){
        h=SHA256(h);
}
```

The implementation uses AES-128 for encryption and HMAC-SHA256 for message authentication. In those cases where the generated keys are longer than required, the last $256 - x$ bits of key material are discarded (least significant bits), where x is the desired key length.

E. Usage Scenarios

This section describes some examples of usage scenarios for the middlebox detection protocol described above. Note that the proposed scenarios typically require the server to be located out of the client's network (Fig. 1). Although this is not true for all cases, for simplicity we have omitted that kind of details from the descriptions.

1) Scenario 1, detect address translation: clients may detect the presence of a NAT device in their local network if they observe that their packets reach the server with a different source IP address. In such case. the observed address would be the NAT's public IP (or the last NAT's public address if there are multiple nested NAT devices).

2) Scenario 2, list blocked port numbers: clients may determine which ports are being blocked by a firewall by sending packets to all possible 2^{16} port numbers on the server. Packets for which an H_e response was received indicate that the firewall does not block the corresponding port.

3) Scenario 3, detect blocked protocols or message types: clients may determine if a particular protocol or message type is being blocked by a firewall, by sending packets with the desired characteristics and checking if they were blocked in transit, based on the presence or absence of H_e messages. A typical example may be to

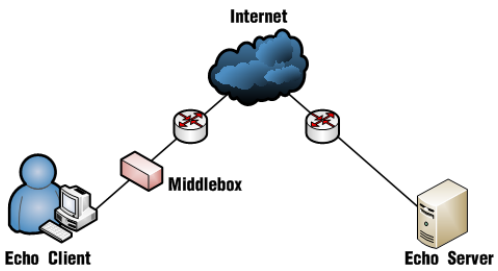

Figure 1. Typical setup.

detect the ability to send ICMP Echo requests to the Internet.

4) Scenario 4, diagnose connectivity problems: when the establishment of a TCP connection times out, clients may determine if the problem occurred because the SYN packet never reached the server, because the SYN-ACK did not reach the client, etc. This can be easily extrapolated to other protocols.

5) Scenario 5, detect path MTU through IP fragmentation: Path MTU Discovery (PMTUD) does not work if some intermediate firewall blocks ICMP Destination Unreachable messages. In that case, a client may determine a path's minimum MTU by sending IP datagrams of various sizes, without the DF bit set, and checking received H_e messages for signs of fragmentation along the path.

6) Scenario 6, detect anti-spoofing policies: clients may test whether their network gateway filters out spoofed packets (packets leaving the network whose source address does not belong to the network address space), by sending IP datagrams with spoofed IP addresses, and checking if such datagrams reach the server.

7) Scenario 7, detect in-line IDSs: clients may detect the presence of an in-line Intrusion Detection System by sending packets that are known to trigger IDS alarms to the server. The fact that one or more packets do not reach the server could indicate the presence of an in-line IDS that prevents attacks by blocking suspicious traffic.

IV. Experimental Results

In this section, we evaluate our implementation of the protocol through four different experiments.

A. Experiment 1: Router

In this experiment we set up a client, E_c, and a server, E_s, located in two different subnets that are interconnected by a router, E_r. All three participants are regular desktop machines running a GNU/Linux operating system. E_r has two network interfaces, I_c and I_s which are connected to the client's subnet and the server's subnet respectively. We configure E_c to send five ICMP Echo requests to E_s. After we run the experiment, from the client's perspective, we observe the following:

- ICMP Echo packets reached the server's machine successfully: E_c received five H_e messages which contained the ICMP Echo requests that arrived to the server's network interface.
- Server's machine responded to the requests with ICMP Echo replies: E_c captured ICMP Echo replies that contained the appropriate ICMP message identifiers, sequence numbers, and IP source addresses.
- There is a network distance of one hop between E_c and E_s: the packets included in the received H_e messages had a TTL of one unit less than the originals.

B. Experiment 2: Firewall

This experiment is a modification of the previous one, where the router now also assumes the role of a firewall. At the server side, we set up a trivial network service that accepts connections on port k. We configure E_c to send 10 TCP packets with the SYN flag set and a destination port number that equals k. We set up firewall rules in E_r to allow forwarding of any IP datagrams except those that contain a TCP header whose source port matches k. After we run the experiment, from the client's perspective, we observe the following:

- TCP packets issued by E_c reached the server's machine successfully: E_c received 10 H_e messages which contained such packets.
- There is a network firewall between E_c and E_s that drops some packets: the client did not receive any response from the server, even though the trivial network service was supposed to send TCP packets with the SYN and ACK flags set in response, or at least with the RST flag set to refuse the connection.

C. Experiment 3: NAT device

In this experiment we modify E_r to provide address translation between the two subnets. We configure E_c to send five UDP packets with a random payload to a closed port on E_s. After we run the experiment, from the client's perspective, we observe the following:

- All UDP packets reached the server's machine successfully: E_c received five H_e messages which contained the packets.
- Server's machine responded to the requests with ICMP Port Unreachable messages: E_c captured five ICMP error messages that contained the original UDP datagrams that caused the error.
- There is NAT device between E_c and E_s, that operates at the network and transport layers: the packets included in the received H_e messages had a different source IP address and a different source port number than the originals.
- The NAT device handles ICMP error messages correctly: source IP addresses, source port numbers, and checksums found in the datagrams encapsulated inside ICMP messages were altered accordingly by the NAT device. Even when we performed a second experiment where we instructed E_c to set the UDP checksum to zero, the NAT device behaved correctly and did not attempt to recompute checksums.

D. Experiment 4: HTTP caching proxy

In this experiment we replace the E_r device running GNU/Linux with a machine E_p that runs an ISA Server 2006 on a Microsoft Windows 2003 Server system. E_p is configured to act as a transparent HTTP caching proxy. We configure E_c to send 10 TCP packets with the SYN flag set. Half of the packets are destined to port 80

(HTTP) and the other half to port 22 (SSH). Additionally, at the server side, we set up a network service that accepts connections on port 80 and port 22. After we run the experiment, from the client's perspective, we observe the following:

- All TCP packets destined to port 22 reached the server: E_c received five H_e messages which contained such packets.
- None of the packets destined to port 80 reached the server: E_c did not receive H_e messages for those packets.
- Server's machine responded to all port 22 requests: E_c captured five TCP packets with the SYN-ACK flags set and source port 22.
- Some intermediate device forged a response to one of the packets destined to port 80: E_c captured a valid response to the first TCP packet (SYN-ACK flags and proper acknowledgment number) but such response presented significant differences with the packets received from port 22, what suggests that such responses were produced by two different end systems. In particular, responses from port 22 had a TTL value of 63, an IP Identification value of zero, and a TCP window size of 14600 bytes, while the response from port 80 had a TTL value of 128, non-zero IP Identification values and a TCP window size of 16384.

V. Conclusions and future work

In this paper we have presented a novel technique for the detection of intermediate devices in the path between two end nodes. We have suggested a client/server approach where the client, assisted by the server, gets access to two versions of the same network packet: the one generated by the sender, and the packet that was actually received at the other end. The analysis of the differences between those two packets allows the detection of middleboxes that produced alterations to the packets that traversed them, without requiring their explicit cooperation.

We have not only presented the general outline of the technique, but a complete design of a protocol that achieves our goals. We have analyzed its main problems and security concerns, and provided solutions to mitigate them. We also demonstrated the flexibility of the protocol and suggested many different applications and usage scenarios. It must be noted that the theoretical concepts that were introduced in this document are backed up by an actual implementation, the *Nping* tool, which is freely available under an open source license [13]. Anyone may download the application and test the client side against a publicly accessible instance of the echo server located at *echo.nmap.org*.

However, neither the protocol nor the implementation are fully complete. Our proposal should be considered an initial approach to the problem since there are several issues that have been left out of the scope of this paper. First of all, effort must be put in the creation of a database of middlebox models to assist clients in the identification of particular intermediate device types. Secondly, in our protocol, the role of the sender is always assumed by the client side. This limits the ability to detect devices that alter flows in the opposite direction (server to client). Allowing clients and servers to exchange their roles dynamically would improve the overall detection capabilities of the system. Finally, there is certainly room for improvement in the way our implementation handles application layer sessions. Its inability to establish full TCP connections limits the types of middleboxes that can be detected. However, this an area we are working on so we expect to offer such functionality in the near future.

References

[1] Yegneswaran, V., Barford, P. and Ullrich, J. (2003). *Internet Intrusions: Global Characteristics and Prevalence.* SIGMETRICS'03, ACM. California, United States. [Available on-line] <http://pages.cs.wisc.edu/~pb/dshield_paper.pdf>

[2] Honeynet Project. (2001). *Know Your Enemy: Statistics. Analyzing the past, predicting the future.* The Honeynet Project. United States. [Available on-line] <http://old.honeynet.org/papers/stats/>

[3] Pang, R., Yegneswaran, V., Barford, P., Paxson, V. and Peterson, L. (2004). *Characteristics of Internet Background Radiation*. Proceedings of the 4th ACM SIGCOMM conference on Internet measurement. P. 27-40, October 2004. [Available on-line] <http://www.icir.org/vern/papers/radiation-imc04.pdf>

[4] IEEE. (2008). Link Aggregation. IEEE Standard for Local and metropolitan area networks. IEEE Computer Society. [Available on-line] http://standards.ieee.org/getieee802/download/802.1AX-2008.pdf

[5] Aoun, C. and Davies, E. (2007). Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status. RFC 4966. Network Working Group. Internet Engineering Task Force. [Available on-line] http://www.ietf.org/rfc/rfc4966.txt

[6] Carpenter, B. and Brim, S. (2002). Middleboxes: Taxonomy and Issues. RFC 3234. Network Working Group. Internet Engineering Task Force. [Available on-line] http://www.ietf.org/rfc/rfc3234.txt

[7] Joseph, D. and Stoica, I. (2008). Modeling middleboxes. IEEE Network. September-October 2008. Volume 22, Issue 5. p20-25. [Available on-line] http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4626228&tag=1

[8] Kaliski, B. (2000). PKCS #5: Password-Based Cryptography Specification. Version 2.0. RFC 2898. Network Working Group. Internet Engineering Task Force. [Available on-line] http://www.ietf.org/rfc/rfc2898.txt

[9] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D. and Jones, L. (1996). SOCKS Protocol Version 5. RFC 1928. Network Working Group. Internet Engineering Task Force. [Available on-line] http://www.ietf.org/rfc/rfc1928.txt

[10] Karsten, M., Keshav, S., Prasad, S. and Beg, MO. (2007). An Axiomatic Basis for Communication. SIGCOMM 2007. Association for Computing Machinery. [Available on-line] http://conferences.sigcomm.org/hotnets/2006/karsten06axiomatic.pdf

[11] Nalepa, G. (2007). A Unified Firewall Model for Web Security. Advances in Intelligent and Soft Computing, 2007. Volume 43. p248-253. Springer Berlin / Heidelberg. [Available on-line] http://www.springerlink.com/content/k7114364j0j27447/

[12] Maldone. S., Bohra, A. and Iftode, L. (2007). Filewall: A Firewall for Network File Systems. Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic,

and Secure Computing (DASC'07), Baltimore, MD, September 2007. [Available on-line] http://www.cs.rutgers.edu/discolab/filewall/pubs/filewall_dasc07.pdf

[13] MartinGarcia, L. (2011). Nping. Nmap Security Scanner. Insecure.com LLC. [Available on-line] http://nmap.org/nping/

[14] Stiemerling, M., Quittek, J. and Taylor, T. (2008). Middlebox Communication (MIDCOM) Protocol Semantics. RFC 5189. Network Working Group. Internet Engineering Task Force. [Available on-line] http://www.ietf.org/rfc/rfc5189.txt

[15] Tsirtsis, G. and Srisuresh, P. Network Address Translation - Protocol Translation (NAT-PT). RFC 2766. Network Working Group. Internet Engineering Task Force. [Available on-line] http://www.ietf.org/rfc/rfc2766.txt

[16] Lear, E. (2001). Requirements for Discovering Middleboxes. Internet draft. Network Working Group. Internet Engineering Task Force. [Available on-line] http://tools.ietf.org/html/draft-lear-middlebox-discovery-requirements-00.txt

[17] Knutsen, A., Frederick, R., Li, Q., and Yeh, WJ. (2010). TCP Option for Transparent Middlebox Discovery. Internet draft. Network Working Group. Internet Engineering Task Force. [Available on-line] http://tools.ietf.org/html/draft-knutsen-tcpm-middlebox-discovery-04.txt

[18] Wustrow, E., Karir, M., Bailey, M., Jahanian, F. and Huston, G. (2010). Internet background radiation revisited. Proceedings of the 10th annual conference on Internet measurement. Melbourne, Australia. [Available on-line] http://doi.acm.org/10.1145/1879141.1879149