MIDDLEWARE APPROACHES TO MIDDLEBOX TRAVERSAL

BY

SECHANG SON

A dissertation submitted in partial fulfillment

Of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

At the

University of Wisconsin—Madison

2006

Abstract

The networking community is working toward a *differentially universal network*, a network that provides universal connectivity to benign endpoints, while isolating malicious ones. *Middleboxes* (a.k.a. firewalls and NATs) could be regarded as important components that contribute to this effort by passing benign traffic, while blocking malicious traffic. However, many benign applications cannot communicate with each other because middleboxes block their traffic in an effort to block malicious traffic (i.e. a connectivity problem). On the other hand, malicious traffic passes middleboxes through openings made for benign applications (i.e. a security problem).

To deal with the connectivity and security problems, this dissertation researches *middlebox traversal*, which we define as a technique which enables authorized applications to communicate through middleboxes while minimizing the increase in the chance that unauthorized applications will be able to do so. We develop a framework of middlebox traversal and a family of new traversal techniques leveraging the framework.

Our framework identifies four types of middlebox policies, which we believe cover most typical policies. These types are “outbound only”, “fixed white-list”, “fixed black-list”, and “dynamic white-list”. The framework also captures major characteristics of a traversal mechanism, including targeted middlebox policies. Logically, the framework defines a multi-dimensional space defined by those characteristics.

Next, we select several representative points in the multi-dimensional space such that at least one point is selected for each type of middlebox policy and develop a family of new middlebox traversal techniques—*GCB*, *XRAY*, and *CODO*—for these points. Each technique can be an enabling tool that helps applications to communicate through middleboxes but also a security tool that network administrators may use for perimeter defense. These traversal techniques form a family because they are developed under a connection arrangement scheme.

GCB (Generic Connection Brokering) targets the "outbound only" type of middlebox policy. GCB decouples the direction of a connection from the role of an endpoint (i.e. whether it is a client or server) to enable inbound communications. Unlike the conventional Berkeley socket system, GCB decides the direction of a connection based upon the relative topology of communicating parties instead of their roles. *XRAY* (middleboX traversal by RelAYing) targets the "fixed white-list" and "fixed black-list" types of middlebox policies. XRAY is a proxy-based mechanism: a proxy of a network relays packets to/from next hop proxies or authorized applications. *CODO* (Cooperative On-Demand Opening) targets the "dynamic white-list" type of policy. CODO dynamically opens and closes middleboxes for authorized applications. Through a secure channel, CODO-enabled applications report their address usage to the CODO agent, add-on software of middleboxes. Based upon the report, the CODO agent dynamically adds and deletes middlebox rules for authorized applications.

The major contributions of this dissertation are (1) a family of new middlebox traversal techniques which not only facilitate application's communication through middleboxes but also provide security tools for perimeter defense and (2) a framework that lays a common ground for the further discussion of middlebox traversal.

Contents

1. Introduction

The networking community is working toward a *differentially universal network*, a network that provides universal connectivity to benign endpoints, while isolating malicious ones. The Internet was originally designed to support communications between any two endpoints and network administrators sincerely worked to provide users better connectivity (aiming for universal connectivity). As network usage expanded and malicious users became a problem, network researchers and administrators began to try to block as much malicious traffic as possible (aiming to isolate malicious endpoints), while continuing their efforts to provide better connectivity for benign endpoints.

A *network perimeter* is the boundary between the private and locally managed-and-owned side of a network and the public and usually provider-managed side [1]. It generally consists of *middleboxes* [3] (a.k.a. firewalls and NATs[1] [2]) and other entities such as NIDS (Network Intrusion Detection Systems) and proxies in the DMZ (Demilitarized Zone). *Perimeter defense* refers to a security measure for monitoring, analyzing, and controlling traffic passing a network perimeter. It is an important means to get closer to a "differentially universal network". It also has many benefits: it provides the first line defense of resources within a network, complementing end-to-end security; it provides an easy way of centralized defense; and some attacks such as network worms (e.g. [4] [6]) and DOS (Denial Of Service) attacks [5] [47] can be prevented more effectively at the perimeter than at endpoints [66]. However, today's Internet is still far from the ideal network. Many benign applications cannot communicate with each other because network perimeters block their traffic in an effort to block malicious traffic (i.e. a connectivity problem). On the other hand, malicious traffic passes network perimeters through openings made for benign applications (i.e. a security problem).

[1] Though IETF uses "middleboxes" to refer to more than just NATs and firewalls [55], it currently focuses on those two devices.

Middleboxes are major components of network perimeters and most traffic qualifications (i.e. decisions to pass or drop traffic) are made at those devices. In this dissertation, we particularly focus on middleboxes and study middlebox traversal, for which we will introduce the traditional definition and then our reformulated definition.

Middlebox traversal has been understood mostly, if not solely, from an application's connectivity perspective. From this perspective, it is traditionally defined as a mechanism that enables applications to communicate through middleboxes. This definition does not distinguish between applications that network administrators want to allow to pass through middleboxes and those that they want to block. For this reason, network administrators generally consider middlebox traversal as breaking the purpose of middleboxes. In this dissertation, we consider middlebox traversal from a perimeter defense perspective as well as an application's connectivity perspective and define *middlebox traversal* as a technique which enables authorized applications to communicate through middleboxes while minimizing the increase in the chance that unauthorized applications will be able to do so. According to this definition, middlebox traversal is not only an enabling technique that helps applications to communicate through network perimeters but also a security technique that network administrators can use as a component of perimeter defense.

This dissertation develops a framework of middlebox traversal and a family of new traversal techniques leveraging the framework. The framework identifies representative types of middlebox policies and captures major characteristics of a traversal mechanism, including targeted middlebox policies, and trade-offs between characteristics. Logically, the framework defines a multi-dimensional space defined by those characteristics. Next, we select several representative points in the multi-dimensional space such that at least one point is selected for each type of middlebox policy and develop a family of new middlebox traversal techniques for these points.

The major contributions of this dissertation are (1) a family of new middlebox traversal techniques which not only facilitate application's communication through middleboxes but also provide security tools for perimeter defense and (2) a framework that lays a common ground for the further discussion of middlebox traversal.

1.1 Framework

1.1.1 Middlebox policy classification

Our framework identifies four types of middlebox policies, which we believe cover most typical policies. These types are “outbound only”, “fixed white-list”, “fixed black-list”, and “dynamic white-list”. (Three more types are presented in the main text. However, these are added only for reference or completeness of our classification.)

The “outbound only” type permits outbound but no inbound connections as its name implies. Most middleboxes keep track of the state of network connection and distinguish very well between reply and unsolicited packets. Therefore, this type of policy can be enforced effectively by using current middlebox technology. It may also be the most appropriate type for organizations using private networks for dealing with an IPv4 address shortage rather than for security reasons. However, this type puts much of the security responsibility on endpoints. It may provide a perimeter defense secure enough[2] for networks within which endpoints are well hardened. However, it is difficult with this type of policy to achieve a perimeter defense secure enough for networks where endpoints are managed by end users who may not be as security conscious as network administrators and may install applications insufficiently hardened.

The “fixed white-list” type permits a small fixed number of authorized points (i.e. hosts, ports, subnets, or combinations of them) to pass through the middlebox. These points can be inside, outside, or on a network perimeter and can be either endpoints or relay points via which others can communicate. Organizations having this type of policy often use authorized points to relay traffic for specific applications. In this case, relay points generally understand application traffic better than middleboxes because they usually terminate transport connections and security associations, and also because they are often customized to understand applications’ communication semantics. Another benefit of this type is that since a relatively small number of openings need be made at

[2] Perimeter defense and end-to-end security complement each other. We may achieve a desired level of security through various combinations of perimeter defense and end-to-end security. Therefore, whether a perimeter defense is secure enough or not depends on how well endpoints handle attacks that the perimeter defense may not block.

middleboxes, network administrators may be able to closely watch traffic which passes through those openings. However, some networks running various applications may have to define too many authorized points. Also, it may not be possible for certain networks to prevent unauthorized points from communicating through the openings made for authorized points.

The “fixed black-list” type blocks traffic from a fixed number of points but permits all the others. Security experts do not recommend this type for general cases because a network may end up having indefinite authorized points. However, this may be an appropriate type when most neighbors are trusted. In a nested network, for example, this type may be used to prevent a few publicly accessible nodes inside the outer perimeter from accessing nodes inside the inner perimeter.

The “dynamic white-list” type allows a dynamic set of points to pass through the middlebox. As middleboxes become smarter with newer capabilities such as application or protocol awareness, sites define only higher level policies and let middleboxes decide dynamically which points can communicate through them. For example, administrators specify a high level policy allowing an application to communicate through, and then middleboxes dynamically create openings (i.e. add points to the white-list) for the application as needed. Middleboxes generally learn application’s need for openings either by watching and analyzing application’s traffic—the in-band method—or by receiving explicit requests for openings from applications via control channels—the out-of-band method. This type benefits from advanced middlebox technologies and places fewer burdens on network administrators. It generally allows a smaller middlebox rule set and therefore reduces human errors. However, there are several problems with this approach. Network administrators may have little knowledge of the low level consequence of high level policies. This may make it difficult to notice and track security problems. Also, as new applications or protocols need be supported, new intelligence may have to be added to middleboxes if the in-band method is used.

1.1.2 Major dimensions of middlebox traversal

Our framework also identifies major dimensions which address aspects of a middlebox traversal mechanism. We group related dimensions into connectivity, security,

deployability, and performance classes. Connectivity and security classes particularly address the traversal mechanism in an application's connectivity and a perimeter defense context, respectively.

Dimensions in the connectivity class address the versatility or limitations of a traversal mechanism in enabling authorized applications to communicate through various middleboxes. Traversal mechanisms have many restrictions and limitations. No single system supports all types of middlebox policies we identified; some mechanisms support only certain types of applications or protocols; some assume specific middlebox behaviors.

Security dimensions address potential impacts of a traversal mechanism on perimeter defense and endpoint security. A traversal mechanism can be a security tool for perimeter defense because it may help a network perimeter to distinguish between traffic for authorized and unauthorized applications. It may change a network perimeter or end host by installing new or modifying existing entities. Also, it may affect end-to-end security by securing a certain or the entire part of an end-to-end channel.

Deployability dimensions address expected barriers for a traversal mechanism to be deployed and accepted by user communities. They address issues such as what changes of network perimeters and endpoints a traversal mechanism requires, what privileges it requires, and whether it can be gradually deployed if global deployment is required.

Performance addresses expected overheads of a traversal mechanism in connection setup (i.e. latency) and data communication (i.e. bandwidth).

1.2 A Family of Middlebox Traversal Techniques

We have developed a family of three traversal techniques—*GCB*, *XRAY*, and *CODO*—for representative points in the space that the framework defines. Each technique can be used as a security tool for perimeter defense as well as an enabling tool that facilitates communications through middleboxes. Each has different characteristics and may support different groups of organizations. Those techniques form a family because they are developed under the same connection arrangement scheme. Because they are interoperable with each other as a family, organizations using different

techniques can communicate with each other [52].

Overlay networks [13]-[16] [64] build customized networks at the application layer on top of the unchanged Internet substrate to improve the Internet, which is hard to change. This dissertation takes a similar approach for middlebox traversal. Without changing the Internet or the operating system, we move one step closer to a "differentially universal network" by building an application layer communication system which allows authorized applications to communicate through middleboxes while blocking unauthorized ones as much as possible. In this system, each network uses one or more traversal techniques that are appropriate for its middlebox type, middlebox policy, and other constraints and criteria. Because those techniques are interoperable with each other, an end-to-end channel is provisioned with each network opening its middleboxes using its chosen technique.

GCB (Generic Connection Brokering) [51] targets the "outbound only" type of middlebox policy. With GCB, clients inside the network perimeter communicate with outside servers as usual. To enable inside servers to communicate with outside clients, GCB uses the idea of a reception desk. Each network has one or more GCB agents (i.e. receptionists) running outside or on its perimeter. An outside client first contacts a GCB agent (i.e. checks in) to ask for a connection to an inside server. After authenticating the client, the agent asks the insider to connect to the client. Then, a connection is made from the server to the client (i.e. insider escorts the visitor). From a different perspective, GCB decouples the direction of a connection from the role of an endpoint (i.e. whether it is a client or server) to handle the Internet asymmetry. Unlike the conventional Berkeley socket system, GCB decides the direction of a connection based upon the relative topology of communicating parties instead of their roles.

Through authentication using X.509 certificates and secure communication using session keys established during the authentication process, GCB arranges connections only for authorized applications. However, it lets applications communicate through existing middlebox openings (i.e. "outbound allowed") without changing a network perimeter—i.e. creating openings in the middlebox or installing new entities on the network perimeter. Therefore, a network perimeter is protected as securely as it was

before GCB is deployed. GCB can support any type of middlebox. It also has few restrictions on the type of applications it can support. It can be easily deployed because it neither requires root privilege nor modifies any sensitive component such as the operating system or the network perimeter. However, GCB is not gradually deployable because it uses a strong security mechanism to authorize applications. It has small overheads in connection setup and data transfer.

XRAY (middleboX traversal by RelAYing) [73] targets the "fixed white-list" and "fixed black-list" types of middlebox policies. XRAY is a proxy-based mechanism: the XRAY proxy, which is deployed as one of "fixed number of authorized points" or one that is not in "fixed number of unauthorized points", relays packets to/from next hop proxies or authorized applications. As parts of processing socket calls by authorized applications, XRAY builds on-demand overlay channels, each of which consists of multiple TCP or UDP connections concatenated by relay points. XRAY proxies add relay points between two points that cannot directly communicate (a connectivity perspective), or to a place where traffic control must occur (a perimeter defense perspective). Since the XRAY mechanism is absolutely generic and does not have any application dependency, one XRAY proxy can support many applications, avoiding the "too many authorized points" problem of the "fixed white-list" type.

Because each overlay link is authenticated using X.509 certificates and secured using session keys, only packets for authorized applications can pass through a middlebox and its XRAY proxy. Like GCB, XRAY can support any type of middlebox and has few restrictions on the type of applications it can support. It is easy to deploy, though not as easy as GCB. It requires the network administrator's commitment to create an opening at the middlebox and to install the XRAY proxy on the network perimeter. It is not gradually deployable because it uses a strong security mechanism to authorize applications as GCB does. Among our three systems, XRAY has the biggest overhead in most cases. However, this overhead is still reasonable (see the performance discussion in Chapter 7).

CODO (Cooperative On-Demand Opening) [74] targets the "dynamic white-list" type of policy. It dynamically adds and deletes middlebox rules for authorized applications. As

parts of processing socket calls by authorized applications, CODO builds on-demand communication channels by dynamically opening middleboxes in the communication path. Through secure control connections, CODO-enabled applications ask the CODO agent, add-on software for middleboxes, to build communication channels for them. Then, the CODO agents of intervening middleboxes collaborate to build requested channels. Because CODO uses the out-of-band method to learn an application's need for channels (i.e. openings), CODO agents need not have application intelligence nor investigate packets.

CODO agents and applications exchange CODO commands through control channels, which are authenticated using X.509 certificates and secured with strong security keys. Therefore, only authorized applications can have a CODO agent open a middlebox for them. Furthermore, each opening is made with the fully qualified addresses of a connection and exists in the middlebox only while the connection is open. Therefore, unauthorized applications can communicate through those openings only by a connection hijacking. CODO can only support middleboxes that can be programmatically controlled. However, it has few restrictions on type of applications that can be supported. It is a little more difficult to deploy CODO than XRAY because CODO requires installing software on middleboxes. It is not gradually deployable because it uses a strong security mechanism to authorize applications as GCB and XRAY do. Among three systems, CODO has the least overhead in most cases.

1.3 Condor as a Driving Application

Condor [9] [10] has been a driving application of this thesis. We use Condor to identify key requirements of applications and evaluate our approaches within its context, especially deployability and connectivity dimensions and other general characteristics such as scalability and fault tolerance. Condor gives one of the most challenging requirements, which no previous traversal system can satisfy. It uses many-to-many communications between its entities possibly managed by different organizations; it uses TCP and UDP; it uses both static and dynamic addresses; endpoint addresses are passed directly or indirectly to entities which need those addresses; a node can be a client, server, or both in communications. Furthermore, it has started using virtual machine

technologies [96] [97], increasing dynamism and complexity even further. We believe that Condor includes all the representative communication patterns and also believe that other applications can be easily supported by a traversal system that supports Condor. Condor is not the only application which requires such a generic traversal mechanism. Applications in Grid [7] [8], internet telephony [81] [82], and peer-to-peer (P2P) [11] [12] [13] [14] have similar properties as Condor.

The rest of the dissertation is organized as follows. Chapter 2 discusses some middlebox techniques and research into middlebox traversal and related fields. This chapter will show that middlebox traversal is understood very differently or even vaguely among people and that no previous system can support demanding applications such as Grid and internet telephony applications. In Chapter 3, we first give a new definition of middlebox traversal with some clarifications of our definition and present a framework of middlebox traversal. We also discuss representative previous systems under the framework. In Chapters 4 and 5, we present our middlebox traversal techniques and our approach to the development. Chapter 4 discusses the connection arrangement scheme that the three traversal techniques share as a common part. Chapter 6 discusses the implementation and challenges. Chapter 7 shows the results of our experiments, and Chapter 8 concludes the dissertation.

2. Background and Related Work

2.1 Middlebox Techniques

One of the slowest operations in middlebox's packet processing is *packet classification*, the operation to find all or the best rule that applies to a packet in question. Since every packet leaving or entering a network must be processed by a middlebox, the packet classification should be done very fast—its speed may have to be comparable to that of typical routing decisions. However, the packet classification is much more difficult than routing decisions because middlebox rules are much longer and more complex than routing rules, which are generally specified only with destination IPs. A middlebox rule is generally specified with source IP, destination IP, source port, destination port, flags (e.g. SYN and ACK flag of TCP), etc. Many rules contain ranges, wildcards, and logical operators. It is known that the general packet classification is slow or requires lots of memory [33]-[35]. For this reason, many studies have been performed to find efficient packet classification algorithms [36]-[43].

On the other hand, little progress has been made in enhancing the middlebox's ability to distinguish desirable from unwanted traffic. Stateful middlebox technology and recent efforts from commercial vendors have slightly improved such abilities in middleboxes. Stateful middleboxes keep track of the communication context of packet flows. After the first packet of a network flow is allowed to pass through a stateful middlebox, all subsequent packets in the flow are allowed to traverse it. These middleboxes not only improve the quality of packet filtering but also filtering speed. The quality is improved because they accurately distinguish legitimate packets from those out of the context. For example, they keep track of a TCP connection status and pass only packets which are legitimate for the current status of the connection—e.g., packets only with a valid sequence number and flags are allowed; an ICMP [46] message is allowed only when it carries a control information for the connection and it is a valid type for the current

connection status. Stateful middlebox technology also improves filtering speed because the slow packet classification need be done only for initial packets. When a stateful middlebox allows a packet to traverse (through the expensive packet classification), it creates a state entry for the flow that the initial packet belongs to. Packets are first checked if they belong to an existing state and go through the packet classification only when they fail to match a state [43] [32]. Since the state entry has fewer fields than middlebox rules and is in a well-defined format, the search for a matching state is much faster than the packet classification.

A few recent commercial middleboxes improve filtering quality by adding application intelligence to the filtering logic. These devices have the knowledge of the communication protocol that the intended application uses. Using this knowledge, these devices define legitimate communication patterns and block traffic out of those patterns. For instance, recent Checkpoint firewalls [17] [18] define formats and sequences of HTTP [19] messages that any legitimate web traffic can have. In order to pass web traffic but block traffic disguised to look like web traffic, these devices passes traffic with port 80 only if it matches one of predefined formats and patterns.

Bellovin suggested an approach called “distributed firewall” [89]. In this approach, each host in a network operates as a middlebox of a single-node network but enforces the site policy which is centrally defined and distributed to them. Distributed firewalls have several advantages over conventional (i.e. centralized) middlebox approaches. They allow topology-independent enforcement. They do not have a single point of failure or a chokepoint that can be a performance bottleneck. Another benefit, which is especially relevant to our discussion, is that enforcement points are closer to applications and may do better filtering because more information about applications may be available. However, this approach can be used only by tightly controlled networks. The site policy that may be stored in each host should not be modified by unauthorized personnel; new patches and fixes to the operating system must be centrally installed so as to maintain each host as hardened as centrally managed middleboxes; and so on.

2.2 Middlebox Traversal and Related Techniques

Many approaches to facilitate communications through middleboxes have been

studied from both academia and industry. Several new Internet architectures that invite the middlebox as a legitimate citizen were proposed. A few middleware systems have been developed to deal with the connectivity/security problem related to middleboxes. Many ad hoc or application specific approaches were also developed. The following sections briefly discuss those systems for each category. We should note that our discussion here is far from being exhaustive because so many systems have been developed with little public announcement.

Most of the previous systems have been developed with no or little consideration of the security of the networks being traversed. For this reason, many systems can be used as an attacker's vehicle as well as as a convenience tool for benign users. Furthermore, people understand middlebox traversal very differently or even vaguely. Some systems, though described as such, can hardly be considered as traversal systems.

2.2.1 Architectural approaches

Some Internet architectural efforts are relevant to middlebox traversal, though none of them directly deals with the problem. TRIAD [57] and IPNL (IP Next Layer) [58] propose an Internet architecture which has a new layer between layer 3 (i.e. IP) and 4 (i.e. TCP/UDP). They envision the Internet as a collection of IP networks, which they call realms. Within a realm each node must have a unique IP address and routing is performed by the conventional IP layer. However, nodes in different realms can use the same IP—i.e. they propose a private IP scheme in units of realms—and the routing between realms is performed by the new layer. The layer 4 routers—i.e. entities that route packets between realms—are similar to NAT devices. These systems elegantly solve the private IP address issue and propose a way that hosts in different realms (i.e. in different address spaces) can freely communicate. However, they do not deal with how middleboxes (i.e. layer 4 routers) can pass packets for authorized applications while blocking those for unauthorized ones.

Balakrishnan et al. [48] propose a new naming scheme to deal with the problem of the two layer—i.e. DNS and IP—naming scheme of the current Internet. They claim that the current naming scheme does not support mobility, content replication among multiple hosts, and multi-homed machines well. To remedy this problem, they suggest using 4

layers for naming. At the top layer, ULD (User Level Descriptor) identifies data objects and services in a human-friendly way such as search string and email address. SID (Service Identifier) is an opaque and host-independent identifier of an object so that an object can be replicated in multiple hosts. HID (Host Identifier) identifies a host in the Internet such that multi-homed and mobile hosts can be uniquely identified. Finally, the conventional IP address at the lowest layer identifies a location in the Internet. They also propose resolution layers that translate names between adjacent layers. As a way to invite middleboxes into the Internet architecture, the resolution layer between HID and IP layer can translate an HID into a concatenation of multiple IP addresses that packets to/from a host must pass through. This approach can be used to specify what middleboxes must be passed through to reach a host. Like TRIAD and IPNL, however, this does not deal with how to authorize packets passing through those middleboxes.

IPv6 [59] is beginning to be widely deployed. It provides enough address space and enables easy network management. Thus, it solves most problems that private IP and NATs try to solve. Therefore, middlebox traversal may become unnecessary for private networks that switch to use IPv6. However, it is still questionable whether IPv6 can replace NATs completely. Furthermore, firewalls will certainly exist after the full deployment of IPv6.

2.2.2 Middleware approaches

STUN (Simple Traversal of UDP through NAT) [53] allows an endpoint to discover whether it is behind a NAT (and the type of the device) and then enables it to receive inbound UDP packets through the NAT. STUN-enabled endpoints, called STUN clients, exchange several UDP messages with their agents, called STUN servers. Through this probing, a STUN client learns not only the type of NAT device on the communication path but also the addresses that the NAT allocates for address translation. Then the client passes the address that it learned to its peers so that they can send UDP messages to the client via that address. STUN targets the “outbound only” middlebox policy. It only works for UDP and, still worse, only works for very rare type of NAT, what they call "full-cone" NAT [53].

TURN (Traversal Using Relay NAT) [54] is a proxy-based system. An endpoint

(TURN client) behind a NAT opens a channel—i.e. it makes a TCP connection or sends a UDP message—to its agent (TURN server) in the public network. Next, the agent creates a proxy socket and sends the endpoint the address that the proxy socket is bound to. Then, the endpoint passes the returned address to *one* peer. The connection and data (from *the peer*) to the proxy socket is forwarded to the endpoint through the channel made in the first step. After receiving a TCP connection or a UDP message from a peer, the agent "locks down" the proxy socket so that no other peer can connect or send UDP messages to the socket. TRUN targets the "outbound only" middlebox policy. Unlike STUN, it supports both TCP and UDP communications and works with any type of NAT (and firewall). However, it can support only cases that a TRUN client expects a single connection from a single designated outside client. It could not support general server applications that must be able to receive connections or messages from arbitrary clients.

SOCKS [56] enables communications through a middlebox by a proxy that relays connections and data. The proxy, called SOCKS server, generally runs on a network perimeter and creates proxy sockets on demand from outside endpoints that want to communicate with endpoints inside the perimeter. Connections and data are relayed via those proxy sockets. The proxy is application-independent and can be configured to use strong security mechanisms to authenticate applications. SOCKS targets the "fixed white-list" and "fixed black-list" type of middlebox policy. Outside endpoints can have proxy sockets created for them not only for inbound but also outbound connections. However, an outbound connection (via the proxy) is possible only when it is a part of a session that has been initiated by an inbound connection—e.g. FTP [45] data connection from a server to a client after the control connection is made the other direction. Therefore, it works only for client-server model applications. It particularly supports only client-server model sessions between outside client and inside server. Another problem of SOCKS is that it cannot support private networks. (Note that in STUN and TURN, inside endpoints use proxy socket addresses to uniquely identify themselves in the Internet.) In SOCKS, inside endpoints are identified by their real addresses and therefore two private endpoints may be ambiguously identified by the same address.

Many applications utilize middlebox openings made for other applications and disguise their traffic to look like other application's traffic. For instance, peer-to-peer

systems such as JXTA [72] and P2P file sharing systems use port 80 and wrap their traffic in HTTP messages to traverse middleboxes that allow Web traffic. SoftEther [69], a VPN system, disguises its traffic to look like web, SSH [20], or SOCKS for the communication between VPN nodes over middleboxes. By exploiting common middlebox configurations, this mechanism may work in many cases without any change to middleboxes. However, this mechanism exploits middlebox openings created for other purposes. Network administrators do not like this mechanism in general and try to prevent it from working.

UPnP (Universal PnP) [60] is a forum of major OS and device vendors. Its major idea is to apply the idea of PnP (Plug and Play) to network environments. With UPnP, network devices (called "devices") such as network printer can be seamlessly attached to and detached from a network, and be controlled by hosts or other devices (called "control points") in a standard way [61]. If a middlebox is UPnP-enabled, it can be detected and dynamically controlled by end hosts or applications. However, it should be understood as a component technique to dynamically control middleboxes rather than an integral traversal mechanism. UPnP is developed for easy management of home or small business networks. Within its architecture, control points (i.e. end hosts or applications in our context) can control only local devices (middleboxes) in the same network or in the same organization. Therefore, in middlebox traversal mechanisms using UPnP, applications can open/close local middleboxes but cannot ask remote middleboxes to allow them access.

Realm Specific IP (RSIP) [62] [63] has been proposed by IETF as an alternate to NAT, especially in order to deal with problems concerning IPSec [49] [50] and inbound connections. A host within a private network leases public addresses—IP and port pair—from its RSIP server and uses those address as network endpoint identities. When the lessee needs to send a packet to a public peer, it prepares the packet as if it is from one of those leased addresses and then sends it to the RSIP server, through a tunnel to the server. Upon receiving a packet through the tunnel, the server gets rid of the tunnel header and forwards it to the public peer. Inbound communications, including replies from the public peer, are handled in the reverse way. When the RSIP server receives a packet delivered to a leased address, it forwards the packet to the leaser, through the tunnel to the leaser. In

addition to supporting inbound communications, RSIP solves NAT's incompatibility with IPSec. Since NATs alter packet headers, those packets fail the end-to-end integrity check by IPSec. RSIP solves this problem because the RSIP server relays original packets untouched. Since it supports inbound connections into private networks, some people consider RSIP to be a middlebox traversal mechanism. However, it does not address authorization issues at all. RSIP should be regarded as achieving the same goal as TRIAD and IPNL without changing the Internet. Another problem of RSIP is that it does not handle firewall problems at all.

2.2.3 Application specific and ad hoc approaches

ALG (Application Layer Gateway), add-on software for middleboxes, understands communication protocols and semantics of an application or a group of applications. Using this knowledge, it analyzes the payloads of the application traffic—generally connections to a well-known IP and port—and dynamically creates/destroys openings so that the application may communicate using subsequent connections. For example, FTP ALG analyzes control channels—i.e. connections to port 21—and creates opening for data channels. It may also modify payload, if necessary, to facilitate end-to-end communications. This mechanism does not require any change of applications. However, it may not work if application payload is encrypted. Furthermore, it requires installing new ALGs to middleboxes for supporting new applications. To solve this problem, IETF MIDCOM group developed a standardized way [55] to control middleboxes so that new ALGs can be added without modification of middleboxes. However, it still requires the development of new logic for new applications.

Application proxies such as web proxies [76] and mail gateways relay traffic for specific applications to/from nodes behind a middlebox. Proxies can be configured to block traffic with certain characteristics. For example, mail gateways may understand various protocols related to email [22]-[27] and drop mails containing dangerous attachments. Application proxies can filter traffic more effectively than middleboxes because they may use customized filters for the intended applications.

One way to handle the middlebox traversal problem is to change applications to use a few communication patterns that intervening middleboxes would pass. Napster [77]

server acts as a connection broker for its clients. Normally it arranges that a downloading site make a connection to an uploading site. However, when the uploader is behind a middlebox, it asks the uploader to push files to the downloader in the public network. Old versions of Gnutella [78] also use the same idea, but without any server. When an uploader is behind a middlebox, the downloader in the public network asks the uploader to actively push a file.

3. A Framework of Middlebox Traversal

This chapter presents a framework of middlebox traversal, which provides a foundation of our development of traversal mechanisms. First, we consider middlebox traversal both from application's connectivity and perimeter defense perspectives and redefine it to take both perspectives into account. Second, we classify middlebox policies into seven types, which we believe cover most typical policies. Third, we identify dimensions that address various aspects of a middlebox traversal mechanism. Lastly, we analyze representative mechanisms under the framework we present in this chapter.

3.1 Middlebox Traversal from Two Perspectives

To the best of our knowledge, nobody has precisely defined middlebox traversal. However, we believe that most people understand it from an application's connectivity perspective with little consideration of perimeter defense and view middlebox traversal as *a mechanism that enables applications to communicate through middleboxes*. This conventional viewpoint has a few problems.

First, it does not distinguish between applications that network administrators want to allow to pass through middleboxes and those that they want to block. Actually, many previous mechanisms were developed based upon this viewpoint and can be used as an attacker's vehicle. For this reason, network administrators generally consider middlebox traversal as a way to break the purpose of middleboxes. Since middleboxes are used as a component of perimeter defense, we must consider middlebox traversal from perimeter defense perspectives as well as from application's connectivity perspectives. We claim that middlebox traversal must be defined so that only authorized applications benefit from it.

Second, when a traversal mechanism enables an authorized application to communicate through a middlebox, it may inadvertently allow unauthorized applications

to pass through the device. Therefore, the definition must include the fact that traversal mechanisms should not or should only minimally increase the chance that unauthorized applications can pass through the middlebox together with authorized applications. To this light, we define middlebox traversal as follows:

> **Definition-3.1** *Middlebox traversal* is a technique that enables an authorized application α to communicate through a middlebox M with a minimum increase in the chance that unauthorized applications can also communicate through M.

The middlebox is a part of a network perimeter, in which many parts may interact in a complex way. We realize that the middlebox traversal problem is very subtle and needs to be further clarified:

- A perimeter defense policy may define a set of authorized applications that can pass through a network perimeter. In turn, a middlebox, as a component of perimeter defense, may be given a set of authorized applications that can communicate through it. We define these applications as "authorized". Any other applications are defined as "unauthorized".
- Authorized applications are not necessarily benign. For example, an authorized application may become malicious if it is contaminated by a virus. Middlebox traversal generally cares only whether an application is authorized or not. Other components of perimeter defense such as NIDS [30] [31] and anti-virus mechanisms should block malicious traffic. A middlebox traversal mechanism can be plugged into an overall perimeter defense that passes only authorized and benign traffic.
- Packets that pass through a middlebox may still be dropped by other parts of a perimeter such as NIDS and anti-virus. Furthermore, some packets for authorized applications may be blocked even by middleboxes. For instance, a middlebox may drop packets for an authorized application after a bandwidth limitation is reached. A middlebox may also block oddly fragmented packets no matter whether they are for authorized applications or not [86]-[88]. Middlebox traversal does not handle such issues.

According to our definition, middlebox traversal is not only an enabling technique that helps applications to communicate through network perimeters but also a security

technique that network administrators can use as a component of perimeter defense.

3.2 Classification of Middlebox Policies

This section classifies middlebox policies into seven types. Four types (i.e. Type 1 ~ 4) are practical and require middlebox traversal mechanisms in general. They together cover most typical policies. Other three (i.e. Type A, B, and C) are added for reference or completeness of our classification.

3.2.1 Type A: Nothing Allowed

This type permits no traffic through a middlebox. This type of policy protects the network perimeter most securely but makes the network isolated! Therefore, this policy is meaningful as a reference or starting point rather than a practical choice.

3.2.2 Type B: Everything Allowed

This type allows every point (i.e. host or IP, port, subnet, or combination of them) to communicate through a middlebox. It is very unlikely that an organization installs a middlebox and allows every inside and outside point to pass through it. Furthermore, a middlebox traversal mechanism may not be needed for this type of policy. As type A, this type is only meaningful as a reference.

3.2.3 Type 1: Outbound Only

This type permits outbound (i.e. to the world) but no inbound (i.e. from the world) connections. Any application inside the network perimeter can send packets to outsiders through the middlebox. However, inbound packets can pass through the middlebox only if they are reply to an outbound packet. For TCP, this means that only connections initiated by insiders are possible. For UDP, when an insider sends a UDP message to an outsider, the outsider can send UDP messages back to the insider within a certain timeout.

With this type of middlebox policy, clients in a network can freely communicate with outside servers. However, servers in a network may not communicate with outside clients because initial packets are always sent from a client to a server in the Berkeley socket system [84] and systems derived from it. We need a middlebox traversal mechanism to allow authorized inside servers to communicate with outside clients.

Benefits of this type are as follows:

- Today, most middleboxes are stateful. These devices keep track of the state of network connection and are able to distinguish between reply and unsolicited packets very well. For example, many stateful middleboxes understand the TCP protocol and take inbound packets as reply packets only if they have correct source and destination addresses, valid sequence numbers, and flags. Therefore, this type of policy can be effectively enforced by using current middlebox technology.
- Some organizations use private IP addresses and NAT to deal with an IPv4 address shortage rather than for security reasons. This type may be the most appropriate type for such organizations.

However, this type has the following weakness as well:

- This type puts much security responsibility on endpoints. It may provide a perimeter defense secure enough for networks within which endpoints are well hardened and are managed by the network administrator (e.g. dedicated clusters). However, it is difficult with this type of policy to achieve perimeter defense secure enough for networks where endpoints are managed by end users who may not be as security conscious as network administrators and may install insufficiently hardened applications.

3.2.4 Type 2: Fixed White-List

This type permits a fixed small number of authorized points to pass through the middlebox. These points can be inside, outside, or on a network perimeter and can be either endpoints or relay points via which others can communicate.

Middlebox traversal is required when the point a desired application is supposed to use cannot be easily included in the white-list without allowing other applications to communicate through the middlebox. This may be because other applications are also allowed to use the point that the intended application is supposed to use or because network administrators do not know what points a desired application may use a priori.

Benefits of this type are as follows:

- Organizations having this type of policy often use authorized points to relay traffic

for specific applications. In this case, relay points generally understand application traffic better than middleboxes because they usually terminate transport connections and security associations and because they are often customized to understand applications' communication semantics.

- Since a relatively small number of openings need be made at middleboxes, network administrators may be able to closely watch traffic through those openings. Middleboxes are considered as one of most sensitive components of the network perimeter. Also, middleboxes are shipped with many tools for monitoring, logging, and analyzing traffic. Therefore, having small number of openings at the middlebox can be an advantage even if many endpoints may communicate through these openings (maybe via relay points).

This type has the following weaknesses:

- Some general networks may have many applications using various points. In this case, the network must define too many authorized points.
- It may not be possible for certain networks to prevent unauthorized points from communicating through the openings made for authorized points.

3.2.5 Type 3: Fixed Black-List

This type blocks traffic from a fixed number of dangerous points but permits all the others. This type of policy is not recommended for general cases because it may define indefinite authorized points. However, this may be an appropriate type when most neighbors are trusted. In a nested network, for example, this type may be used to prevent a few publicly accessible nodes inside the outer perimeter from accessing nodes inside the inner perimeter.

Middlebox traversal mechanisms may be needed when the point on which a desired application is running is included in the black-list and cannot be removed easily without allowing other applications to pass through the middlebox.

3.2.6 Type 4: Dynamic White-List

This type allows a dynamic set of points to pass through the middlebox. As middleboxes become smarter with newer capabilities such as application or protocol

awareness, sites define only higher level policies and let middleboxes dynamically decide what points can communicate through them. For example, administrators specify a high level policy allowing VoIP (Voice over IP) traffic using SIP (Session Initiation Protocol) [81] and let SIP-aware middleboxes analyze call establishment traffic and dynamically create openings for subsequent voice or other media communications [79] [80].

A middlebox traversal mechanism is needed when a middlebox does not have necessary intelligence for a desired application. A traversal mechanism should learn that an authorized application needs to communicate through the middlebox and dynamically creates/deletes pinholes at the device.

This type has a few benefits:

- This type benefits from advanced middlebox technologies and places fewer burdens on network administrators. It generally allows smaller middlebox rule set and, therefore, reduces human errors. It is generally believed that middlebox configurations are too complex and contain many errors. Avishai Wool also confirmed this through a quantitative study of middlebox configurations [94].

However, it has several weaknesses and issues as well:

- Since openings are dynamically made by the middlebox itself, network administrators may have little knowledge of the low level consequence of high level policies. This may make it difficult to notice and track security problems.
- As new applications or protocols need be supported, new intelligence may have to be added to middleboxes.

3.2.7 Type C: Dynamic Black-List

This type is the opposite of “dynamic white-list”. It defines a dynamic set of points that cannot communicate through the middlebox. A site may have security tools that watch for or lure attack attempts. When an attack is detected by such a tool, it may add the point that the attack is launched from to a black-list.

We include this type only to make our classification as complete as possible. When the endpoint that a desired application is using is included in the black-list, we may need a middlebox traversal mechanism that removes the endpoint from the list. However, we

do not believe that such traversal mechanisms should be used. Points are added to a black-list for (strong) reasons. Even if a point is added to a black-list due to a false alarm, it should be removed from the list through a human investigation rather than by a traversal mechanism.

3.3 Major Dimensions of Middlebox Traversal

In this section, we identify major dimensions which address aspects of a middlebox traversal mechanism. We group related dimensions into connectivity, security, deployability, and performance classes. Connectivity and security classes particularly address the traversal mechanism in an application's connectivity and a perimeter defense context, respectively.

3.3.1 Connectivity

Dimensions in this class address the versatility or limitations of a traversal mechanism in enabling authorized applications to communicate through various middleboxes. Traversal mechanisms have many restrictions and limitations. No single system supports all types of middlebox policies we identified; some mechanisms support only certain types of applications or protocols; some assume specific middlebox behaviors or configurations. Dimensions in this class address these issues.

- **Application coverage**. This dimension addresses the versatility of a traversal mechanism in supporting applications. A traversal mechanism must support as many applications as possible. However, mechanisms have many restrictions in supporting applications. For instance, several existing mechanisms can support only applications using TCP or UDP but not both; some support only a specific application or applications using a specific protocol such as HTTP; and some others can support only client-server model applications.
- **Middlebox policy coverage**. This dimension explains the types of middlebox policies (Section 3.2) that a traversal mechanism may support.
- **Middlebox coverage**. This dimension addresses the versatility of a traversal mechanism in supporting various middleboxes. Traversal mechanisms have many restrictions in supporting middleboxes. Some mechanisms support only firewalls or

NAT but not both; some targets specific middlebox behaviors; others support middleboxes that can be programmatically controlled.

3.3.2 Security

Security dimensions address potential impacts of a traversal mechanism on perimeter defense and endpoint security. A traversal mechanism can be a security tool for perimeter defense because it may help a network perimeter to distinguish between traffic for authorized and unauthorized applications. A traversal mechanism may require applying changes to the network perimeter or end host such as the installation of new software or the modification of certain components. Also, it may affect end-to-end security by securing a certain or the entire part of an end-to-end channel. Security dimensions address these issues.

- **False positive traffic**. This dimension addresses the possibility that unauthorized applications can communicate through openings a traversal mechanism creates or utilizes for authorized applications. If a traversal mechanism creates or utilizes openings that unauthorized applications cannot pass through (with a reasonable cost), it can be used as a security tool for a strong perimeter defense.
- **Granularity of traffic control**. This dimension explains the granularity with which a traversal mechanism can control traffic. Representative granularities (not necessarily in increasing order) that we have seen in traversal mechanisms are as follows: networks—i.e. a traversal mechanism passes packets to/from a certain network but blocks those to/from other networks, protocols, hosts, applications, and implementations or versions of an application. Fine-grained control means flexibility in enforcing security policies. A traversal mechanism that allows fine-grained control can be used to enforce various perimeter defense policies.
- **Perimeter softening**. This dimension addresses the potential effect of a traversal mechanism on the hardness of a network perimeter. Components of a network perimeter are usually hardened through various protective measures and through software engineering practice. Some middlebox traversal mechanisms require installing new components to or modifying existing components of a network perimeter, which generally result in changing the hardness of the perimeter. Several

issues must be addressed to discuss this dimension. The most important issue may be the hardness of new entities that a traversal mechanism adds to a perimeter. Unfortunately, this is mostly a software engineering issue. However, we can still use some general criteria, e.g. how simple new entities are and how well-defined their functionalities are. Since not every part of a network perimeter has the same consequence when it is attacked, the size and scope of changes to a perimeter are also important factors.

- **Endpoint security**. This dimension addresses the potential effect of a traversal mechanism on endpoint security. A traversal mechanism may require changing the end host. Changes may include the modification of the operating system, the installation of libraries, etc., which generally results in changing the hardness of the host. Similar issues as "perimeter softening" dimension must be addressed—i.e. the hardness of new entities and the sensitivity of entities being modified or replaced. A traversal mechanism may also affect the security of end-to-end communication, too. As a part of providing connectivity to applications, some traversal mechanisms secure the whole or a part of end-to-end communication channels.

3.3.3 Deployability

Deployability dimensions address expected barriers for a traversal mechanism to be deployed and accepted by user communities.

- **Single deployment**. This dimension addresses barriers that a traversal mechanism may have when it is deployed at a single site. A traversal mechanism may require installing new entities or modifying existing ones on network perimeters or endpoints. Mechanisms requiring installation or modification of entities in sensitive places such as middleboxes or operating systems are less deployable than those requiring no modification or modifications in less sensitive places. Similarly, mechanisms requiring higher privileges to operate are less deployable than those requiring lower privileges.
- **Multiple deployment**. In order to support communications between multiple sites possibly with middleboxes on their network perimeters, middlebox traversal mechanisms generally need global deployment that may require agreements and

collaborations between sites. This dimension addresses barriers that a traversal mechanism may have when it is deployed to multiple sites. Representative issues that must be addressed are the number of sites that must be involved in an agreement or collaboration (e.g. no, bilateral, or multi-lateral agreements or collaborations), personnel who must be involved in an agreement or collaborations (e.g. end users or network administrators), and whether a mechanism is gradually deployable or not. In general, global deployment is not necessary (i.e. no barrier) when outsiders (e.g. applications, hosts, middleboxes managed by other sites) need not be changed.

3.3.4 Performance

Performance dimensions address expected overheads of a traversal mechanism in connection setup (i.e. latency) and data communication (i.e. bandwidth).

3.3.5 Other dimensions

In addition to dimensions that we present in this section, a traversal mechanism must be discussed or evaluated for dimensions such as scalability and fault tolerance. We do not discuss such dimensions here because they address aspects of general systems rather than middlebox traversal system in particular.

3.4 Analysis of Existing Systems

This section discusses popular middlebox traversal approaches under the framework that we present in this chapter. For each mechanism, we briefly explain how it works and discuss it for each dimension in 3.3.

3.4.1 Manual opening

- **General description**. The network administrator opens the middlebox for a set of addresses—generally an IP and a range of ports—that an authorized application is expected to use. If the application uses dynamic addresses, it is changed to use only the addresses within a given range.
- **Formal description**. A set X of addresses is statically allocated to an authorized application α. α is modified, if necessary, to use only addresses within X. Any packet p whose source/destination address is $x \in X$ is considered as a packet for α and

allowed to traverse the middlebox.

- **Connectivity**
  - **Application coverage**. This approach has little restrictions on types of applications it can support. However, it may not be easy to calculate the correct address range (i.e. X in the formal description) for applications that use dynamic range of addresses—i.e. the number of addresses it uses changes very much over time. If the range is too small, the application may run out of addresses. If the range is too big, the middlebox is opened wider than necessary.
  - **Middlebox policy coverage**. Fixed white-list and fixed black-list.
  - **Middlebox coverage**. This mechanism may work for any type of middlebox.
- **Security**
  - **False positive traffic**. We cannot generally prevent unauthorized applications from using addresses allocated to authorized ones. In other words, it is generally impossible to make unauthorized applications respect the address allocation to authorized applications. If an unauthorized application β uses an address $x \in X$, which is allocated to an authorized application α, packets for β will be considered as those for α and will be allowed to traverse the middlebox.
  - **Granularity of traffic control**. Control in unit of applications or versions is possible by allocating different address ranges to different applications or versions.
  - **Perimeter softening**. It does not soften the network perimeter because it makes no change to the network perimeter.
  - **Endpoint security**. This mechanism softens end hosts very little, if any. It makes very simple changes to applications or does not change at all. It does not provide any level of end-to-end security.
- **Deployability**
  - **Single deployment**. This mechanism requires a few rule changes at middleboxes, which requires administrator intervention. Applications must be changed in general.

- **Multiple deployment**. When sites control traffic only based upon the address allocation to local applications and/or well-known addresses being used by outsiders—i.e. if dynamic addresses being used by outsiders do not matter, this mechanism does not require global deployment. Otherwise, bilateral agreements must be made between sites to restrict application's address usage and address remapping by NAT devices.

- **Performance**. Since endpoints directly communicate with each other, this mechanism has very little overhead in general in connection setup and data communications.
- **Other issues**. This mechanism has a resource usage problem. It restricts authorized applications to use only addresses within their allocated ranges. On the other hand, other applications are free to use any address including those in the ranges. Therefore, authorized applications must contend with unauthorized applications for addresses allocated to them. It may happen that an authorized application becomes short of addresses because addresses within its range are taken by other unrestricted applications, while there are plenty of addresses outside the range available in the host.

3.4.2 Application proxy

- **General description**. The network administrator installs, generally in the DMZ, a proxy for an application or a group of applications. Often a proxy is a regular application entity (mostly in store-and-forward applications such as email) or one with some extra functionality such as forwarding requests and servicing requests from a local cache. The administrator also configures middleboxes to pass traffic to/from a proxy. For end-to-end transactions across a network perimeter, applications are configured or modified to contact appropriate proxies instead of peers. Proxies use various mechanisms to authenticate and authorize applications.
- **Connectivity**
  - **Applications coverage**. A proxy mechanism can support only one or a few applications using similar protocols. In general, each application requires its own proxy mechanism to be developed and deployed.

- **Middlebox policy coverage**. Fixed white-list and fixed black-list.
- **Middlebox coverage**. This mechanism may work for any type of middlebox.

▪ **Security**

- **False positive traffic**. If a proxy is attached to an opening (i.e. it is installed right next a middlebox so that there is no exit, entry, or alteration of packets in the path between the middlebox and the proxy) and uses a strong security mechanism to authenticate traffic, only authorized applications can communicate through the opening that the proxy is attached. On the other hand, if a proxy (e.g. Web proxies) authenticates traffic based upon a traffic pattern such as payload format and the sequence of commands exchanged, unauthorized applications may be able to fabricate their traffic to look legitimate and can pass it through. If a proxy is installed far from a middlebox so that packets can be injected into the path between the middlebox and the proxy, unauthorized applications may pass through the middlebox no matter what mechanism the proxy uses to authenticate traffic.
- **Granularity of traffic control**. Mechanisms of this type may vary in this dimension depending on how a proxy authenticates traffic. If a strong security mechanism is used, fine-grained control is possible. For example, different applications, versions, or implementations of an application can be given different certificates and the proxy can distinguish endpoints based upon certificates they provide. On the other hand, if a traffic pattern is used for authentication fine-grained control may not be possible because different versions or implementations of an application may have the same traffic pattern.
- **Perimeter softening**. This mechanism may add new components to the network perimeter and may soften the perimeter. In particular, the proxy generally performs application logic, which may be complex and not well-defined. Also, each time a new application need be supported a new proxy may have to be added to the perimeter, potentially adding new vulnerabilities.
- **Endpoint security**. This approach does not require any change to sensitive components of end hosts such as the operating system. For some cases (e.g. Web

proxy, email gateway), applications just need to be appropriately configured and therefore end hosts are not softened at all. However, in many cases, applications must be modified for indirect interaction with peers via the proxy. The authentication by the proxy may improve end-to-end security at some level because all or some parts of the end-to-end channel are secured in some systems.

- **Deployability**
  - **Single deployment**. Every time a new application need be supported, this mechanism may have to install a new proxy, which may require administrator intervention.
  - **Multiple deployment**. This mechanism may or may not require global deployment depending on the nature of targeted applications. Also, individuals who must be involved in an agreement may vary depending on targeted applications.
- **Performance**. Since the proxy often terminates transport connections and security associations, this mechanism has a fair amount of overhead in both connection setup and data communication: instead of one direct connection between endpoints, multiple secure connections must be made (high latency); packets must traverse protocol stacks up and down and must be encrypted and decrypted at the proxy (low bandwidth). However, if a smart proxy is used, this mechanism may shorten interaction paths and provide even better performance than direct communications. For example, Web proxies cache popular web pages and serve requests from local caches.

3.4.3 ALG and MIDCOM

- **General description**. The agent (i.e. ALG or MIDCOM agent) watches traffic passing through the (primary) openings, which is generally made for initial communication of the targeted application. Through the investigation of the traffic, it notices that the application needs more openings for subsequent communications and dynamically creates (secondary) openings accordingly. The agent dynamically deletes secondary openings when the application finishes communicating through them. Note that primary openings are static and remain in the middlebox over multiple creations

and deletions of secondary openings. Therefore, this approach controls traffic through primary openings in the same way as manual opening (3.4.1). We regard the primary openings as a way of specifying high level policies for the "dynamic white-list" middlebox policy type and only discuss the way that traffic is enabled through secondary openings.

- **Formal description**. By monitoring traffic through primary openings, the agent learns that an authorized application α uses (or will use shortly) a set X of addresses. Using this knowledge, any packet p whose source/destination address is $x \in X$ is considered as a packet for α and allowed to traverse the middlebox.
- **Connectivity**
  - **Applications coverage**. For this mechanism to be realistic, the application conversation through the primary openings must be simple. If an application's conversation is very complex and dynamic, it will be difficult to develop an agent that can monitor and analyze the conversation with a reasonable overhead. Also, the conversation must not be encrypted.
  - **Middlebox policy coverage**. Dynamic white-list.
  - **Middlebox coverage**. This mechanism requires adding a new ALG for each application. Therefore, it supports only cases that new software can be easily added to middleboxes. MIDCOM supports only MIDCOM-enabled middleboxes.
- **Security**
  - **False positive traffic**. The agent learns address usage of an application by analyzing the conversation through primary openings. Therefore, unauthorized applications can inject packets into the conversation and deceive the agent to create openings for them. Also, address spoofing attackers may be able to communicate through openings created for authorized applications.
  - **Granularity of traffic control**. Traffic can be controlled at the granularity of applications. If different versions or implementations of an application have different conversation patterns, though not usual, it may be possible to control at the granularity of versions or implementations.

- **Perimeter softening**. ALG adds new software (or firmware) for each authorized application to the middleboxes, which are generally considered one of the most sensitive components of the network perimeter. MIDCOM adds new software for each authorized application to the network perimeter. Therefore, this mechanism may soften network perimeters each time a new application is supported.
- **Endpoint security**. This approach does not change end hosts at all. It does not provide any level of end-to-end security.

- **Deployability**
  - **Single deployment**. Every time a new application need be supported, a new ALG must be added to middleboxes, which network administrators are reluctant to change. MIDCOM alleviates this problem a little. However, still a new agent must be added on the network perimeter.
  - **Multiple deployment**. This mechanism does not require global deployment.
- **Performance**. Since endpoints directly communicate with each other without any modification, this mechanism generally adds small overhead to connection setup and data communication. However, the packet analysis by an agent may slow down both connection setup and data transfer.

3.4.4 STUN

- **General description**. Outbound communications are done as usual. A STUN client (i.e. a server application) sends a UDP message to its STUN server through a middlebox. The middlebox creates an opening so that any remote client (i.e. any client application in the public network) can send UDP message to the client. (Only a small number of middleboxes, called “full cone” middleboxes [53], work this way.) A remote client sends UDP messages to the STUN client through the middlebox opening.
- **Connectivity**
  - **Application coverage**. This approach supports only applications using UDP.
  - **Middlebox policy coverage**. Outbound only.
  - **Middlebox coverage**. This mechanism works only for full cone middleboxes.

- **Security**
  - **False positive traffic**. When an authorized application α sends an outbound packet to β, any packet addressed to α, including replies from β, is allowed to pass through the middlebox. Therefore, unauthorized packets (i.e. neither outbound nor reply packets) can easily pass through the middlebox.
  - **Granularity of traffic control**. Fine grained control is possible. A site can make certain applications or versions of them STUN-enabled.
  - **Perimeter softening**. This mechanism does not change the network perimeter.
  - **Endpoint security**. This requires applications to become STUN-enabled. However, it does not require operating system changes. It does not provide any level of end-to-end security.
- **Deployability**
  - **Single deployment**. Authorized server applications inside the network perimeter must become STUN-enabled. The STUN server must be installed outside the perimeter, which may not require administrator intervention in general.
  - **Multiple deployment**. This does not require global deployment.
- **Performance**. Applications communicate with each other directly. Therefore, it generally has small overhead in connection setup and data communication.

3.4.5 TURN

- **General description**. Outbound communications are done as usual. An inbound connection is enabled as follows. An inside server (i.e. TURN client) opens a channel to its TURN server by making a TCP connection or sending a UDP message. The TURN server creates a proxy socket on the public network for the server application. A client connects to or sends messages to the proxy socket instead of the inside server directly. The TURN server then “locks down” the proxy socket so that no other clients can talk to the server application via the proxy socket. The TURN server then forwards packets to the server through the channel that was opened by the server application.
- **Connectivity**

  - **Applications coverage**. This approach allows a TURN client to accept only one connection from a single remote client. General server applications that expect multiple connections from undefined clients cannot be supported.
  - **Middlebox policy coverage**. Outbound only.
  - **Middlebox coverage**. Any middlebox can be supported.

- **Security**
  - **False positive traffic**. The channel between the TURN client and the TURN server can be secured. Therefore, only connections to authorized applications can be arranged by the TURN server.
  - **Granularity of traffic control**. Fine grained control is possible. A site can make certain applications or versions of them TURN-enabled.
  - **Perimeter softening**. This mechanism does not change a network perimeter.
  - **Endpoint security**. This requires applications becoming TURN-enabled. However, it does not require operating system changes. It does not provide any level of end-to-end security.
- **Deployability**
  - **Single deployment**. Authorized server applications must become TURN-enabled. The TURN server must be deployed outside the network perimeter, which may not require administrator intervention in general.
  - **Multiple deployment**. This does not require global deployment.
- **Performance**. This may have some overhead in connection setup and data communication because of relaying by the TURN server.

3.4.6 SOCKS

- **General description**. In order to make a connection to a server application inside a network perimeter, a SOCKS-enabled client contacts the SOCKS proxy of the server network and asks for a connection to the server. Then, the proxy forwards the connection to the server application and relays data between them. The outside client can also accept connections, but only from the same inside server. This makes SOCKS work only for client-server model applications. In order to accept a

connection, the client creates a proxy socket at the SOCKS proxy of the server network. The server then connects to the proxy socket, which is forwarded to the client by the SOCKS proxy. If an application makes a connection to a server and then must accept connections from other endpoints, it cannot use SOCKS.

- **Connectivity**
  - **Applications coverage**. Only client-server model applications can be supported.
  - **Middlebox policy coverage**. Fixed white-list and fixed black-list.
  - **Middlebox coverage**. Any middlebox can be supported. However, private networks may not be supported.
- **Security**
  - **False positive traffic**. If a SOCKS proxy is attached to an opening (i.e. it is installed right next a middlebox) and uses a strong security mechanism to authenticate traffic, only authorized applications can communicate through the opening to which the proxy is attached.
  - **Granularity of traffic control**. Fine-grained control is possible. For example, different versions or implementations can be given different certificates and the SOCKS proxy can distinguish endpoints based upon certificates they provide.
  - **Perimeter softening**. This mechanism adds new components to the network perimeter and may soften the perimeter. However, unlike application proxy (3.4.2), the SOCKS proxy does not include any application logic in it and performs very simple operations.
  - **Endpoint security**. This approach does not require any change to sensitive components of end hosts such as the operating system. Applications must be SOCKS-enabled, which generally requires linking with the SOCKS library. The authentication by the proxy may improve end-to-end security at some level because a part of the end-to-end channel is secured.
- **Deployability**
  - **Single deployment**. To deploy SOCKS to a network *X*, the SOCKS proxy must be installed on *X*'s perimeter, which requires network administrator intervention

in general. The client applications authorized to make inbound connection to *X* must become SOCKS-enabled. However, server applications in *X* need not be changed.

- **Multiple deployment**. In order to deploy SOCKS to two networks *X* and *Y*, the client applications authorized to make inbound connections into *X* or *Y* must become SOCKS-enabled. The SOCKS proxy must be installed on *X*'s and *Y*'s perimeters.

- **Performance**. This adds a fair amount of overhead to connection setup and data communication because of relaying and security checks by the SOCKS proxy.

4. Connection Arrangement Scheme

The success of Grid, internet telephony, and peer-to-peer (P2P) software has brought special attention to the connectivity and security problem addressed in this dissertation. Communications in these areas are generally characterized as inter-organizational, many-to-many, bidirectional, and dynamic. Middleboxes are deployed on organizational boundaries. Therefore, the connectivity problem arises in these areas much more often than in other areas due to their inter-organizational communications. However, it is very difficult to open middleboxes securely due to their complex communications. In addition, virtual machine technologies [95]-[97] increase dynamism of communication and the chance that peer machines are in different network perimeters—virtual machines in a host are often configured to form a private network headed by the host machine. Furthermore, some areas such as Grid started using virtual machine technologies, increasing its complexity even further. Most traversal systems were developed or are under development in this context—the IETF MIDCOM [67] group deals with middlebox traversal mainly in the context of SIP (Session Initiation Protocol), an internet telephony protocol; GGF (Global Grid Forum) [100] recently formed a working group to deal with middlebox traversal in the context of Grid.

Organizations use different types of middleboxes, have different middlebox policies, constraints and criteria. Therefore, a middlebox traversal technique that assumes a specific middlebox type, policy, etc. may work well for certain organizations but may not for other organizations. We strongly believe that no single traversal mechanism can support all organizations given this diversity. The challenge is how to support inter-organizational communications (of Grid, internet telephony, and P2P applications) when organizations cannot use the same traversal mechanism due to their differences in policies, constraints, network configuration, etc.

To handle this challenge, we develop multiple traversal techniques under the same

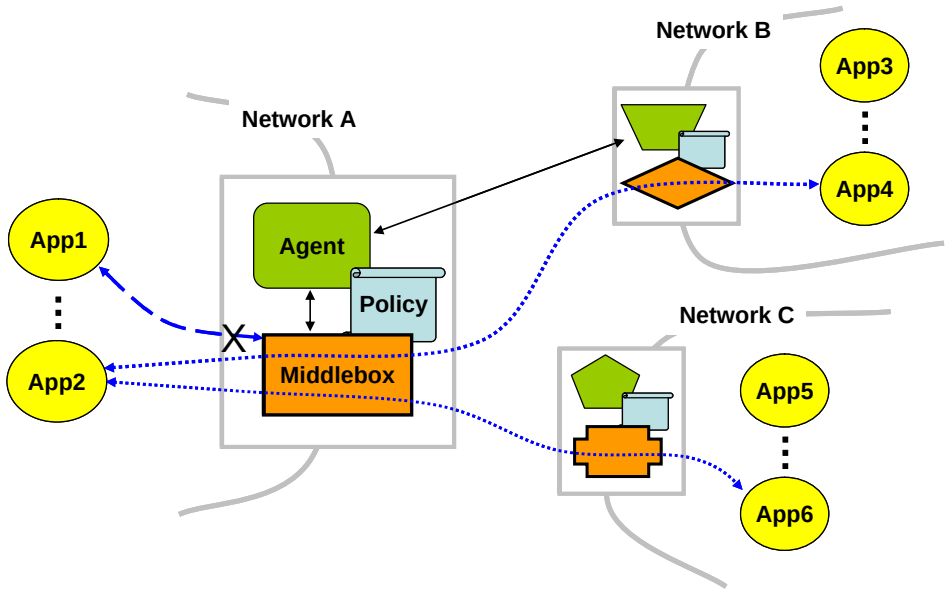


Figure-4.1: Topology. *Each network may have one or more agents that control traffic into and out of it. In controlling traffic, each agent uses a traversal technique chosen by the network administrator.*

connection arrangement scheme. Under this scheme, traversal techniques can be coordinated so that organizations using different techniques can communicate with each other. In this chapter, we present the connection arrangement scheme, which can be regarded as (1) a common part of our traversal techniques presented in Chapter 5 or (2) a meta-traversal mechanism that contains our traversal techniques as components.

4.1 Architecture

Figure-4.1 shows a typical topology. Each network may have one or more agents, a middlebox and possibly other perimeter defense components (not shown in the figure for brevity), and a policy that defines authorized applications allowed to communicate through the middlebox of the network. Each agent can be configured to support a middlebox traversal technique. Networks may use different middleboxes and different traversal techniques to control traffic passing through their middleboxes. To emphasize this, in the figure we use different shapes to denote agents and middleboxes for each network. The location of an agent also varies depending on the traversal mechanism being used.

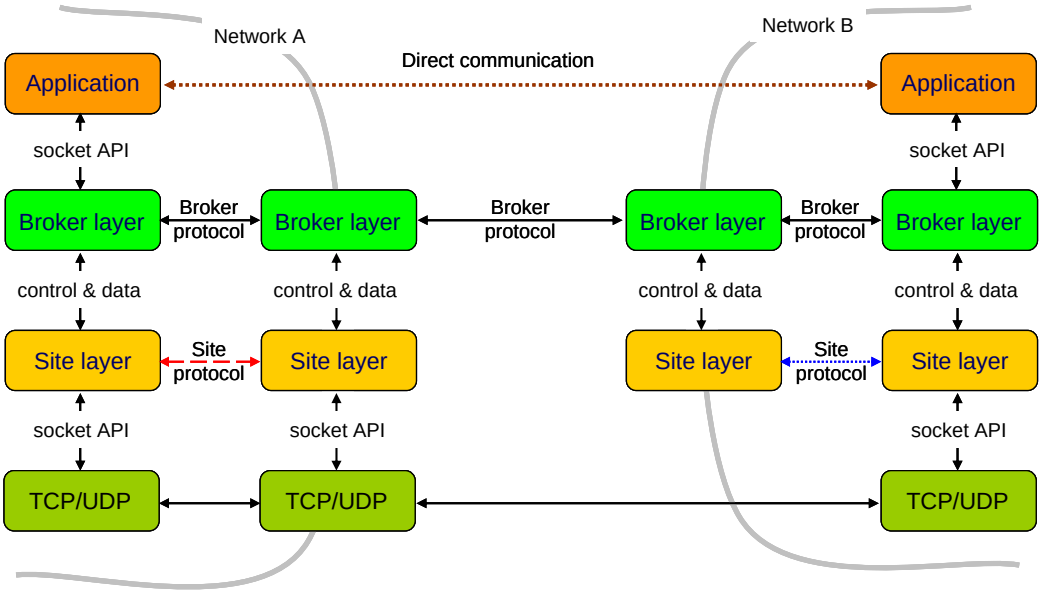


Figure-4.2: Layered architecture. *The broker layer provides applications the abstraction of direct communication by coordinating difference between site protocols.*

The basic idea of our scheme can be expressed as "you open your middlebox in your way, I will open mine my way". Upon receiving a connection arrangement request from a client application[3], the agent of the client network talks with the agent of the destination network on behalf of the application. Then, two agents cooperate and coordinate their traversal mechanisms. The end-to-end communication channel is provided by each network opening its middlebox in its own way—i.e. using its own traversal mechanism. Figure-4.1 shows that an authorized application ('app-2') can communicate with multiple peers ('app-4' and 'app-6') in different networks, which use traversal mechanisms different from each other and from that of its network.

Figure-4.2 shows a layered structure that explains how the connection arrangement scheme (denoted as "broker layer") interacts with the application and the traversal technique ("site layer"). Applications interact with the broker layer through the socket API. The broker layer provides the same syntax and almost same semantics as Berkeley

[3] Our scheme is not restricted to client-server model applications. Whether an application is a 'client' or 'server' is defined per connection. Therefore, an application may become a client for a connection and become a server for another connection later or at the same time.

sockets. Only a few functions (e.g. `bind` and `listen`) have modified semantics. Thus, the broker layer provides applications the same abstraction as the Berkeley socket system—i.e. the abstraction of direct communications. To create end-to-end channels for authorized applications, the broker layer opens intervening middleboxes by using services that the lower layer provides. The site layer actually opens middleboxes upon receiving requests from the broker layer. There is only one protocol at the broker layer. However, there could be multiple protocols at the site layer. As initial members of this layer, we developed three protocols—GCB, XRAY, and CODO. Each protocol at this layer may open middleboxes differently. Depending on site layer protocols in play, an end-to-end channel may be a single TCP/UDP connection or multiple connections concatenated together. A site layer protocol may create intermediate sockets and concatenate two connections (as the case for 'Network A' in Figure-4.2).

4.2 Connection Overview

A communication path between two endpoints is arranged on demand as part of handling socket calls by applications. This section briefly explains how an end-to-end communication channel between two endpoints is arranged by the connection arrangement scheme. Details that depend on the traversal mechanisms being used by each network are ignored here and will be explained in Chapter 5.

4.2.1 Server registration and address leasing

When an application creates a *passive* socket—i.e. a TCP socket for accepting connections or a UDP socket for receiving messages, the information about the socket is registered to the local agent—i.e. the agent of its network. The agent checks if the application is authorized to accept inbound connections[4] and, if authorized, records the socket information. If the socket is bound to a private address, the agent leases a public address to it. This section explains this registration and address leasing process in detail.

In order to be able to accept connections from outside, passive sockets behind a middlebox must be *locally bound*, *registered* to the local agent, and *officially bound*. *Local binding* is just the regular process of binding a socket to an address. Through the

[4] UDP connections will be defined in 4.2.4

local binding, an (IP, port) pair, called the *local address*, is assigned to the socket.

To arrange inbound connections, the local agent must have the information about passive sockets in the network. The *registration* process provides such information to the agent. After an application creates a passive socket bound to a local address, the broker layer at the application side sends the agent a registration request with the local address and other information such as the socket type. After authentication/authorization and the official bind (explained shortly), the agent records the information that the application has passed and other information that it collects from the official binding process.

Official binding is the process of assigning the public and globally unique address to a passive socket. This is necessary to support passive sockets locally bound to private addresses. When the agent receives a registration request with a private local address, it finds a public address and leases the address to the passive socket. This leased address becomes the *official address* of the socket. Of course, if the local address is public, then the local address becomes the official address without address leasing. As a successful response to the registration request, the agent replies with the official address.

Now that a socket could have two addresses: local and official addresses, while Berkeley socket API allows only one per socket, what address shall be known to the application? The answer is the official address as its name implies. When the application asks (by calling `getsockname`) for the address that a socket is bound to, the broker layer returns the official address instead of the local (real) address.

4.2.2 Private-to-private TCP connection

This section briefly explains how an end-to-end TCP communication channel between a client and a server application is arranged by the connection arrangement scheme (Figure-4.3). Communication channels for simpler settings such as private-to-public are arranged similarly with some steps omitted.

We assume that a passive socket has been registered and officially bound through the process explained in 4.2.1. We also assume that the client application knows the official address of the passive socket that the server application is listening on. For example, Condor components such as the job scheduler and machine manager advertise their

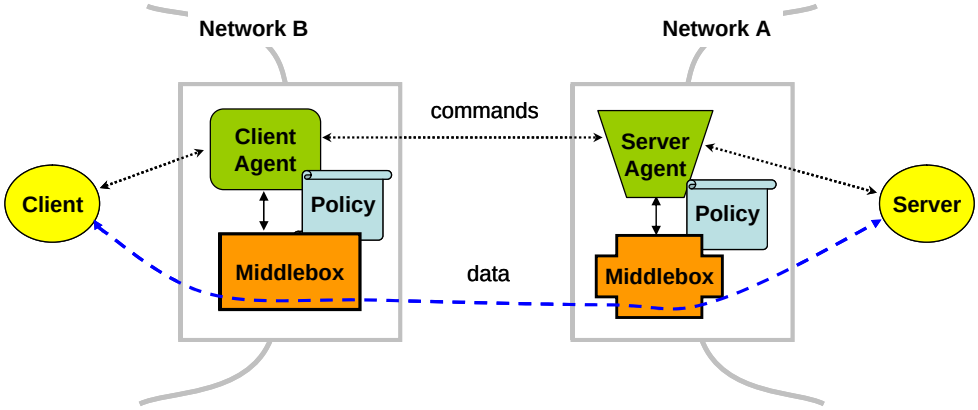


Figure-4.3: Connection overview. *Applications (client and server in the figure) interact (dotted lines) with their local agents and then agents interact with each other on behalf of applications. Communication channels between applications (dashed line) are provisioned through cooperation between agents.*

addresses to a central manager that collects and maintains information about jobs and machines. To connect to a Condor component, its address is retrieved from a central manager. In this case, the official address of a component will be stored in and retrieved from the central manager because only official addresses are known to applications.

When a client application (‘client’) wants to make a connection to a server application (‘server’)—i.e. when the client calls `connect`, its broker layer contacts the local agent (‘client agent’) and asks for the arrangement of a connection to the server. The client agent checks if the client is authorized to make outbound connections. If authorized, it asks the site layer to prepare for an outbound connection. As a response to this request, the site layer reserves, if necessary, a public address (for NAT binding or as a relay point address). Then, the client agent contacts the agent of the server (‘server agent’) and asks for an inbound connection to the server. The address reservation ensures that the connection request that the client agent sends to the server agent contains the actual source address of packets that the server agent or its middlebox will see.

Upon receiving a connection request, the server agent checks if the server is registered and the client or its agent is authorized to make inbound connections. If the test passes, it asks the site layer to open the middlebox and returns the result to the client agent. The result from the site layer may contain information such as what address or

direction the client can make the inbound connection to and if the connection should be secured.

When the client agent receives the reply from the server agent, it enables an outbound connection using its own mechanism. Finally, the client agent notifies the client of the arrangement result.

4.2.3 Intra-network connections

Connection establishment within a private network also needs to be arranged. Passive sockets in a private network are officially represented by their (leased) public addresses. Therefore, client applications in the same private network as a server application may not make direct connections to the passive socket with the leased address. As with inter-network connections, a client application must ask for a connection arrangement to the local agent. In this case, the agent replies to the request with the (private) local address of the passive socket so that the client can make a direct connection to the server.

4.2.4 UDP connections

Because there is no connection setup in UDP communications[5], we define a UDP connection as a series of UDP messages exchanged between two addresses without long interruptions. A UDP connection begins when an application (i.e. client) sends the first UDP message to another application (i.e. server). It ends when there is no communication between two applications for a predefined timeout. A new connection begins when either side sends a UDP message after the timeout. Note that the role of an application, either client or server, can be changed between connections agnostic to the application. The connection arrangement procedures explained in 4.2.2 and 4.2.3 are performed when a new UDP connection begins.

[5] There is UDP `connect` call in Berkeley sockets. However, no actual connection establishment is performed for this call. The kernel simply sets a data structure with the destination address of the `connect` call so that UDP messages can be sent to the address by default.

5. Middlebox Traversal Techniques

This chapter presents the middlebox traversal techniques that we developed. In order to develop traversal techniques which support intended organizations well, yet collectively support representative organizations, we (1) select representative points on the multi-dimensional space that the framework defines and (2) develop traversal techniques for those points.

(1) We use middlebox policy type as the main dimension in selecting points. Our first criterion is to cover all practical middlebox policy types in Section 3.2. Next, we select reasonable combinations of "false positive traffic" (from the security group) and "performance" for each middlebox policy type. Finally, we choose desirable values for remaining dimensions under the constraints of the first three dimensions and also considering trade-offs between dimensions.

(2) We develop three traversal techniques—GCB, CODO, and XRAY. The framework allows us to develop techniques that are specialized for targeting points and support intended use cases well. Because they are based upon our new definition of middlebox traversal, each technique can be used as a security tool for perimeter defense as well as an enabling tool that facilitates communications through middleboxes.

5.1 GCB

GCB (Generic Connection Brokering) targets the "outbound only" type of middlebox policy. GCB lets authorized applications communicate through existing middlebox openings (i.e. "outbound allowed") without changing the network perimeter—i.e. creating openings in the middlebox or installing new entities on the network perimeter. Therefore, a network perimeter is protected as securely as it was before GCB is deployed. GCB has small overheads in connection setup and data transfer. It can support any type

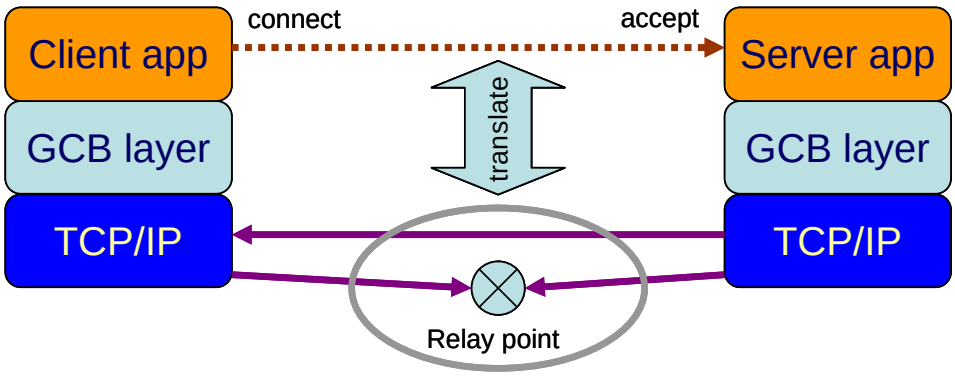


Figure-5.1: GCB basic idea. *GCB decouples the direction of a connection from the role. It provides applications transparency by hiding the fact that connections may be made in the opposite direction or through relay points. The broker layer between the application and GCB layer is omitted for brevity.*

---

of middlebox and has few restrictions on the type of applications it can support.

5.1.1 Architecture

In the traditional Berkeley socket system, the direction of a connection is *tightly coupled* with the role (i.e. who the client/server is) in the communication. A connection is always made from a *client* to a *server*. In a TCP connection setup, for example, the first SYN packet always traverses from the client, who calls `connect`, to the server, who calls `listen` and `accept`.

The main idea of GCB is *decoupling* the direction from the role in a connection. In GCB, an application becomes either a client or a server in a connection as usual. However, the direction of a connection is determined independently from the role but based on the relative network topology of communicating parties. A connection can be made either (1) from a client to a server, (2) from a server to a client when a client-to-server connection is impossible but the opposite direction is possible, or (3) from both a client and a server to a rendezvous point when neither can directly talk to the other. As a layer between application and Berkeley socket, GCB arranges the direction of connections without applications' notice (Figure-5.1).

From a middlebox traversal perspective, GCB transforms connections that would

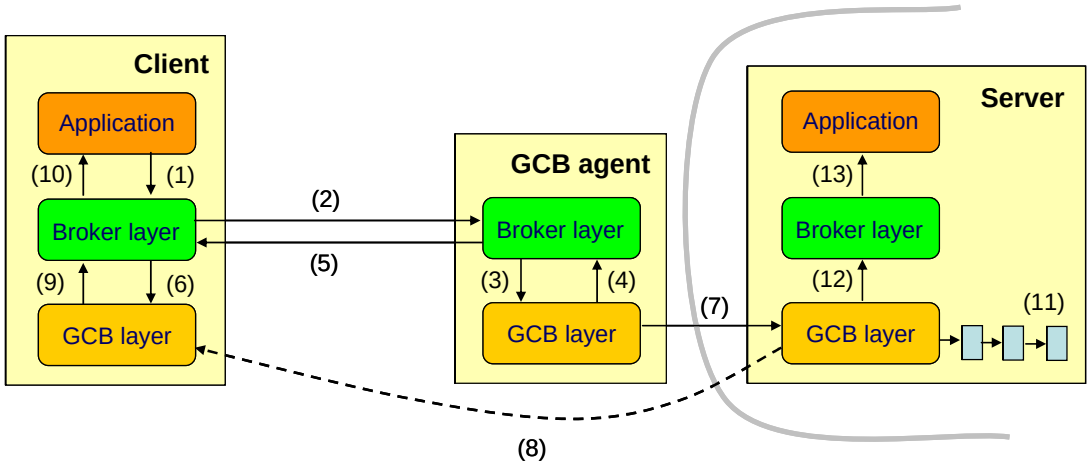


Figure-5.2: Public-to-private connection in GCB. *GCB agent arranges a connection from a client in the public network to a server in the network it manages. Actual connection is made from the server to the client.*

have been inbound and therefore blocked by the "outbound only" middlebox policy into outbound connections, which the policy allows to traverse, while hiding the fact that the direction is reversed from the application. From an Internet architectural perspective, it deals with the Internet asymmetry by decoupling the role and the direction.

5.1.2 Connection establishment

With GCB, clients inside the network perimeter communicate with outside servers as usual. To enable inside servers to communicate with outside clients, however, GCB uses the idea of a reception desk. Each network has one or more GCB agents [6] (i.e. receptionists) running outside or on its perimeter. An outside client first contacts a GCB agent (i.e. checks in) to ask for a connection to an inside server. After authenticating the client, the agent asks the insider to connect to the client. Then, a connection is made from the server to the client (i.e. the insider escorts the visitor).

Figure-5.2 shows in detail how an inbound channel is made in GCB. In Figure-5.2, we have a client in the public network and a server behind a middlebox that allows only outbound connections. A connection from the client to the server is made as follows:

[6] The GCB agent is an agent in Figure-4.1 that is configured to use GCB as the site layer protocol.

Step 0: The passive socket created by the server has been registered to the GCB agent through the process we described in Section 4.2.1. As a result of that process, a secure connection has been established from the server to the agent. The agent can send GCB commands to the server through the connection.

Step 1-3: When the client calls `connect` with the official address of the passive socket, a connection request is passed to the GCB layer of the agent. To locate the agent, the broker layer of the client must refer to a service that returns the address of the server's agent given the official address of a server.

Step 4-6: The GCB layer of the agent replies that the connection must be made from the server to the client. When the GCB layer at the client is notified of this, it makes a passive socket and waits for a connection.

Step 7: The GCB layer at the agent notifies the GCB layer of the server to make a connection to the client.

Step 8: A connection is made from the server to the client.

Step 9-10: The client's `connect` call returns. If the active socket is non-blocking, the broker layer marks the socket descriptor appropriately so that `select` call later may return the descriptor write/read ready.

Step 11: When the connection is established in Step 8, the GCB layer of the server puts the connection into a connection queue. The GCB layer of the server also accepts connections, e.g. direct connections from nodes in the same network, and puts them into the queue as well. By having a connection queue at the application layer, GCB can pass connections to the application in the order that they are established independently to directions they are made.

Step 12-13: When the application calls `accept`, the connection at the head of the queue is returned.

In case that a client cannot accept connections maybe because it is in another network that only allows outbound connections (i.e. private-to-private connections), the GCB agent of the server network creates a relay point (i.e. two sockets: one at the client side and the other at the server side) and notifies both the client and server to make

connections to the relay point. The GCB mechanism for private-to-private connections is very similar to the XRAY mechanism, which will be discussed in 5.2. However, we should note that GCB targets the “outbound only” type of middlebox policies and therefore every connection in GCB is outbound even when it uses the relay mechanism.

UDP connections are opened similarly as part of a UDP `connect` or the first `sendto` call. In Step 8, the GCB layer of the server sends a special message that the GCB layer of the client consumes instead of passing it to the client application. This has the same effect as when a TCP SYN packet is sent from the server to the client. Both sides can send UDP messages to each other after a UDP message passes through the middlebox. If the GCB layer of the client does not receive the special message within a timeout after Step 6, it retriggers the connection process until it gives up after predefined number of failures.

5.1.3 Analysis

This section analyzes the GCB mechanism under the framework presented in Chapter 3. The performance of GCB is discussed in Chapter 7.

- **Connectivity**
  - **Applications coverage**. GCB provides the same abstraction as Berkeley sockets, as much as concealing the fact that connections may be established in the opposite direction or via relay points. Therefore, almost any application that uses Berkeley sockets may be supported. However, new errors that cannot occur in regular connections may occur because connections are reversed or relayed. Also, a few functions such as `bind`, `listen`, UDP `connect`, and first `sendto` of a UDP connection become blocking calls. Such semantic changes may affect a few applications that require absolutely the same semantics as Berkeley sockets.
  - **Middlebox policy coverage**. GCB targets “outbound only” type.
  - **Middlebox coverage**. GCB does not assume particular middlebox type or behavior. Any type of middlebox can be supported.
- **Security**

- **False positive traffic**. Since GCB commands are exchanged through control channels which are authenticated using X.509 certificates and are secured with session keys established during the authentication process, only authorized applications can have the GCB agent arrange connections for them.
- **Granularity of traffic control**. Very fine-grained control is possible. For example, different versions or implementations can be given different certificates and the GCB agent can distinguish endpoints based upon the certificates they provide.
- **Perimeter softening**. It does not soften the network perimeter because it makes no change to the network perimeter.
- **Endpoint security**. GCB does not require changes to sensitive components of end hosts such as the operating system. Applications must be modified to become GCB-enabled. The authentication of applications by the GCB agent may improve end-to-end security at some level. GCB ensures that there are two authenticated endpoints (i.e. address quadruple) between which a connection can be made. When relaying connections, GCB authenticates each connection using X.509 certificates and encrypts data using the secret keys established during the authentication. This also helps end-to-end security.

▪ **Deployability**

- **Single deployment**. The GCB agent can be deployed outside the network perimeter, which does not require network administrator intervention in general. The agent does not require special privilege either. Both client and server applications must become GCB-enabled.
- **Multiple deployment**. Since both the client and server must be GCB-enabled, GCB may not be gradually deployed.

5.2 XRAY

XRAY (middleboX traversal by RelAYing) targets the “fixed white-list” and “fixed black-list” types of middlebox policies. XRAY is a proxy-based mechanism: a proxy (i.e.

XRAY agent[7]), which is deployed as one of "fixed number of authorized points" or one that is not in "fixed number of unauthorized points", relays packets to/from next hop proxies or authorized applications. Because the XRAY agent checks every packet it relays by a strong security key, it knows whether a packet is intended for an authorized application or not. Therefore, only authorized applications can pass through the middlebox and the agent. Among three traversal techniques we developed, XRAY has the largest overhead in most cases.

5.2.1 Architecture

XRAY builds an on-demand overlay channel between a client and a server socket, which may consist of multiple connections concatenated by relay points (Figure-5.3). A relay point is a pair of sockets of the same type as the endpoint sockets and is added between two points that cannot directly communicate (a connectivity perspective), or to a place where traffic control must occur (a perimeter defense perspective). Each connection to a relay point is mutually authenticated using X.509 certificates, and data over the connection are encrypted and integrity checked using the secret keys established during the authentication. Therefore, only the intended endpoint or relay point can communicate via a relay point. Like GCB, XRAY provides applications the abstraction of direct communication, as much as concealing the fact that an end-to-end communication channel may contain many intermediate relay points. XRAY has several benefits:

- **Strong security tool**. Because only authorized applications can communicate via relay points, XRAY can be a strong security tool for perimeter defense. If an organization deploys XRAY so that relay points are created next to its middleboxes—i.e. relay points are conceptually attached to one end of an opening of a middlebox so that there is no exit, entry, or alteration of packets in the path between the end of the opening and the relay points—only authorized applications can communicate through middleboxes and relay points.
- **Symmetry.** Most organizations used to use the "outbound only" type of middlebox policies. However, more and more organizations want to control communications in both directions for reasons such as security and legal issues. XRAY can be used to

[7] The XRAY agent is an agent in Figure-4.1 that is configured to use XRAY as the site layer protocol.

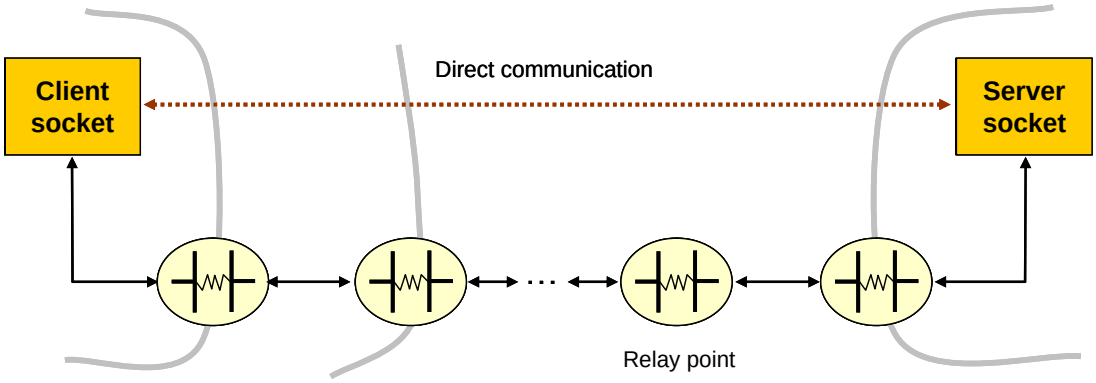


Figure-5.3: XRAY basic idea. *XRAY creates on-demand end-to-end channels by placing relay points when direct communication between two points is impossible or undesirable.*

control both inbound and outbound traffic.

- **Application independence.** XRAY mechanism is absolutely generic and does not have any application dependency. This application independence allows one XRAY agent to support many applications, avoiding the "too many authorized points" problem of the "fixed white-list" policy (Section 3.2.4).
- **Conflict [70] between applications and middleboxes resolved**. Many applications encrypt contents with strong security mechanisms to protect their payloads from observation or modification. On the other hand, network perimeters need to inspect payloads for better filtering, intrusion detection, etc. When network perimeters cannot look inside packets, they generally drop packets. XRAY provides a reasonable resolution for this conflict.

  Relay points in XRAY terminate security associations as well as transport connections. This hop-by-hop security provides each relay agent full access to payloads so that packet inspection can be done here. On the other hand, applications can protect their contents from being observed or modified by other than relaying agents. Clearly, this approach is not ideal for applications because network perimeters have full access to payload, meaning applications lose some end-to-end security. However, perimeters have the power of arbitration and may completely block applications when they cannot get information from packets.

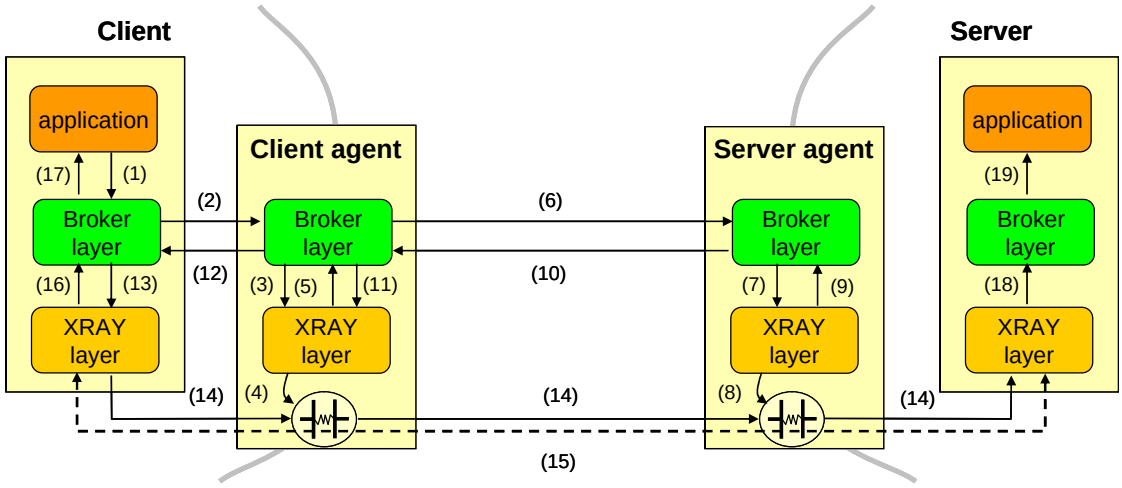


Figure-5.4: Private-to-private connection in XRAY. *Both client and server agents create relay points to allow a connection from a client to a server in different networks.*

5.2.2 Connection establishment

This section explains how a client application in a network makes a connection to a server in another network (Figure-5.4). Each network has a middlebox (not shown in the figure) and an XRAY agent on its perimeter. A connection from the client to the server is made as follows:

Step 0: The passive socket created by the server has been registered to the local XRAY agent (‘server agent’) through the process we described in Section 4.2.1. As a result of that process, a secure connection has been established from the server to the agent. The agent can send commands to the server through the connection.

Step 1-3: When the client calls `connect`, a connection request is passed to the XRAY layer of the local agent (‘client agent’). To locate the client agent, the broker layer of the client must refer to a service that returns the agent address for the official address of a server.

Step 4-5: The client agent checks if the client is authorized to make outbound connections. If allowed, it creates a relay point and returns the result to the broker layer. One half (i.e. a socket) of the relay point is assigned with the

client's name.

Step 6-7: The broker layer of the client agent contacts, on behalf of the client, the server agent and asks for a connection to the server. The request is passed to the XRAY layer of the server agent. To locate the server agent, the client agent must refer to a service that returns the address of the server agent for the official address of a server.

Step 8-10: The XRAY layer of the server agent creates a relay point and returns the result to the broker layer. The result is forwarded to the broker layer of the client agent. Sockets of the relay point are assigned with the client agent's name and the server's name, respectively.

Step 11: The broker layer of the client agent notifies the XRAY layer of the fact that the server agent has created a relay point. The XRAY layer assigns the other half of the relay point with the server agent's name.

Step 12-13: The client agent notifies the client that it can connect to the socket of the relay point at the client agent, which has been assigned with the client name (Step 4).

Step 14: Three overlay links—i.e. client application-to-client agent, client agent-to-server agent, and server agent-to-server application—are established in parallel. All links are authenticated through security handshakes. Also, secret keys are established for encryption/decryption and integrity check.

Step 15: XRAY layers of the client and server send acknowledgments to each other through the channel established in Step 14. Upon receiving an acknowledgment, each party knows the end-to-end channel was successfully established.

Step 16-17: The client's `connect` call returns. If the active socket is non-blocking, the broker layer marks the socket descriptor appropriately so that `select` call later may return the descriptor write/read ready.

Step 18-19: When the application calls `accept`, the connection at the head of the queue is returned.

The end-to-end acknowledgement in Step 15 may seem unnecessary because the application can send data as soon as the overlay link between itself and the next hop is

established. If one or more intermediate links have not been established yet, the data could be buffered at the relay points and pushed later. We did not take this approach because it would change error semantics. Connection failures often involve human errors such as mistyping the address or a bad network configuration, while data transfer failures after the connection establishment are mostly caused by network errors and happen much less frequently. Therefore, most applications are prepared to handle connection errors but not well prepared to handle transfer errors. If XRAY reports a successful connection but some intermediate links are not finished yet, applications will see more data transfer errors that are not the result of network errors. Error semantics of UDP connection setup do not require any level of guarantee from the network. Therefore, the end-to-end acknowledgement is performed only for TCP connections.

5.2.3 Analysis

This section analyzes the XRAY mechanism under the framework presented in Chapter 3. The performance is discussed in Chapter 7.

- **Connectivity**
  - **Applications coverage**. As with GCB, XRAY provides the same abstraction as Berkeley sockets and therefore can support almost any application that uses Berkeley sockets. A few functions such as `bind`, `listen`, UDP `connect`, and the first `sendto` of a UDP connection become blocking calls. These changes in semantics may affect a few applications that require absolutely the same semantics as Berkeley sockets.
  - **Middlebox policy coverage**. XRAY targets the “fixed white-list” and “fixed black-list” types.
  - **Middlebox coverage**. XRAY does not assume particular middlebox type or behavior. Any type of middlebox can be supported.
- **Security**
  - **False positive traffic**. If an XRAY agent is attached to an opening of a middlebox, only authorized applications can communicate through the opening

and the relay points that the agent creates. Therefore, no false positive traffic is possible.

- **Granularity of traffic control**. As with GCB, XRAY can enforce various policies and can do very fine-grained control of traffic.
- **Perimeter softening**. XRAY adds new components to the network perimeter and may soften the perimeter. However, unlike application specific proxies, the XRAY agent does not include any application logic in it and performs very simple operations.
- **Endpoint security**. XRAY does not require changes to sensitive components of end hosts such as the operating system. Applications must be modified to become XRAY-enabled. Since each overlay link is secured by a strong security mechanism, XRAY can be seen as improving end-to-end security.

▪ **Deployability**

- **Single deployment**. To deploy XRAY to a site, the XRAY agent must be installed (generally on the network perimeter). This usually requires network administrator's intervention. However, the agent does not require special privileges to run. Both client and server applications must become XRAY-enabled.
- **Multiple deployment**. In order to communicate through middleboxes, both the client and server must be XRAY-enabled. Therefore, XRAY may not be gradually deployed in certain circumstances.

5.3 CODO

CODO (Cooperative On-Demand Opening) targets the "dynamic white-list" type of policy. It dynamically adds and deletes middlebox rules on demand from authorized applications. Since CODO command channels are secured with strong security mechanisms, only authorized application can have CODO create openings for them. Therefore, unauthorized applications can communicate through openings made for authorized ones only by connection hijacking. CODO has small overheads in both connection setup and data communications.

5.3.1 Architecture

CODO builds on-demand communication channels for authorized application by dynamically opening middleboxes in the communication path. Through secure control connections, CODO-enabled applications ask the CODO agent[8], add-on software for middleboxes, to build communication channels for them. Then, the CODO agents of intervening middleboxes collaborate to build requested channels.

In CODO, middleboxes and applications cooperate for better connectivity and perimeter defense. Authorized applications report to the CODO agent (i.e. to the middlebox) their address usage such as binding a socket to an address and closing the socket. This enables the middlebox to bind addresses (i.e. IP and port pairs) to authorized applications using them at each moment. Using this binding capability, the middlebox can efficiently pass packets for authorized applications while blocking those for unauthorized ones: packets with addresses being used by an authorized application are forwarded while others are dropped. Since no two applications can use a unicast address at the same time, unauthorized applications can traverse middleboxes only through connection hijacking. Furthermore, the CODO agent informs the application of middlebox parameters such as flush rate of inactive connections[9]. The CODO layer at the application side can use such information to help the application communicate through the middlebox with fewer problems. For example, when the application keeps a connection open without sending any data for near the flush rate of an intervening middlebox, the CODO layer sends a heartbeat data to prevent the middlebox from closing the opening.

CODO opens middleboxes as narrow and short as possible (1) by opening middleboxes with fully qualified address quadruples whenever possible and (2) through leveraging stateful middlebox technology:

(1) Under the framework of the connection arrangement scheme presented in Chapter 4, CODO can add middlebox rules with minimum wildcard addresses. In the scheme, agents wait until both source and destination addresses are fully determined before

[8] The CODO agent is an agent in Figure-4.1 that is configured to use CODO as the site layer protocol.
[9] Most middleboxes close the opening made for a connection after a certain period of inactivity.

opening their middleboxes. Therefore, CODO agents can add middlebox rules with fully qualified (protocol, source IP, source port, destination IP, and destination port). This means that middleboxes are opened as narrowly as possible and only when there is an authorized pair of applications. Cases in which CODO has to open middleboxes with wildcards will be discussed in 5.3.2.

(2) After the first packet of a connection is allowed to pass through a stateful middlebox, all subsequent packets of the connection are allowed to traverse it. Stateful middleboxes filter initial packets through slow packet classification, which requires referring to middlebox rules, but filter subsequent packets using state entries (see Section 2.1 for detail). The stateful middlebox keeps track of a connection and deletes the corresponding state entry immediately after the connection is closed. Therefore, a middlebox rule that permits a connection becomes unnecessary after a state entry is created for the connection. The middlebox rule is necessary only for next connections. Since CODO opens middleboxes per connection, the rule should be deleted as soon as it becomes unnecessary. In order to prevent attackers from exploiting the middlebox rules created for authorized applications, CODO limits the duration of middlebox rules as much as possible by deleting the rules it adds as soon as the stateful middlebox creates the necessary states.

In addition to those explained above, CODO has several benefits. Because CODO uses the out-of-band method to learn an application's need for channels (i.e. openings at middleboxes), The CODO agents need not have application intelligence nor investigate packets. This makes CODO efficient and generic, avoiding the "new intelligence per new application" problem of the "dynamic white-list" type of middlebox policy. It also can control both inbound and outbound connections similar to XRAY.

5.3.2 Connection establishment

This section explains the connection mechanism of CODO for a TCP connection from a client in one network to a server in another network (Figure-5.5).

Step 0: The passive socket created by the server has been registered to the local CODO agent ('server agent') through the process we described in Section 4.2.1. As a result of that process, a secure connection has been established

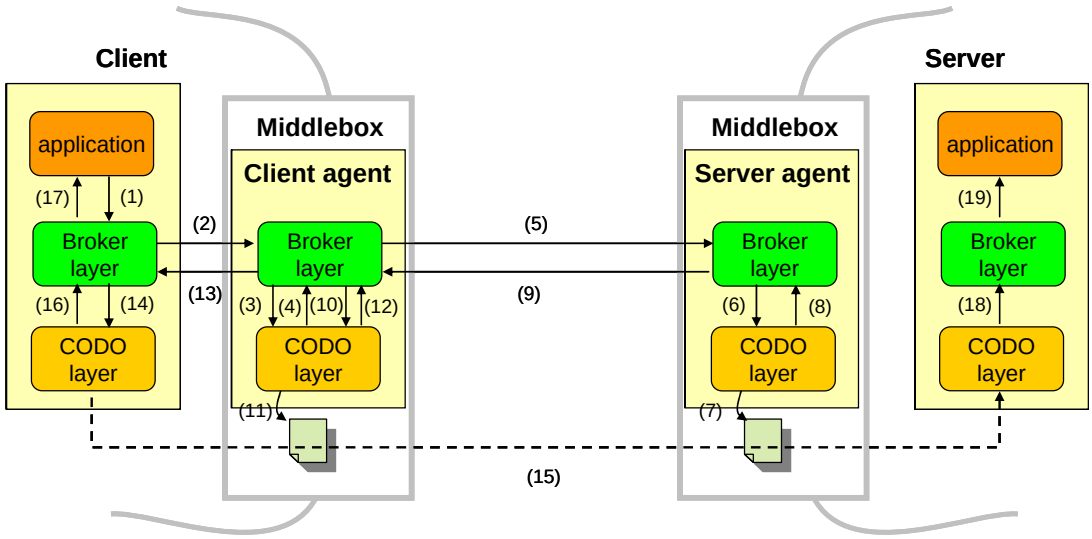


Figure-5.5: Private-to-private connection in CODO. *Both client and server agents add rules to their middleboxes to allow a connection from a client to a server in different networks.*

from the server to the agent. The agent can send commands to the server through the connection.

Step 1-3: When the client calls `connect`, a connection request is passed to the CODO layer of the local CODO agent (‘client agent’).

Step 4: The client agent checks if the client is authorized to make outbound connections. Even if the client is allowed to make outbound connections, the agent does not immediately open the middlebox. Instead, it records the connection request. If the middlebox is a NAT, the agent reserves a NAT binding and returns the public address of the binding to the broker layer.

Step 5-6: The broker layer of the client agent contacts, on behalf of the client, the server agent and asks for a connection to the server. If a NAT binding was reserved in Step 4, the source address field in the request will be the public address of the binding. The request is passed to the CODO layer of the server agent.

Step 7-9: Note that here the server agent has a fully qualified address quadruple for the connection. The CODO layer of the server agent adds a middlebox rule

that allows inbound connections from the client to the server and returns the result to the broker layer. The result is forwarded to the broker layer of the client agent. If a client is in a private network that does not use a CODO agent, it has to contact the server agent itself. In this case, a NAT binding cannot be reserved (Step 4) and neither the client nor the server agent knows what the source address of the connection will be. To handle such cases, the server agent adds a middlebox rule with the wildcard source address when the source address field of a connection request is different from the address that the request comes from.

Step 10-14: The broker layer of the client agent notifies the CODO layer of the fact that the server agent has opened its middlebox for inbound connections. The CODO layer adds a rule that allows outbound connections from the client to the server. The result is passed to the CODO layer of the client.

Step 15: Initial packets (i.e. TCP SYN and SYN-ACK) are exchanged between the client and server. At this point, state entries are created in both middleboxes and then both CODO agents delete rules that they have added.

Step 16-17: The client's `connect` call returns. If the active socket is non-blocking, the broker layer marks the socket descriptor appropriately so that a `select` call later may return the descriptor write/read ready.

Step 18-19: When the application calls `accept`, the connection at the head of the queue is returned.

5.3.3 Analysis

- **Connectivity**
  - **Applications coverage**. As with GCB and XRAY, CODO provides the same abstraction as Berkeley sockets and therefore can support almost any application that uses Berkeley sockets. A few functions such as `bind`, `listen`, UDP `connect`, and the first `sendto` of a UDP connection become blocking calls. Such semantic changes may affect a few applications that require absolutely the same semantics as Berkeley sockets.
  - **Middlebox policy coverage**. CODO targets the "dynamic white-list" type.

- **Middlebox coverage**. CODO can support only middleboxes that can be programmatically controlled.

- **Security**
  - **False positive traffic**. Through address spoofing, unauthorized applications can communicate through openings that CODO makes for authorized applications. However, the window is small because CODO opens middleboxes as briefly and narrowly as possible.
  - **Granularity of traffic control**. As with GCB and XRAY, CODO can enforce various policies and can do very fine-grained control of traffic.
  - **Perimeter softening**. CODO may soften a network perimeter because it adds software to middleboxes, one of the most sensitive components of the network perimeter. However, unlike ALG (see Section 3.4.3), the CODO agent must be deployed only once because it does not include any application logic in it.
  - **Endpoint security**. CODO does not require changes to sensitive components of end hosts such as the operating system. Applications must be changed, e.g. re-linked, to become CODO-enabled. The authentication of applications by the CODO agent may improve end-to-end security at some level. CODO ensures that there are two authenticated endpoints (i.e. address quadruples) between which a connection can be made.

- **Deployability**
  - **Single deployment**. To deploy CODO at a site, the CODO agent must be installed on the middlebox machine, which generally requires network administrator intervention. Both client and server applications must become CODO-enabled.
  - **Multiple deployment**. In order to communicate through middleboxes, both the client and server must be CODO-enabled. Therefore, CODO may not be gradually deployed in certain circumstances.

6. Implementation

We have implemented our traversal techniques as a *client* and a *server*. To avoid confusion, we will use *italic* to denote our implementations. The *client* corresponds with the application side and the *server* corresponds with the agent. The *client* is a library that applications can link with in order to communicate through middleboxes using our techniques. The *server* is implemented as a daemon process that can be installed on or near network perimeters.

6.1 Client Implementation

The *client* is implemented as a layer between the application and the kernel (Figure-6.1). It provides applications the same abstraction as Berkeley sockets for socket calls and the same abstraction as Unix for some file system and process control calls.

Using socket calls that the *client* provides, applications can create a socket, bind it to an address, connect to a server, accept a connection from a client, and so forth. The *client* provides some file system calls so that applications may duplicate socket descriptors, make a socket non-blocking, and multiplex multiple file descriptors. It also has a few functions for process control, such as `fork` and `execve`. These are mainly for inheriting information of open sockets to child processes. All calls have the same APIs as their regular counterparts. Therefore, applications may use the *client* very easily. With help from interposition mechanisms [28] [29], applications can use the *client* even without re-linking. Using link options available for some compilers [98], an application's regular calls can also be converted to our calls.

As shown in Figure-6.1, the overall structure of the *client* is similar to BSD's implementation of TCP/IP [85]. Each socket has a socket structure ('SockInfo'), which stores status for and other information about the socket. Each socket structure is pointed to by one or more entries of the file descriptor table ('fd table'). A socket structure is

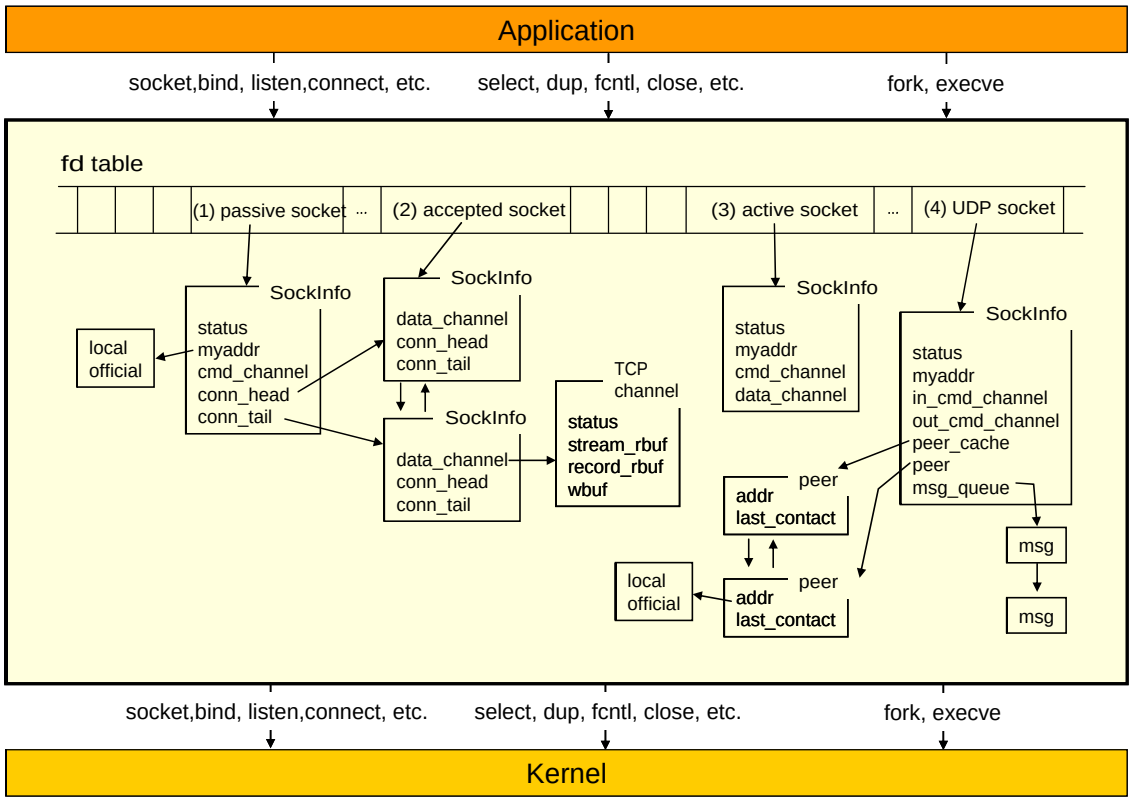


Figure-6.1: Client structure. *Information ("SockInfo" in the figure) of each socket is linked from the file descriptor table ("fd table"). Auxiliary information of a socket is linked from its "SockInfo" structure.*

created when the application creates a socket. A new socket structure is also created when the *client* creates a socket for internal use. For instance, when a connection is accepted, the *client* creates a socket structure for the accepted socket. A socket structure is pointed to by multiple 'fd table' entries when the application duplicates a socket descriptor. When the application closes a socket, the corresponding entry in the 'fd table' is disconnected from the socket structure. A socket structure is deleted when no entry refers to it. In the following sections, we explain important data structures of the *client* implementation and major operations within it.

6.1.1 TCP socket: (1), (2), and (3)

Within the *client*, each socket is implemented as a state-transition machine. Upon an application's socket call or when an event such as receiving a message occurs, a socket

moves from its current state to another. The ‘status’ contains the current state of a socket. (We will talk a little more about the state machine in Section 6.1.3.) A socket may have two addresses: a local and official address (Section 4.2). The ‘myaddr’ field contains those addresses. A passive socket may have a secure TCP channel (‘cmd_channel’) that is connected to the local agent. Through this channel, the client registers the socket, borrows the official address, and receives commands from the agent. The `bind` call does not return until both local and official addresses are determined.

The ‘conn_head’ and ‘conn_tail’ point the head and the tail, respectively, of the queue of connections accepted on a passive socket. When a connection is accepted or an active connection needs to be made (in the GCB case), the *client* creates a socket structure (i.e. ‘SockInfo’). A connection is added to the connection queue when it is ready for data communication—e.g. when an active GCB connection is established or when a security handshake over an accepted connection is finished. When the connection queue is not empty, the `accept` call returns a connection from the head. Similarly, a `select` call returns the socket descriptor as read-ready when the queue is not empty.

As a response to a connection arrangement request, an agent may require that a secure connection be used for an overlay link (Section 4.2). Also, the *client* can be configured to use secure or plain communication with peers at certain IP. The ‘data_channel’ points to the object (‘TCP channel’) that represents a TCP overlay link between the application and the next hop agent or the peer application. Data are sent and received via the overlay link, which can be either a secure or plain connection. The ‘TCP channel’ object hides the fact that the overlay link it represents may be secured and uniformly provides the ordinary TCP semantics for data communication. For example, it has a buffering mechanism that provides applications the stream semantics of TCP over the record-based communication of secure connections that use a block-cipher [65] [71]. It reads encrypted blocks (i.e. records) from the underlying TCP and queues decrypted data into a stream buffer (‘stream_rbuf’). Since the underlying TCP is stream-based, it is possible that only a partial record is received from the network. In this case, it queues the partial record in a record buffer (‘record_rbuf’) until more data arrive to make a whole record. A socket descriptor is not tagged as read-ready unless the stream buffer contains data. In other

words, a `select` call returns a socket descriptor as read-ready only when its stream buffer contains plaintext data more than its low-water-mark [84].

The fields of socket structures of active sockets are similar to those of passive ones. Active sockets need not have official addresses. Therefore, they are set to the same value as local addresses. Active sockets also have ‘data_channel’ fields.

6.1.2 **UDP sockets:** (4)

UDP sockets have a socket structure similar to TCP sockets. A UDP socket can be both active (i.e. sending initial packets to some peers) and passive (i.e. receiving initial packets from others) at the same time. For this reason, a UDP socket structure has two control channels (‘in_cmd_channel’ and ‘out_cmd_channel’). The first one is connected to the agent that arranges connections toward the socket, while the second is connected to the agent that arranges connections from the socket.

Each UDP socket has the cache of peers (‘peer_cache’). If a UDP socket is connected to a peer, its socket structure (‘peer’ field of ‘SockInfo’) points to a specific peer in the cache. Each peer structure in the cache has a (local, official) address pair linked from its ‘addr’ field. A (*local*, *official*) pair means that messages to the peer represented by the *official* address must be sent to the *local* address. When the application calls `sendto` with a destination address *X*, the peer cache is searched for an entry whose official address is equal to *X*. If no entry is found, a new connection is opened and the message is sent to the peer as explained in Chapters 4 and 5, and an entry for the peer is created and added to the cache. If an entry with (*X*, *Y*) is found, the message is simply sent to *Y*. Each time a message is sent to or received from a peer, the ‘last_contact’ field is updated with the current time. Peer entries that have been inactive for longer than a predefined timeout are deleted from the cache.

When a UDP message is received from the network, the cache is searched for the peer structure with the local address equal to the sender’s (IP, port). If no entry is found, a new peer structure is added to the cache. If the *client* is configured to authenticate messages from the peer, the message is security checked, and then the plaintext UDP message is queued to the receive buffer (‘msg_queue’). When ‘msg_queue’ is not empty,

`recvfrom` returns a message from the head of the queue. Also, `select` returns a UDP socket as read-ready in this case.

To authenticate UDP messages, the current implementation uses a public key method. Each secure UDP message is signed and verified by the sender's private and public key pair. Obviously this approach adds big overhead to data communications. We will enhance our implementation to support better mechanisms as a future work.

6.1.3 Handling asynchronous events

There are many asynchronous events that the *client* must handle. For instance, when a new connection is accepted to a TCP passive socket, the *client* must create a new 'SockInfo' for the accepted socket, perform a security handshake if necessary, and put the socket into the connection queue. Handling an event may takes a while and involve blocking operations such as TCP `connect` and `recv`. If the *client* handles an event from start to end or is blocked on a system call, it may not promptly handle events that occur during it is handling that event. Therefore, the *client* should not call a blocking function inside routines that handle events or spend so much time to handle a single event. This requires that the *client* divide an event handling into multiple parts and switch between events by performing small number of parts from each event at a time. For the same reason, the *client* should not process an application's socket call from start to end.

To handle asynchronous events efficiently, each socket is implemented as a state machine in the *client*. In a state machine, each state consists of multiple non-blocking operations possibly followed by a blocking operation. A socket moves from a state to another when an event occurs or when the application issues certain calls for the socket (Figure-6.2). When an event occurs or when the application issues a call, the *client* performs non-blocking operations in the current state of the socket and then calls the last blocking operation in non-blocking fashion. If it can finish the last operation immediately, it proceeds to the next state and keeps traversing the state machine. If not, it stops traversing the state machine and resumes later when another event occurs. A socket call by the application returns when its state machine reaches the appropriate state.

Because an application may have multiple sockets open, the *client* generally has

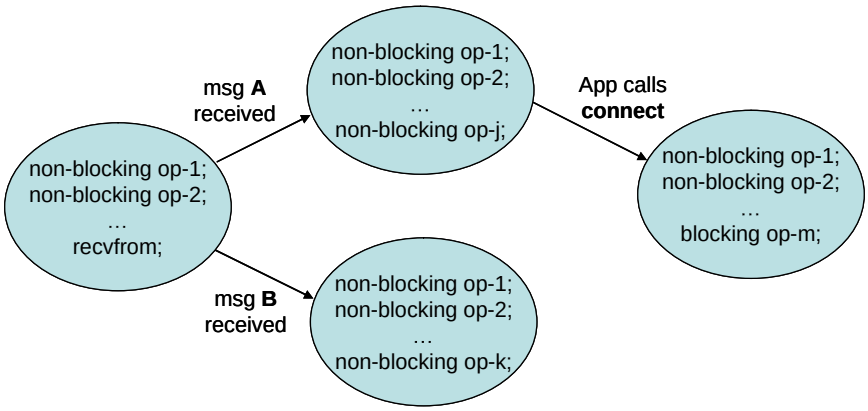


Figure-6.2: State machine. *A socket moves from a state to another when the application calls a function for the socket or an event occurs. In this example, the state change from the leftmost state depends on the type of messages received (i.e. event type). On the other hand, the change from the top to the rightmost state is triggered by the application's connect call.*

---

multiple state machines. Some machines are just waiting for the application to issue a socket call, while others waiting for event. To handle asynchronous events that these state machines are waiting for, the *client* dynamically collects events of interest into a set ('interest event set'), whose members are determined by the current state of each state machine (Figure-6.3). Some members of the set are directly passed from the application when the application is also waiting for events by calling `select`. Notice that some events that the application is interested in are not included in the set ('X' mark in Figure-6.3). The *client* translates them into other events and adds the translated events into the set. For example, when the application issues a non-blocking `connect` and waits for the socket descriptor to become write-ready, the *client* should wait for the socket to become read-ready if the connection is being made in the reverse direction (because the GCB mechanism is being used for the connection). After collecting events of interest, the *client* waits for all of those events. When some of those events occur, it handles them by executing corresponding state machines or passes the events to the application.

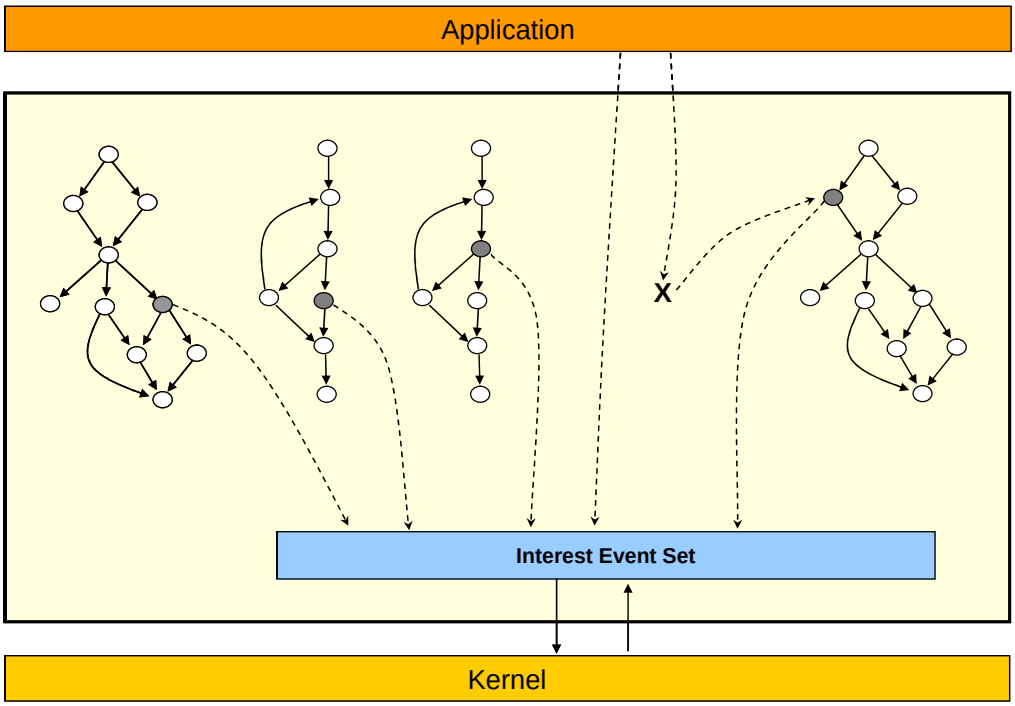


Figure-6.3: Event handling. *Events of interest are collected from state machines of each socket. Gray circles represent the current states of state machines. The application may ask the client to add some event to the collection. However, some events may be translated into other events.*

6.2 Server Implementation

The *server* is implemented as a daemon process that can be configured to use GCB, XRAY, or CODO. The *server* has a similar structure to the *client* (Figure-6.4).

Transactions in a *server* such as an inbound or outbound connection arrangement always start with a peer (i.e. a *client* or another *server*) making a TCP connection to the *server*. Requests and responses for a transaction are exchanged via the TCP connection. When a connection is accepted, the *server* creates a 'client' structure that represents a transaction (or a request that the *server* must handle). The *server* also creates two 'relay' structures to represent a relay point (i.e. XRAY and GCB cases). 'Client' and 'relay' structures are linked from the 'fd table' that is indexed by the socket descriptor. Each 'client' and 'relay' structure is associated with a state machine as the *client*

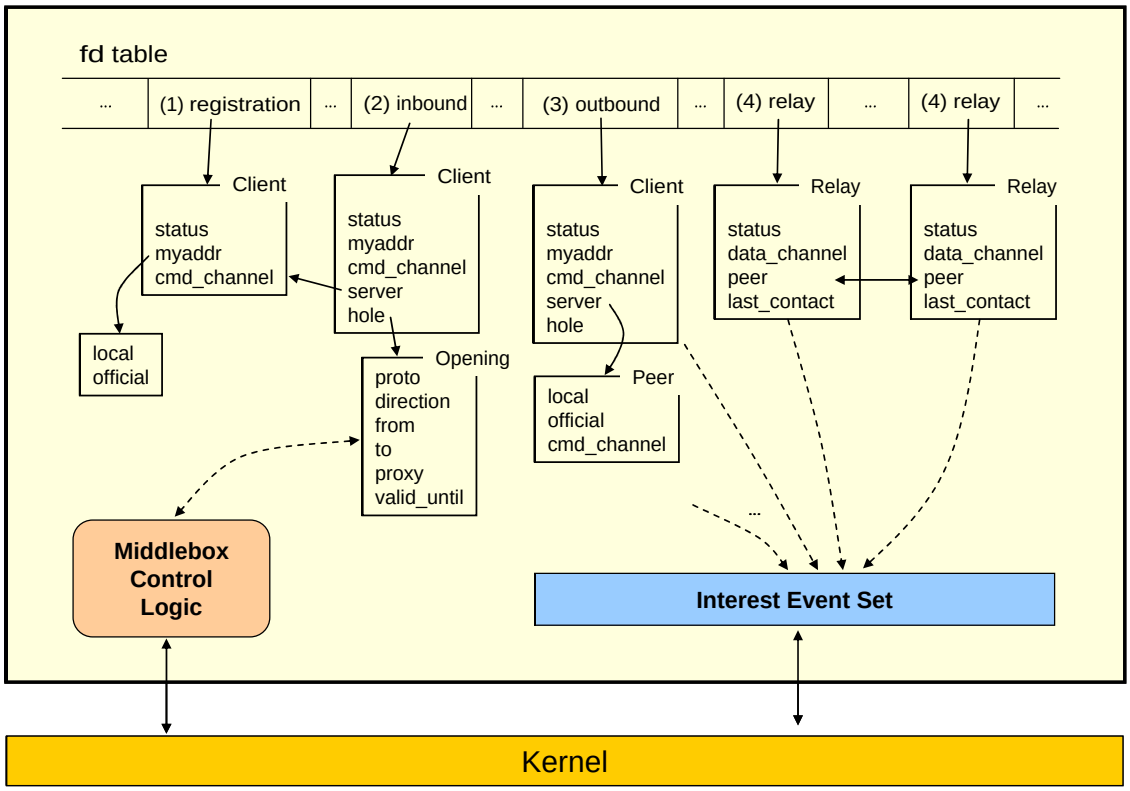


Figure-6.4: Server structure. *Each transaction and relay point is represented by 'client' and 'relay' structures, respectively, which are linked from 'fd table'. Events of interest are dynamically collected into 'interest event set'. 'Middlebox control logic' interacts with Netfilter to add or delete rules to the kernel.*

implementation.

The structure for a registration (1) contains information about a passive socket being registered. The 'myaddr' field contains the local and official addresses of the passive socket. Commands can be sent to or received from the socket via the 'cmd_channel'. The 'status' contains the current state of the registration.

The inbound structure (2) contains information about the active socket (i.e. the source endpoint) of the inbound connection being arranged. The 'status' contains the state of the inbound arrangement. The 'myaddr' contains the local and official addresses of the active socket. Commands are exchanged via the 'cmd_channel'. The 'server' points the 'client' structure that corresponds to the passive (i.e. destination) socket of the connection being arranged. The inbound structure may have an opening ('hole') if the *server* creates a

pinhole for the inbound connection (i.e. CODO case). The ‘opening’ structure represents a firewall opening or NAT binding. Similarly, the outbound structure (3) contains information about the active socket of the outbound connection being arranged. It has very similar fields as the inbound one.

Each opening has the expiration time (‘valid_until’). As soon as the *server* notices that the stateful middlebox creates a state entry for a connection, it deletes the opening that it has created for the connection. Any opening that is not deleted until its expiration time is deleted by a periodic cleanup routine.

A relay point is represented by two ‘relay’ structures linked to each other via the ‘peer’ field (4). Each relay point is implemented by two state machines—one for each half (i.e. the ‘relay’ structure). Each half independently establishes an overlay link by executing its state machine. However, data relay (through ‘data_channel’) starts only when both overlay links have been established. Each time a message is relayed, the ‘last_contact’ is updated with the current time. Relay points that have been inactive for a while are deleted by the periodic cleanup routine.

The ‘middlebox control logic’ dynamically creates openings at middleboxes. Currently it is implemented using Linux Netfilter libraries [32]. It also deletes openings when the state entries are created inside a stateful middlebox. Originally we used Netfilter’s user space packet-processing mechanism. Netfilter allows user processes to specify various conditions and to handle packets satisfying those conditions. In addition to the rule that opens the middlebox for a connection, the ‘middlebox control logic’ adds a rule which asks Netfilter to pass packets from the connection after its state becomes “ESTABLISHED”, which is equivalent to the condition that the state entry has been created for the connection. When such packets are passed by Netfilter, the *server* deletes the rules (for allowing the connection and for catching packets) and pushes them back into the kernel for forwarding. This mechanism worked very well. However, we found that Netfilter’s user space packet-processing mechanism is very slow. It took about 20 msec to get a packet from the kernel and to push it back into the kernel. Because of this overhead, CODO connection setup was very inefficient.

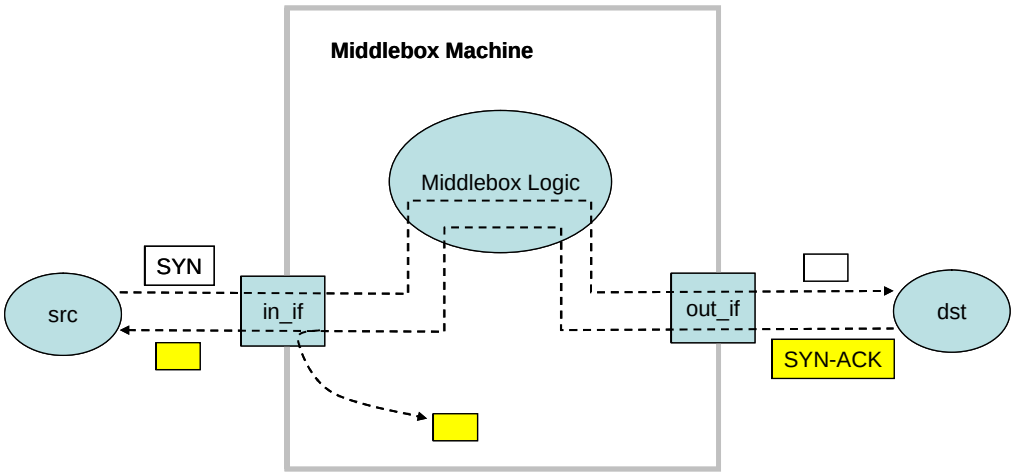


Figure-6.5: Catching state creation. *Within a middlebox machine, a SYN or initial UDP message traverses the input interface ('in_if'), IP layer and middlebox logic, and the output interface ('out_if') in that order. Reply packets traverse in the opposite order. Reply packets are caught at the input interface to notice a state entry creation.*

To handle this problem, we switched to use *libpcap* [99]. When opening a middlebox for a connection from a source to a destination, the *server* registers at the input interface a *filter* that catches reply packets from the destination to the source (with SYN-ACK flag on for TCP) (Figure-6.5). When such packets are caught, the *server* deletes the opening and the filter. We deliberately register the filter to the input interface. Reply packets arrive to the input interface only if they are allowed by the middlebox logic. Therefore, it is almost certain that a state entry has been created when a reply packet is caught at the input interface[10]. Unlike our previous approach, the kernel immediately forwards packets after it copies them to the user space without waiting for the verdict from the user process. For this reason, this mechanism is much faster than the previous one.

[10] We have never seen any middlebox implementation that defers state creation beyond this point.

7. Experiments

To measure the performance, we set up two private networks. Each network has a Linux NAT box with two network interfaces as a headnode. 100Mbps Ethernet connects nodes within each private network. A departmental network (100Mbps) connects the two private networks. Every machine has two 2.4 GHz CPUs with 512KB cache and 2GB RAM.

Using a test suite that we wrote, we measured TCP connection setup and data transfer times. In our test suite, a client program makes a connection to a server and then sends 100 messages of 10K bytes long back-to-back. The server echoes back to the client. Upon receiving all echoes, the client tears down the connection. The whole process is repeated multiple times as specified as an argument to the client program. We used X.509 (RSA) certificate for authentication and session key establishment. SHA-1 and 3DES were used for integrity and encryption, respectively. In order to understand the overhead, we did the same experiments for regular direct connections. For direct connection, we manually configured middleboxes to allow direct communications between the two networks. We also modified our test programs to use regular socket calls instead of our calls in this case.

For UDP, we did a similar experiment. A client program connects a UDP socket to a server and then sends a 1K message with the timestamp. The server echoes back to the client. The client measures both connection time and the round-trip time of the message. If a message is not echoed within 10 seconds, the client gives up and proceeds to the next experiment. The same security mechanisms are used as the TCP experiment.

Our *client* and *server* implementation use the session resumption [83] to avoid the expensive public key mechanism during the authentication process. Entities reuse security sessions to communicate with others they have recently talked with. In our experiments, the same client and server programs talk with each other repeatedly. Therefore, most of security handshakes are performed without certificate exchange or

public key operations. This will certainly make the connection overhead of our mechanisms look smaller than what will be observed in real situations. However, it is very likely in our connection arrangement scheme that same entities talk with each other over and over: a client application has a good chance that it has already talked with the local agent during a previous outbound connection; there is a good chance that two agents contact each other over and over; at the time of a connection being arranged to a server, the server application and its local agent must have talked with each other during the registration process; and so on. Therefore, the session resumption will happen very often. For example, an experiment using a client in a network and multiple servers in another network may result in the same number of session resumptions used as our experiment. To avoid using too many session resumptions, we purged session caches every 20 connections. After 20 connections, every entity was reset as if it had never talked with any peer.

We performed TCP and UDP experiments explained above for each middlebox traversal mechanism and for three different configurations: private-to-private, public-to-private, and intra-network communications. In private-to-private experiments, both networks were configured to use the same traversal mechanism. Note that in GCB private-to-private experiment, the client program directly contacts the GCB agent of the server network because GCB targets networks that allow outbound connections.

Figure-7.1 shows the TCP connection and data transfer time of each traversal mechanism. As shown in the figure, there is a fair amount of overhead in connection setup (i.e. latency). On the other hand, overheads for data transfer (i.e. bandwidth) are much smaller. We should note that the major part of the overhead (particularly in connection setup) was caused by the security mechanisms we used. If faster mechanisms had been used, the overhead would have been much smaller.

Our three mechanisms show almost the same performance results for intra-network communications. This is because three mechanisms are identical in intra-network connection setup and data transfer—i.e. the client application asks to the local agent for a connection setup to a server application, then the agent tells that a direct connection can be made to the server application at its local address. Also note that overheads in intra-

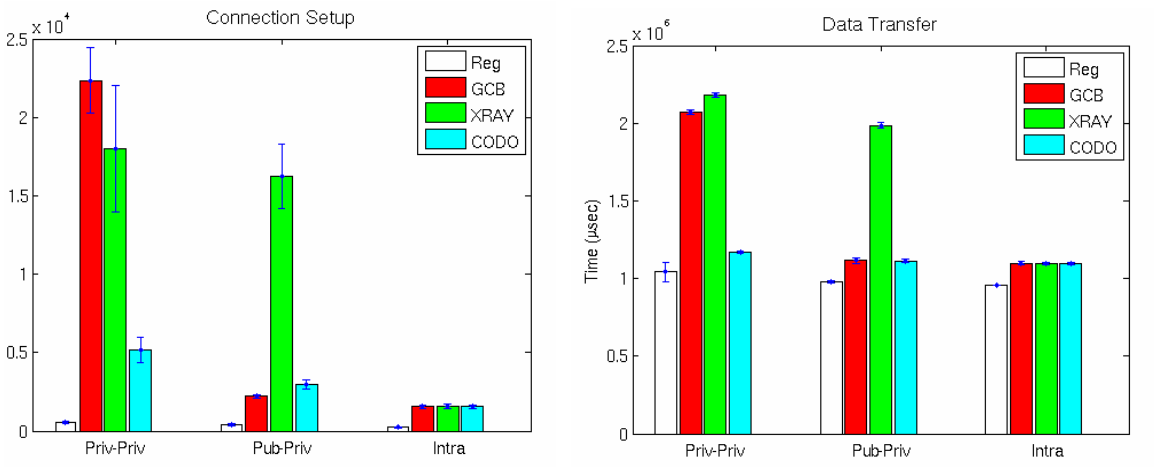


Figure-7.1: TCP performance. *Each bar shows the average connection time of the corresponding middlebox traversal mechanism for each configuration. Error bars represent the standard deviation of the corresponding experiment.*

network connection setup would have been much smaller if test networks use public addresses—i.e. the client application can make a direct connection to the server application without having to ask for a connection arrangement to the local agent.

As expected, XRAY has the biggest overheads in most cases. In order to set up a channel between applications using XRAY, multiple secure TCP connections must be established. For example, five secure TCP connections are made for a private-to-private connection: two for XRAY command exchanges between the client application and the client agent and between the client agent and the server agent, respectively; three overlay links for end-to-end channel between the client application, the client agent, the server agent, and the server application. Considering the number of connections and interactions in XRAY, connection times of XRAY are very short. We measured direct OpenSSL [71] connection times for the same private-to-private settings. It turned out that XRAY is only 1.4 times slower than OpenSSL in connection setup. We determine that the concurrent connection setup for end-to-end channel (5.2.2) and the session resumption help the connection performance of XRAY (and the other two) mechanisms.

CODO has the smallest overheads in most cases. Public-to-private connection setup is the only exception. It has less than 5 msec overheads in connection setups and about 10%

in data transfers. In CODO, applications communicate directly through openings that CODO agents made for them. Therefore, there should have been little overheads in data transfer, if any. We believe that data transfer overheads of CODO are mainly due to the buffering mechanism (see Section 6.1.1). To provide a uniform interface both for secure and insecure TCP connections, our *client* implementation does a few extra memory copies per message.

GCB shows very different performance results for private-to-private and public-to-private communications. For public-to-private, it has the smallest overheads both in connection setup and data transfer. This is because GCB simply utilizes existing openings in middleboxes and lets applications directly communicate through those openings. On the other hand, for private-to-private communications it relays application data between two private networks through secure connections. Therefore, it shows performance results similar to XRAY. In a private-to-private data transfer, GCB is slightly faster than XRAY because fewer relay points are involved in GCB. Even though GCB involves less interactions between entities than XRAY for private-to-private connection setup, its connection is slower than XRAY's. Through an investigation and profiling, we found that GCB is about 20% faster than XRAY when no session resumption is used. However, in regular cases one security handshake per a connection setup is performed without the session resumption for both GCB and XRAY, which dominate the connection time in both mechanisms. The expensive handshake occurs between the client program and the agent that creates a relay point for the client. The handshake in GCB takes much longer than that in XRAY because the relay point is created at the server's agent over the department network in GCB while it is created at the local agent over the local network in XRAY.

Figure-7.2 shows UDP connection time and data transfer time, respectively. UDP connections in regular cases do not include any network operation—i.e. the `connect` call for UDP sockets is non-blocking. Since UDP connections only take a few micro seconds in our experiment, it is hard to see bars corresponding regular UDP connections. On the other hand, there are 1 to 4 msec connection overheads in our mechanisms due to the large number of network operations involved. Because messages are signed and verified by private-public key pairs, mechanisms that use security for data

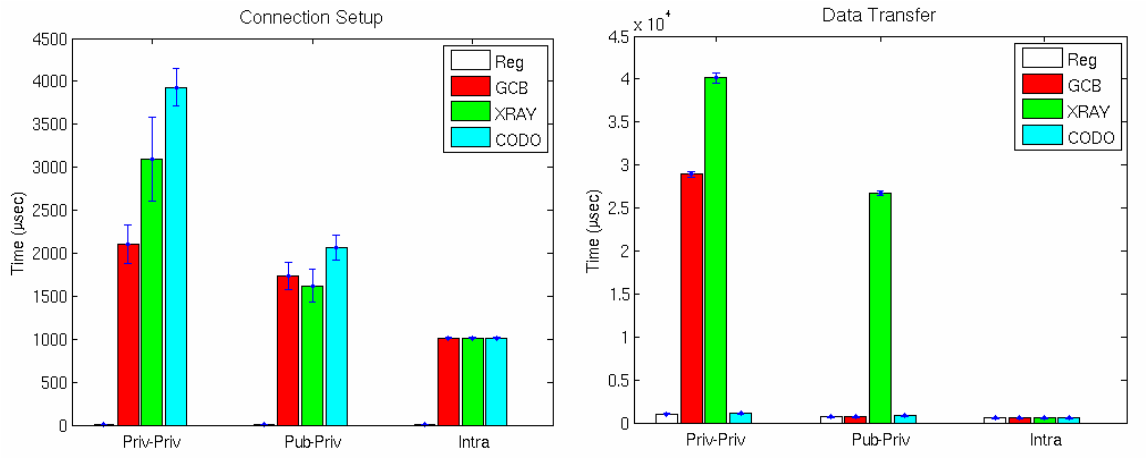


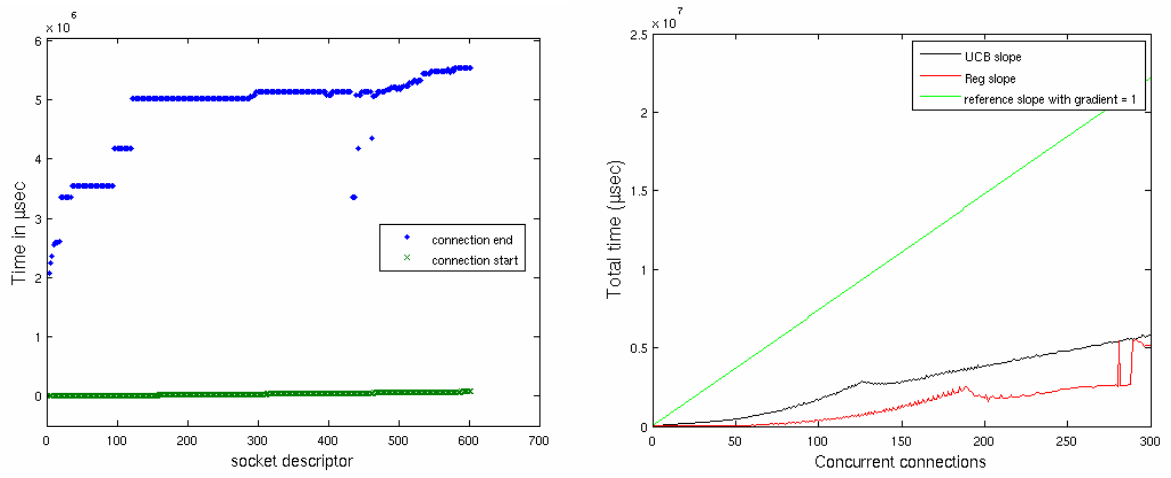
Figure-7.2: UDP performance. *Each bar shows the average connection time of the corresponding middlebox traversal mechanism for each configuration. Error bars represent the standard deviation of the corresponding experiment.*

communications have large overheads in data transfer. However, when plaintext communications are used, our mechanisms have little overhead in data transfer. As in TCP cases, our three mechanisms show the same performance result for intra-network settings.

Both for private-to-private and public-to-private settings, CODO has the biggest overhead in connection setup. Also note that in CODO, UDP connections take a little less than TCP connections. This is because UDP connection setup in CODO involves most time-consuming steps of TCP connection setup. In CODO, a UDP connection setup finishes when all openings are made at intervening middleboxes—only the actual connection through those openings is skipped. For the same reason, GCB's UDP connection for public-to-private setting has similar performance as TCP.

On the other hand, when a relay mechanism is used in GCB and XRAY, UDP connection setup is much faster than TCP and is even faster than UDP connections in CODO. In this case, a UDP connection setup finishes when relay points are created. Thus, secure connection setups via those relay points, which are one of the slowest operations in TCP cases, and acknowledgements between endpoints (5.1.1 and 5.2.2) are skipped.

The public key operations on UDP messages dominate the UDP data transfer



(a) **(b)**

Figure-7.3: Concurrent connection setup. *(a) The X-axis represents the number of the socket descriptor that non-blocking connections were issued with. The Y-axis shows the times of each connection issued ('x' mark) and finished (dot). The time difference between the first connection issued and the last connection finished is 5.54 seconds. (b) The X-axis shows the number of concurrent connections issued. The Y-axis shows the total time to set up multiple connections. The straight line shows the time that would have been taken if connections were made one-by-one. The plot in the bottom shows the result for direct connections using regular sockets.*

---

performance. XRAY in private-to-private setting, which requires the most public key operations, is the slowest in data transfer. GCB in private-to-private and XRAY in public-to-private have similar performance results because they have the same number of public operations.

To see how our system scales, we also tested concurrent connection setup. Figure-7.3 (a) shows that multiple connections can be established concurrently rather than in a serial manner. In this test, we used two private networks using different mechanisms. The client network is configured to use CODO for controlling outbound connections, while the server network uses XRAY for controlling inbound connections. A client in the client network issued 300 concurrent connections (i.e. non-blocking connections) to a server in the server network and recorded the times of connection issued and finished for each socket. The figure shows that multiple connections are established concurrently rather than one-by-one. Some connections finished earlier than those that started earlier (i.e.

three dots between socket descriptor 400 and 500). Figure-7.3 (a) also shows that all 300 connections were finished within 5.54 seconds, which is only 75 times as slow as a single connection setup using the same test program, instead of an expected 300 times for serial establishment. Figure-7.3 (b) shows how the total time varies as the number of concurrent connections increase. In this test, the client simultaneously issued up to 300 non-blocking connections to the server. The client issued one connection to measure the time of single connection setup, and then issued two connections in non-blocking fashion to measure the time to setup two connections, and so on. To compare with regular cases, we did the same experiments for direct regular connections. The result shows that our mechanisms handle concurrent connections almost as well as TCP/IP implementation.

8. Conclusion

This dissertation addressed middlebox traversal in the application layer. This chapter concludes the dissertation by briefly summarizing our contributions, and then discussing open issues and future work.

8.1 Summary

This dissertation developed a family of middlebox traversal techniques that helps us to move one step closer to the "differentially universal network", an ideal network that provides universal connectivity to benign endpoints, while isolating malicious ones. Each traversal technique can be not only an enabling tool that helps applications to communicate through middleboxes but also a security tool that network administrators may use for perimeter defense. By leveraging our framework that classifies representative middlebox policies and discusses major dimensions of middlebox traversal, we were able to develop traversal techniques which are specialized for the type of policies each targets and collectively support representative organizations. We also developed a connection arrangement scheme that coordinates our middlebox traversal techniques. The scheme allows our system to support the most complex applications that require dynamic and many-to-many communications among organizations that may have to use different techniques due to the difference in middlebox types, policies, constraints, and other criteria. The contributions of this dissertation are as follows:

- *A new perspective of middlebox traversal*. We suggested that middlebox traversal be considered not only from an application's connectivity but also from a perimeter defense perspective. Based upon both perspectives, we gave a new definition of middlebox traversal. We expect that our new definition will lead to the development of middlebox-friendly traversal mechanisms that can be used as components of perimeter defense.
- *A framework of middlebox traversal*. We believe that our framework lays a common

ground for the further discussion of middlebox traversal. Existing traversal mechanisms may be assessed using the framework. New developments may also leverage the framework: the need for new mechanisms can be identified using the framework; new mechanisms may be evaluated using the framework; and so forth.

- *An analysis of existing traversal mechanisms.* We analyzed representative traversal mechanisms under our framework. This work not only manifests the validity of the framework but also gives valuable information to users about existing systems.
- *New middlebox traversal techniques*. We developed three traversal techniques—GCB, XRAY, and CODO. Each can be used not only as a convenience tool for benign users but also a security tool for perimeter defense. Since they support representative points in the multi-dimensional space that our framework defines, they collectively support a wide variety of organizations and users.
- *A connection arrangement scheme*. The scheme combines our traversal techniques (and perhaps others in future) into a family that can support many-to-many communications between diverse organizations without sacrificing each organization's policies or constraints.
- *The implementation and performance evaluation of the mechanisms that we develop*. A library and a daemon that implement our mechanisms are now available for use. This dissertation also provides performance data for those systems.

8.2 Future Work

The work presented in this dissertation raises several questions and opens opportunities for further studies.

Issues related to addressing

There are many addressing problems related to middleboxes and middlebox traversal. Because of private IP, endpoints may not be uniquely addressed or identified by the conventional IPv4 addresses; because of address translation by NATs, two ends of a connection may have different address quadruples—i.e. (source IP, source port, destination IP, and destination port); and so forth. In this dissertation, we dealt with some of those issues while using the conventional IPv4 address. For example, the

address leasing mechanism in Section 4.2.1 ameliorates the problem of IPv4 as endpoint's locator. However, there are still many problems that this dissertation has not dealt with. Relatively large overhead for intra-network connection setup is a consequence of those remaining problems. In order to establish an end-to-end communication channel to a server, several control points such as agents in our connection arrangement scheme may have to be contacted. How a client can get the information about those control points in a scalable and efficient way is an open issue. We believe that the addressing issue must be dealt with at a lower layer such as IP. Also, other addressing problems of the Internet [48] [90] [91] [92] [93] must be considered together with problems related to middleboxes.

Integral perimeter defense scheme

We presented middlebox traversal techniques that can be security tools for network administrators. Our traversal techniques particularly bind traffic to sender/receiver applications. Other components of perimeter defense may benefit from such knowledge. For example, exact binding between traffic and applications may reduce the amount of traffic that a network NIDS must analyze. A research on integral perimeter defense scheme that includes middlebox traversal techniques as components may contribute to network security community.

Resource management

In many middlebox traversal systems including ours, the agent allocates resources for endpoints in the network it manages. For example, XRAY and SOCKS agent creates proxy sockets for endpoints. Public addresses are allocated for endpoints bound to private addresses. In consequence, applications in a network contend for resources in the agent. Researches on the management of the agent resources must be done.

References

[1] Network terminology web site, http://searchnetworking.techtarget.com

[2] K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," *IETF RFC 1631*, May 1994.

[3] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues," *IETF RFC 3234*, Feb. 2002.

[4] CERT Advisory CA-2001-19: 'Code Red'Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-19.html, July 2001.

[5] CERT Coordination Center, Denial Of Service Attacks: http://www.cert.org/ tech_tips/denial_of_service.html

[6] CERT Advisory CA-2003-04: MS-SQL Server Worm. http://www.cert.org/ advisories/CA-2003-04.html, January 2003.

[7] I. Foster, C Kesselman and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Intl. Journal of Supercomputing Applications*, 2001.

[8] Globus web site, http://www.globus.org

[9] Condor web site, http://www.cs.wisc.edu/condor

[10] M. Litzkow, M. Livny and M. Mutka, "Condor - A Hunter of Idle Workstations," *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104-111, June, 1988.

[11] P. Druschel and A. Rowstron, "Past: A large-scale, persistent peer-to-peer storage utility," in *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, (Schoss Elmau, Germany), May 2001.

[12] S. Ratnasamy, et el., "A scalable content-addressable network," in *Proceedings of the ACM SIGCOMM 2001 Technical Conference*, (San Diego, CA, USA), August 2001.

[13] I. Stoica, et el., “Chord: A scalable content-addressable network,” in *Proceedings of the ACM SIGCOMM 2001 Technical Conference*, (San Diego, CA, USA), August 2001.

[14] M. Waldman, A. Rubin and L. Cranor, “Publius: A robust, tamper-evident, censorship-resistant, web publishing system,” in *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[15] I. Clarke, et el., “Freenet: A Distributed Anonymous Information Storage and Retrieval System,” *In Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.

[16] B. Zhao, J. Kubiatowicz and A. Joseph, “Tapestry: An Infrastructure for Fault-ResilientWide-area Location and Routing,” *Technical Report* UCB//CSD-01-1141, U. C. Berkeley, April 2001.

[17] Checkpoint web site, http://www.checkpoint.com

[18] Application intelligence white paper, http://www.checkpoint.com/products/downloads/applicationintelligence_whitepaper.pdf

[19] R. Fielding, et el., “Hypertext Transfer Protocol—HTTP/1.1,” *IETF RFC 2616*, June 1999.

[20] T. Ylonen and C. Lonvick, Ed., “The Secure Shell (SSH) Protocol Architecture,” *IETF RFC 4251*, January 2006.

[21] T. Ylonen and C. Lonvick, Ed., “The Secure Shell (SSH) Connection Protocol,” *IETF RFC 4254*, Jan. 2006.

[22] P. Resnick, “Internet Message Format,” *IETF RFC 2822*, April 2001.

[23] N. Freed and N. Borenstein, “Miltipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,” *IETF RFC 2045*, Nov. 1996.

[24] N. Freed and N. Borenstein, “Miltipurpose Internet Mail Extensions (MIME) Part Two: Media Types,” *IETF RFC 2046*, Nov. 1996.

[25] K. Moore, “MIME (Miltipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text,” *IETF RFC 2047*, Nov. 1996.

[26] N. Freed, J. Klensin and Jon Postel, “Miltipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures,” *IETF RFC 2048*, Nov. 1996.

[27] N. Freed and N. Borenstein, “Miltipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples,” *IETF RFC 2046*, Nov. 1996.

[28] D. Thain and M. Livny, "Multiple Bypass: Interposition Agents for Distributed Computing," *The Journal of Cluster Computing*, Volume 4, 2001, pp 39-47.

[29] D. Thain and M. Livny, "Parrot: Transparent User-Level Middleware for Data-Intensive Computing," *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, September 2003.

[30] M. Roesch, "Snort: the Open Source Network Intrusion Detection System," Available at www.snort.org.

[31] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, 31(23/24), Dec. 1999.

[32] Netfilter web site, http://www.netfilter.org

[33] B. Chazelle. "Lower Bounds for Orthogonal Range Searching, I: The reporting case," *J. of the ACM*, 37, pp. 200-212, 1990.

[34] B. Chazelle. "Lower Bounds for Orthogonal Range Searching, II: The arithmetic model," *J. of the ACM*, 37, pp. 439-463, 1990.

[35] M. Overmars and A. F. van der Stappen, "Range Searching and Point Location among Fat Objects," *J of Algorithms*, 21(3), pp 629-656, 1996.

[36] T. Lakshman and D. Stidialis, "High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," *Proc. of SIGCOMM '98* Sept. 1998.

[37] V. Srinivasan, et al., "Fast Scalable Level Four Switching," *Proc. of SIGCOMM '98*, Sept. 1998.

[38] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. of SIGCOMM '99*, Sept. 1999.

[39] P. Gupta and N. McKeown. "Packet Classification using Hierarchical Intelligent Cuttings," *Proc. of Hot Interconnects* VII, Aug. 1999.

[40] J. Xu, M. Singhal and J. Degroat, "A Novel Cache Architecture to Support Layer-Four Packet Classification at Memory Access Speeds," *Proc. of INFOCOM 2000*, March 2000.

[41] L. Qiu, G. Varghese and S. Suri, "Fast Firewall Implementation for Software and Hardware Based Routers," *Proc. of the ICNP 2001*, Nov. 2001.

[42] D. Rovniagin and A. Wool, "The geometric efficient matching algorithm for firewalls," *Tech. Rep.*, Dept. of Electrical Engineering Systems, Tel Aviv University, Ramat Aviv 69978, Israel, 2003.

[43] K. Lakshminarayanan, A. Rangarajan and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," *Proc. of SIGCOMM 2005*, Aug. 2005.

[44] D. Hartmeier, "Design and performance of the OpenBSD stateful packet filter (pf)," *Proc. FREENIX Track: 2002 USENIX Annual Technical Conference*, June 2002.

[45] J. Postel and J. Reynolds, "File Transfer Protocol", *STD* 9, *IETF RFC 959*, October 1985.

[46] J. Postel, "INTERNET CONTROL MESSAGE PROTOCOL," *IETF RFC 792*, September 1981.

[47] CERT Advisory CA-1988-01, Smurf IP Denial-of-Service Attacks.

[48] H. Balakrishnan, et al., "A Layered Naming Architecture for the Internet," *Proc. of SIGCOMM 2004*, Sept. 2004.

[49] S. Kent and P. Atkinson, "Security Architecture for the Internet Protocol", *IETF RFC 2401*, Nov. 1998.

[50] G. Montenegro and M. Borella, "RSIP Support for End-to-End IPSEC", *IETF RFC 3104*, Oct. 2001.

[51] S. Son and M. Livny, "Recovering Internet Symmetry in Distributed Computing," *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

[52] T. Kosar, S. Son, G. Kola, and M. Livny. "Data placement in widely distributed environments," In *Advances in Parallel Computing*, Elsevier Press, 2005.

[53] J. Rosenberg, J. Weinberger, C. Huitema and R. Mahy, "STUN – Simple Traversal of User Data Gram (UDP) Through Network Address Translators (NATs)," *IETF RFC 3489*, March 2003.

[54] J. Rosenberg, R. Mahy and C. Huitema, "Traversal Using Relay NAT (TURN)," *Internet-Draft*, Sept. 2005.

[55] P. Srisuresh, et al., "Middlebox Communication Architecture and Framework," *IETF RFC 3303*, Aug. 2002.

[56] M. Leech, et el., "SOCKS Protocol Version 5," *IETF RFC 1928*, March 1996.

[57] D. R. Cheriton and M. Gritter, "TRIAD: A New Next Generation Internet Architecture," March 2000. http://www-dsg.stanford.edu/triad/triad.ps.gz.

[58] P. Francis and R. Gummadi, "IPNL: A NAT-Extended Internet Architecture," *SIGCOMM 2001*, Aug. 27, 2001.

[59] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *IETF RFC 2460*, Dec. 1988.

[60] UPnP website, http://www.upnp.org

[61] UPnP architecture document, http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf

[62] M. Borella and G. Montenegro, "RSIP: Address Sharing with End-to-End Security", *Special Workshop on Intelligence* at the Network Edge, San Francisco, 2000.

[63] M. Borella, J. Lo, D. Grabelsky and G. Montenegro, "Realm Specific IP: Framework," *IETF RFC 3102*, July 2000.

[64] D. Anderson, et el. "Resilient Overlay Networks," *18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[65] C. Kaufman, R Perlman and M. Speciner, Network Security: Private communication in a Public World, 2nd Edition, *Prentice Hall*, Chapter 23. pp 585-594.

[66] W. Cheswick, S. Bellovin and A. Rubin, Firewalls and Internet Security: Repelling the Wily Hacker, 2nd Edition, *Addison-Wesley*, Chapter 1, pp. 3-18.

[67] MIDCOM Working Group web site, http://www.ietf.org/html.charters/midcom-charter.html

[68] GNU Virtual Private Network web site, http://savannah.gnu.org/projects/gvpe

[69] SoftEther web site, http://www.softether.com/jp

[70] M. Blumenthal and D. Clark, "Rethinking the design of the Internet: The end-to-end argument vs. the brave new world," A*CM Transactions on Internet Technology*, *Vol. 1*, No. 1, 2001.

[71] OpenSSL web site, http://www.openssl.org

[72] JXTA web site, http://www.jxta.org

[73] S. Son, M. Farrellee and M. Livny, "A Generic Proxy Mechanism for Secure Middlebox Traversal,", *Cluster 2005* Boston, Massachusetts, September 27 - 30, 2005.

[74] S. Son, B. Allcock and M. Livny, "CODO: Firewall Traversal by Cooperative On-Demand Opening," *HPDC 2005*, Research Triangle Park, NC, July, 2005.

[75] S. Klous, et el., “Transparent Access to Grid Resources for User Software,” *Concurrency and Computation: Practice and Experience*, accepted for publication *(2005)*.

[76] A. Luotonen and K. Altis, “World-Wide Web Proxies,” *Computer Networks and ISDN Systems*, vol. 27 no.2 147-154, 1994.

[77] “Napster Protocol Specification,” http://opennap.sourceforge.net/napster.txt

[78] “The Gnutella Protocol Specification v0.4 Document Revision 1.2,” http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[79] H. Schulzrinne, et el., “RTP: A Transport Protocol for Real-Time Applications,” *IETF RFC 3550*, July 2003.

[80] M. Handley and V. Jacobson, “SDP: Session Description Protocol,” *IETF RFC 2327*, April 1998.

[81] J. Rosenberg, et el., “SIP: Session Initiation Protocol,” *IETF RFC 3261*, June 2002.

[82] ITU-T Recommendation H.323. “Packet-based Multimedia Communications Systems,” 1998.

[83] T. Dierks and C. Allen, “The TLS Protocol,” *IETF RFC 2246*, Jan. 1999.

[84] W. Stevens, TCP/IP Illustrated Vol. I: The Protocol, *Addison-Wesley*, 1994.

[85] G. Wright and W. Stevens, TCP/IP Illustrated Vol II: The Implementation, *Addison-Wesley*, 1995.

[86] G. Ziemba, D. Reed and P. Traina, “Security considerations for IP fragment filtering,” *IETF RFC 1858*, Oct. 1995.

[87] C. Kent and J. Mogul, “Fragmentation considered harmful,” In *WRL Technical Report* 87/3, December 1987.

[88] C. Kreibich, M. Handley and V. Paxson, “Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics,” *Proc. USENIX Security Symposium 2001*, 2001.

[89] S. Bellovin, “Distributed firewalls”, *;login: the USENIX Association newsletter*, vol. 19, no. Special issue on security, pp. 39-47, Nov. 1999

[90] D. Clark, R. Braden, A. Falk and V. Pingali, “FARA: Reorganizing the addressing architecture,” *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, Karlsruhe, Germany, Aug. 2003.

[91] R. Moskowitz and P. Nikander, “Host identity protocol architecture,” Aug. 2005. draft-ietf-hip-arch-03, *IETF draft* (Work in Progress).

[92] P. Nikander, J. Ylitalo and J. Wall, “Integrating security, mobility, and multi-homing in a HIP way,” *Network and Distributed Systems Security Symposium (NDSS ’03)*, pages 87–99, San Diego, CA, February 2003.

[93] M. Walfish, H. Balakrishnan and S. Shenker, “Untangling the Web from DNS,” *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI ’04)*, San Francisco, CA, March 2004.

[94] A. Wool, “A Quantitative Study of Firewall Configuration Errors,” *IEEE Computer*, 37(6):62 - 67, 2004.

[95] E. Bugnion, S. Devine, K. Govil and M. Rosenblum. “Disco: Running Commodity Operating Systems on Scalable Multiprocessors,” *ACM Transactions on Computer Systems*, 15(4), November 1997.

[96] R. J. Creasy, “The Origin of the VM/370 Time-Sharing System,” *IBM Journal of Research and Development*, 25(5), September 1981.

[97] P. Barham, et el., “Xen and the Art of Virtualization.” *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp 164-177, Bolton Landing, NY, USA, 2003.

[98] GNU compiler web site, http://gcc.gnu.org

[99] Tcpdump and libpcap web site, http://www.tcpdump.org

[100] Global Grid Forum web site, http://www.ggf.org