A Generic Proxy Mechanism for Secure Middlebox Traversal

Sechang Son, Matthew Farrellee, and Miron Livny
Computer Science Department, University of Wisconsin
{sschang, matt, miron}@cs.wisc.edu

Abstract

Firewalls/NATs have brought significant connectivity problems along with their benefits, causing many applications to break or become inefficient. Due to its bi-directional communication, huge scale, and multi-organizational nature, the Grid may be one of the areas damaged most by the connectivity problem. Several ideas to deal with the connectivity problem were investigated and many systems are available. However, many issues still remain unanswered. Most systems are firewall/NAT unfriendly and are considered harmful to network security; the tussle between these devices trying to investigate payloads and applications trying to protect their content from observation and modification must be reconciled. This paper discusses how a simple relay-based system, called XRAY (middleboX traversal by RelAYing), deals with these issues and provides other benefits such as flexible traffic control. This paper also discusses how relay-based traversal systems can help applications to communicate over firewalls/NATs and also complement firewall/NAT operations to help network security.

1. Introduction

Firewalls and NATs [1] (collectively called *middleboxes*[1] in this paper) provide many benefits such as easy address planning, network protection, and a solution to the IPv4 address shortage. However, these devices come at a price, notably non-universal connectivity of the Internet. In general, two endpoints separated by one or more middleboxes cannot communicate with each other. The Internet has become asymmetric because most middleboxes allow outbound (to the world) but block inbound (from the world) communications. Due to this connectivity problem, many applications break or become inefficient. The Grid [2] may be one of the most damaged areas because it generally requires bi-directional and many-to-many connectivity among geographically distributed organizations. Client-server applications can get around the asymmetry problem by placing servers in publicly accessible places such as a DMZ. This approach does not work for the Grid because a node may act both as a client and a server. In grids, the connectivity problem generally results in the waste of resources because researchers may not harness resources separated from their networks by middleboxes. Computing jobs cannot be staged from the public network into a network behind a middlebox, and vice versa [4] [5]; data placement cannot be completed because data cannot move into or out of a network behind a middlebox.

Middleware approaches are very attractive for dealing with the connectivity problem. They are easy to deploy because neither the Internet nor operating systems need be changed, and many applications can benefit from them. Especially middleware providing APIs similar to the Berkeley socket API is desirable for easy deployment because it is well understood and is used by many network applications. Many middleware traversal mechanisms were studied or are under investigation for dealing with the connectivity problem. However, we still have many problems and issues left unanswered:

- **Middlebox friendly?** Some traversal systems, "often ironically called firewall-friendly" [17], have adverse effects to network security. Some systems, notably in P2P file sharing systems, disguise their traffic to deceive middleboxes (or administrators). Other systems such as GCB [10], STUN [11], and TURN [12] exploit common middlebox behavior or configuration to the extent that network administrators never intended. For this reason, network administrators generally consider middlebox traversal systems to harm network security and are reluctant to deploy them. Systems such as DPF [10],

[1] Though IETF uses "middleboxes" to refer to more than just NATs and firewalls [13], it currently focuses on those two devices.

SOCKS [14], and RSIP [15] have little or no adverse effect on network security. However, these systems suffer from similar problems because they do not describe how their traversal mechanisms fit in with network security enforcement.

- **Asymmetry.** Most middleboxes are configured to allow outbound connections while blocking inbound ones. Using this common practice, previous systems assume that outbound connections are allowed and help applications only with inbound connections. More and more organizations want to control communications in both directions for reasons such as security and legal issues. To support such restrictive organizations, a traversal mechanism must help both inbound and outbound connections in a controlled manner.
- **"Tussle" [3] between applications and middleboxes.** Many applications encrypt contents with strong security mechanisms to protect their payloads from observation or modification. On the other hand, some middleboxes want to inspect payloads for better filtering, intrusion detection, etc. When those middleboxes cannot look inside packets, they generally drop packets. Therefore, we must find a resolution or reasonable compromise of this tussle.

This paper discusses how well a simple relay based system, called XRAY (middleboX traversal by RelAYing), deals with these issues and provides other benefits such as flexible traffic control. XRAY helps authorized applications to traverse middleboxes by relaying both inbound and outbound traffic. It also helps network security by dropping packets for unauthorized applications. Since it provides the Berkeley socket API, any network application can be easily XRAY enabled. In our previous work [16], we presented CODO (Cooperative On-Demand Opening), which provides similar benefits as XRAY for organizations using the middleboxes that can be dynamically controlled by the add-on software we provide. However, unlike CODO, XRAY does not require dynamic control over middleboxes and has no restrictions on the type of middlebox it can support. XRAY also provides stronger control over traffic with a minor amount of performance overhead compared to CODO. SOCKS [14] also shares many characteristics and benefits with XRAY. However, it is designed only for client-server applications and may not be used by the Grid. We will further explain this and another limitation of SOCKS in §9.

Relaying mechanisms have been considered as a secure way of middlebox traversal for years. Our contribution is (1) the reconsideration of the relaying mechanism as a middlebox traversal system in a broader and formalized context and (2) XRAY, a relay-based system, which realizes the benefits of the relaying mechanism.

In §2, we present a packet flow model within a middlebox and define the middlebox traversal problem within that model [16]. We include the model to make the paper self-contained. In §3, we introduce a concept that is important to secure traversal of middleboxes. The architecture and connection procedure of XRAY are presented in §4 and §5, respectively. §6 discusses the fault tolerance issue and §7 explains the implementation. §8 and §9 present performance data and related research, respectively.

2. Model and Problem Definition

The middlebox traversal problem has been around for many years, yet it remains vaguely defined, raising many questions such as "if a middlebox is opened for an application, does it blindly pass packets to/from the application?" and "how does a traversal mechanism fit in the security policy the middlebox tries to enforce?" To avoid confusion, we define the problem as follows.

Middleboxes block malicious or unwanted traffic while allowing benign and desired traffic. What is malicious or unwanted (or equivalently benign and desired) is defined by middlebox rules. To traverse a middlebox, a packet must pass a chain of one or more tests defined by the middlebox rules. If a packet fails a test, it is rejected. Otherwise, it continues to traverse the chain of tests until it fails a test or passes all the tests.

Figure 1 shows a packet flow model in a middlebox. When a packet enters a middlebox, it undergoes one or more tests that we collectively call the *application-neutral test.* This test specifies application-independent conditions such as IP address, source routing flag, and ICMP message type. This test drops packets considered dangerous no matter what application sends or receives them. For example, overly fragmented packets are considered dangerous and may be dropped at this stage. If a packet passes this test, it is either accepted or sent to the *owner test*. The owner test allows traffic for authorized applications and blocks traffic for unauthorized or dangerous applications. For example, many middleboxes allow SSH but block telnet and rlogin traffic. If a packet belongs to an authorized application, it may be allowed or sent to *auxiliary tests* specifically designed for individual applications. If an application is known to be vulnerable, say to a buffer overflow attack, an administrator may have an owner test rule to block the application. However, a better approach is to pass the application traffic only if it does not contain an attack signature. The auxiliary tests can be used to block only malicious packets while allowing benign ones.

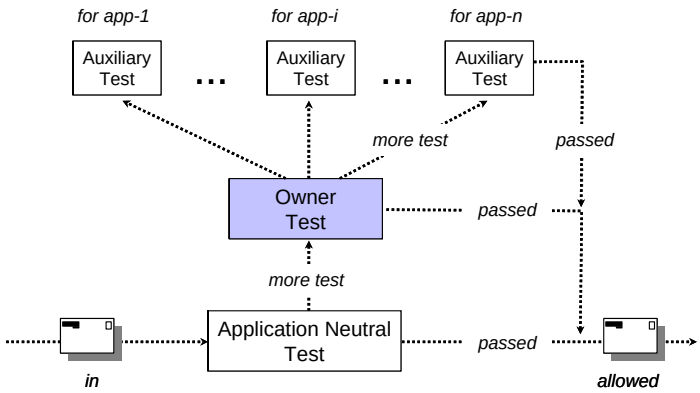

Figure 1: Packet flow model. *Within a middlebox, packets traverse application-neutral, owner, and auxiliary tests in that order. Auxiliary and owner tests are not applied for some packets.*

Depending on middlebox implementations and configurations, packets may flow differently from our model: tests may be applied in a different order; multiple tests from different stages may be combined; some tests are not available in a middlebox and may be performed by a third party product such as an IDS (Intrusion Detection System) [9]. However, we believe that this model is general and accurate enough for our discussion.

We define the connectivity problem as a situation where a desirable (and benign) application cannot traverse a middlebox. We believe that the problem occurs mostly because benign applications fail the owner test (*false negative*), as middleboxes are overzealous in blocking malicious applications. The owner test is also very important to network security because errors in this test may result in (1) malicious or undesirable applications passing middleboxes (*false positive*) or (2) incorrect auxiliary tests being applied to packets, resulting in false negatives and false positives. For these reasons, this paper (and middlebox traversal problems in general) focuses on the owner test. Our goal is to satisfy the following requirement:

Authorized applications' traffic must pass the owner test and unauthorized traffic must not.

Note that the owner test alone does not define the fate of a packet. The packet may fail a test before or after the owner test. Also, note that the problem is defined both from application and network security perspectives. Therefore, our goal is to develop a mechanism that helps applications to traverse middleboxes and helps (or complements) middleboxes with the owner test.

3. Owner Binding

To perform the owner test, a middlebox must know whether the packet under scrutiny is for an authorized application or not. In addition, knowing the sender/receiver applications of authorized packets is essential for logging and for performing further application specific tests (i.e. auxiliary tests in figure 1). Given a packet *p*, we define the *owner binding* $OB_M(p)$ as the mapping function of a middlebox *M* such that

$$OB_M(p) = \begin{cases} \textit{sender / receiver application } A \textit{ of } p, \\ \quad \textit{if } A \textit{ is authorized to traverse } M \\ \textit{null, otherwise} \end{cases}$$

Note that the owner test problem becomes trivial once we can decide the owner binding. If $OB_M(p)$ returns null, the packet *p* fails the owner test at *M*. If an authorized application *A* is returned, the packet *p* passes the owner test and is sent to the auxiliary test for *A*, if any. Thus, an error-free owner binding is the strongest prerequisite for an error-free owner test. Unfortunately, the owner binding is not easy to do because packets generally do not convey information about their source/destination applications. Almost every middlebox uses port numbers to bind packets to their owner applications. For example, middleboxes often consider packets with port 80 as Web traffic. However, a port number is at most a hint to an application's identity because it is a shared resource used by any application with the appropriate privileges. Some P2P file sharing systems use port 80 and wrap their traffic in HTTP messages to deceive middleboxes. We may regard that these systems exploit inherent errors in using port numbers for the owner binding. Recognizing this problem, recent middleboxes [6] investigate payloads and drop packets if their traffic does not follow normal web semantics. Such middleboxes use both port number and content investigation for the owner binding. However, this type of testing cannot be perfect and may not even be

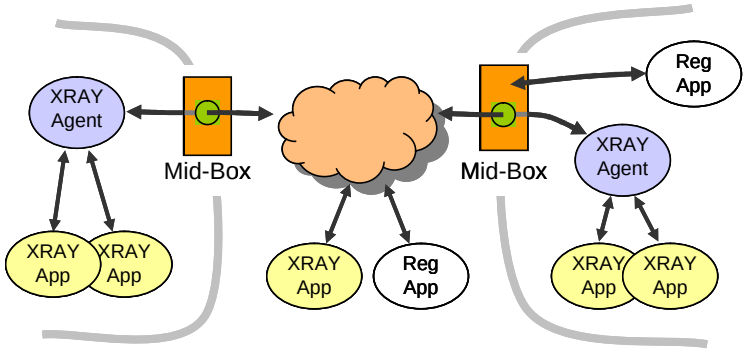

Figure 2: XRAY topology. *Middleboxes trust their XRAY agents and allow packets to/from the agents. XRAY agents relay traffic for applications the network administrator authorizes. Non-XRAY applications may or may not traverse their middleboxes using other mechanisms.*

possible for dynamic applications whose traffic cannot be understood by peeking at payloads.

4. Architecture

Figure 2 shows a typical topology of XRAY. Each middlebox trusts one or more XRAY agents of the network it protects and bypasses some tests for packets addressed to/from them. Each XRAY agent is assigned a set of authorized applications and relays traffic for them but drops for others. To use XRAY, organizations must add or change a few middlebox rules but need not change middleboxes.

Figure 3 shows XRAY components and their interactions. The figure also shows how XRAY fits in the packet flow model of §2. A middlebox filters or passes ordinary packets using various tests as explained in §2. However, it performs only the application neutral test for packets to/from an XRAY agent and delegates the remaining tests to the agent. An XRAY agent has a list of authorized applications that can communicate over its middlebox. Since the middlebox and the XRAY agent collectively enforce the security policy of the network, the list must be considered as a part of middlebox rules. The XRAY library reports, via XRAY commands, its XRAY agent about the application's activities such as listening on a socket, trying to connect a socket to a server outside its network, and closing a socket. Using this information and XRAY commands from remote sites, the agent dynamically creates (and deletes) relay points for the application as needed and optional plug-ins that the administrator defines for that application. Plug-ins can be used for application specific tests or logging. To make sure that only authorized applications can have relay points and accompanied plug-ins, XRAY uses strong security mechanisms for XRAY command exchanges. Strong security mechanisms also protect each relay point so that only intended application or the next hop can communicate through it. Inbound packets for an authorized application (1) undergo the application-neutral test at the middlebox, (2) are authenticated and integrity checked by the key protecting the relay point, (3) and undergo auxiliary tests defined by plug-ins attached to the relay point. The relay point, plug-ins, and the middlebox's application-neutral tests check the outbound packets, in that order. Note that a network can use XRAY and other traversal mechanisms together. In this case, traffic for XRAY-enabled applications (or claim-to-be) is controlled by the XRAY mechanism, while others are controlled by their mechanisms.

XRAY provides many benefits; either as the direct result of using the relay mechanism or those connected to its design:

- **Correct owner test**. It is very difficult for a middlebox alone to achieve a correct owner test especially when applications use dynamic ports. XRAY relays packets only for the intended application via a relay point unless an attacker breaks the security mechanism protecting the relay point. Therefore, the XRAY agent achieves a practically error-free owner binding. This guarantees a correct owner test. Unauthorized application cannot pass the owner test without knowing the address of a relay point for an authorized application and breaking the security mechanism protecting the relay point.
- **Complementing middleboxes**. Not every middlebox provides all the tests of figure 1. For instance, most packet filtering and stateful middleboxes lack auxiliary tests. Organizations can complement such middleboxes by adding plug-ins to XRAY for selected applications without any change to middleboxes.

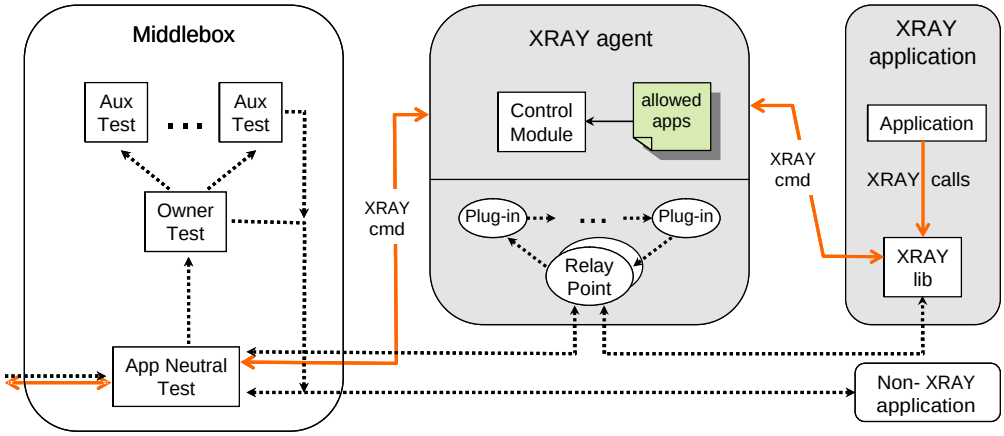

Figure 3: XRAY components. *XRAY consists of the XRAY agent and the XRAY library. The XRAY agent is a daemon process running near its middlebox machine. Applications become XRAY-enabled by linking with the XRAY library. XRAY-enabled applications use XRAY calls instead of Berkeley socket calls.*

- **Tussle compromised**. Relay points terminate security associations as well as transport connections. This hop-by-hop security provides each agent full access to payloads so that they can inspect traffic. 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 middleboxes have full access to payload, meaning applications loose some end-to-end security. However, middleboxes have the power of arbitration and may completely block applications when they cannot get information from packets.
- **Flexible control**. XRAY uses X.509 certificates to authenticate and authorize applications. This means that XRAY is very flexible and can enforce various security policies. For example, XRAY can differentiate versions or implementations of an application. If a vendor's implementation of an application turns out to be vulnerable to a dangerous attack, then it can be given a different certificate from other implementations and disallowed from communicating with the world.

Clearly, one of the biggest weaknesses of relay-based approaches is performance. Hop-by-hop encryption/decryption will slow down not only connection setup but also data transfer. Packets may have to traverse entire protocol stacks up and down in each relaying agent. Also, there are chances that many features such as reliability and flow control, which underlying network already provides, must be implemented again at a higher layer. These duplicated functionalities may make those systems slower. Some of these problems are inevitable costs for achieving a secure traversal with a desired level of control over packets, but others can be avoided or mitigated through careful design. We will explain how XRAY mitigates these performance problems in the following sections.

5. Connection Procedure

With XRAY, applications call XRAY functions. The call sequence is the same as with a Berkeley socket. For instance, a server creates a TCP socket, binds it to an address, makes it passive, and accepts connections from clients. A client creates a TCP socket, optionally binds it to an address, and connects it to a server. The server and client exchange data through the established connection. This section explains how XRAY connections are established and data are transferred over middleboxes as responses to XRAY calls from applications.

5.1. Server binding

In order to be able to accept connections from outside, server sockets behind a middlebox must be *locally bound*, *registered* to the XRAY agent of its network, and *officially bound*.

Local binding is just the regular process of binding a socket to an address. Through the local binding, an (IP, port) pair, called the local address, is assigned to the socket.

Since inbound connections are arranged by the XRAY agent of the network, enough information about a server socket must be kept in it. The *registration* process provides necessary information to the agent. After a server socket is bound to a local address, the server's XRAY library sends a registration request with the local address and the type of the socket. After

authentication/authorization and the official bind (explained shortly), the agent records the information sent by the library and other information that it collects from the official binding process.

Official binding is the process of assigning the official address, public/globally unique address, to a server socket. This is necessary to support server 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 server 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 an XRAY socket is bound to, the library returns the official address instead of the local (real) address.

After the binding process, a server socket can become *locally* bound, *half* bound, or *fully* bound. A socket is in the locally bound state if it is locally bound to a private address but could not lease a public official address because its XRAY agent is not available at binding time. Sockets in this state can accept connections that are possible without XRAY's help. A fully bound socket has a public official address (either leased or not) and is successfully registered to its XRAY agent. Sockets in this state can benefit from XRAY service. Both intra and inter network connections are possible.

A *half* bound socket occurs when registration with the agent is not possible or cancelled, but a global address is assigned. This happens when the agent is not available at the time of binding, but the socket's local address (and therefore official address) is public. A socket also becomes half bound when a socket was fully bound, but later it is deleted from the agent because of agent or network failure. As with locally bound sockets, sockets in this state can only accept connections that do not require XRAY service. However, they can become fully bound whenever the agent or the network recovers from the failure. Sockets in the half bound state periodically try to become fully bound. Note that locally bound sockets cannot be upgraded to the fully bound state because official addresses would be changed as the result of the upgrade.

5.2. Connection setup

This section explains how a client makes a connection, over middleboxes, to a server registered to its XRAY agent through the process explained in §5.1. We also assume that the client knows the official address of the server socket via an out-of-band mechanism. First, we explain how a client behind a middlebox makes a TCP connection to a server behind a different middlebox. This is the most complex and hardest situation. Connections for simpler cases are similar with some steps omitted. For UDP communications, a very similar procedure is performed when the application tries to send UDP data, by calling `XRAY_send` or `XRAY_sendto`, to a peer for the first time or after a certain inactive period. The following steps establish a connection:

(1) The client application calls `XRAY_connect` with server's official address.
(2) The client's XRAY library makes a TCP connection to the agent of the client's network (the client agent), does mutual authentication, establishes secret keys for further communications, and asks for a connection to the server. If a non-blocking connect was called, then the library issues a non-blocking connect to the agent and returns immediately. The library closely watches network events to continue remaining processes as part of other XRAY calls such as `XRAY_select` and `XRAY_connect` for other sockets.
(3) The client agent checks if the client is authorized to make outbound connections. If allowed, it makes a secure TCP connection to the agent of the server network (the server agent) and asks for a connection on behalf of the client.
(4) The server agent checks if the server is authorized to accept connections from outside the network. Then it creates a relay point and informs the client agent that it can make a connection to the relay point. The relay point is actually two sockets, one for connection from the client and the other for a connection to the server. Those sockets are associated with the certificates of the client agent and the server, respectively, so that no one else can communicate via them. If plug-ins are defined for the server, the agent attaches them to the relay point. These plug-ins could be auxiliary tests for the server, specialized log functions, etc.
(5) The client agent creates a relay point and plug-ins in the same way as the server agent did and notifies the client to connect to the relay point.
(6) Overlay links—three in this case: client-to-client agent, client agent-to-server agent, and server agent-to-server are established in parallel. All

links are authenticated and checked to see if their intended peers are connected.

(7) Client and server libraries send acknowledgments to each other. Upon receiving an acknowledgment, each party knows the end-to-end channel was successfully established. At this time the client library notifies the client application that the connection has been established by returning from a blocking `XRAY_connect` call or by returning the corresponding file descriptor as write ready for `XRAY_select` for a non-blocking call. The library on the server side queues the connection so that it may be returned when the application calls `XRAY_accept`.

XRAY uses various techniques from our previous systems [10] [16]. For example, if an agent is running on a middlebox machine, it uses CODO techniques to reserve addresses, if necessary, and dynamically create pinholes for the relay points. If an agent is deployed outside of the server's network and not allowed to make a connection to the server, it instead uses the GCB technique to let the server make a connection to the relay point.

The end-to-end acknowledgement in step (7) may seem unnecessary because the application can send data as soon as the overlay link is attached to the next hop. If one or more intermediate links have not been established yet, the data can be buffered at the relay points and may be pushed later. We did not take this approach in hopes to reduce application's frustration. Connection failures generally involve human errors such as mistyping the address or bad network configurations, 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 for transfer errors. If XRAY reports successful connection with some intermediate links not finished yet, applications will see more data transfer errors not necessarily because of network errors.

Connection establishment within a private network also needs help from XRAY agents. A client within the same private network as a server cannot make a direct connection with the (leased) official address of the server. In this case, the XRAY agent of the private network replies to client's connection request with the server's local address so that the client can make a direct connection to the server. No relay points are created for intra network connections.

5.3. Data communication

The result of the connection setup process is a communication channel between a client and a server composed of one or more overlay links connected together via relay points. Because data are encrypted and decrypted by each hop, relay points have the full access to the contents, while still providing appropriate level of end-to-end protection and secrecy to applications.

Each relay point can have site-specific and application-specific plug-ins attached to it. These plug-ins form a chain to be executed. Data arriving at a relay point are decrypted, checked by each plug-in in order, encrypted using the secret key for the next link, and forwarded to the next hop.

XRAY provides the application transparency for underlying mechanisms not only for connection setup but also for data communication. We use block ciphers to secure each overlay link. It is not trivial to provide the stream-based semantics of TCP over record-based communication of block ciphers. For example, `select` must not return *read ready* when the network buffer has a partial record that cannot be decrypted. Instead, the application should be informed that it can read something from the network only when full records have been received and successfully decrypted. Similarly, `select` should not return *write ready* when the network buffer has small space that can hold only a partial record. To provide the application TCP's stream semantics, XRAY has a buffering mechanism that translates stream (of clear text) to record based communication (of cipher text), and vice versa.

6. Fault tolerance

Successful connection depends on the reliability of XRAY agents. Nevertheless, applications should continue to work with a limited ability in the event of agent failure. For example, when an agent is down, connections that do not require the service from the agent should continue working.

When an agent is down, XRAY tries to provide as much service as possible. The local bound and half bound status explained in §5.1 enables server sockets to continue to be able to accept intra network connections. If a client's XRAY library cannot contact an agent, it tries a direct connection to the server as if the agent were not needed.

If an agent recovers from its failure, sockets that were affected by the failure should become fully functional. To achieve this goal, we just need to upgrade half bound sockets to fully bound status so that they can accept connections from outside. The XRAY library periodically tries to contact the failed local agent. If successful, it registers the information of sockets to the agent and upgrades them to fully bound status.

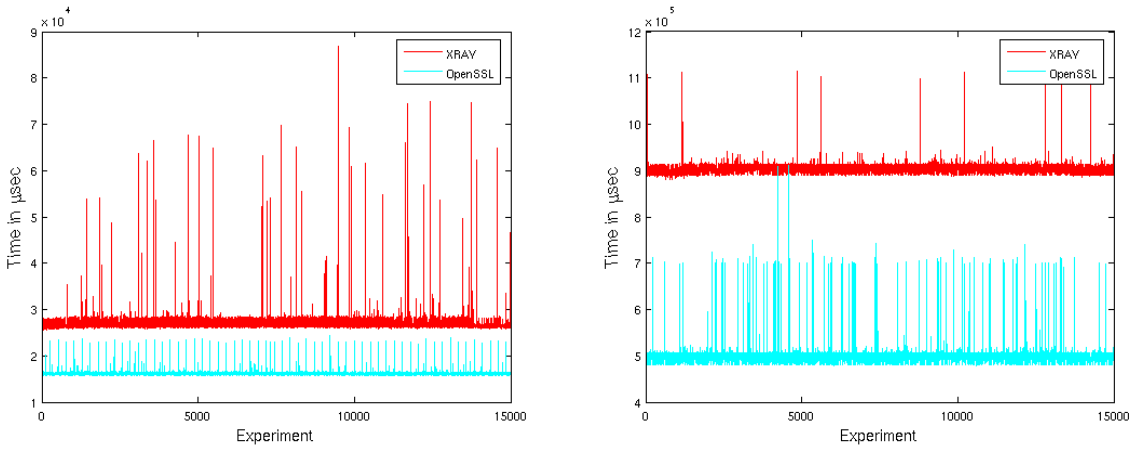

Figure 4: XRAY performance (a) connection time (b) data transfer time. *X-axis represents each experiment and Y-axis shows times in micro second for each experiment. The mean connection time of XRAY is 26,864 with the standard deviation of 1,832, while the average for OpenSSL is 16,166 with the standard deviation of 484. For data transfer, the mean time of XRAY is 903,374 with the standard deviation of 8,203, while the mean time for OpenSSL is 498,273 with the standard deviation of 17,440.*

7. Implementation

The XRAY library is implemented in C/C++ as a layer between the application and the kernel. Applications use XRAY socket calls to create an XRAY socket, bind it to an address, connect to a server, etc. In addition to socket calls, the library provides some file system calls so that applications may duplicate socket descriptors, make a socket non-blocking, and multiplex multiple file descriptors, including XRAY sockets. It also has a few functions for process control, such as `fork` and `execve`. These are mainly for inheriting open sockets to child processes. All XRAY calls have the same APIs as their regular counterparts. This strategy is intended to facilitate application programming and enable dynamically linked applications to use XRAY without recompilation.

8. Performance measurement

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. Neither inbound nor outbound connections are allowed in the 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 connection setup and data transfer times. In our test suite, a client 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. We inserted random delays between connections. Actual delay was determined using a Poisson process with a mean (λ) of 3 seconds. We used X.509 (RSA) public key for authentication and session keys establishment. SHA-1 and 3DES were used for integrity and encryption, respectively, of XRAY commands and application data. In order to understand the overhead of XRAY, we did the same experiments with OpenSSL [7] with NATs manually configured to allow traffic between two networks. For fair comparison, we configured OpenSSL to use the same mechanisms for authentication, encryption/decryption, and integrity. Since XRAY provides mutual authentication, we also configured OpenSSL clients to authenticate servers.

Figure 4 (a) shows the results for connection setup. XRAY connections take 27 msec on average, which is 1.67 times slower than regular OpenSSL connections. For each XRAY connection, five secure TCP connections are made in this experiment: two for XRAY command exchanges between the client 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, the client agent, the server agent, and the server. Considering the number of connections and interactions in XRAY, connection times are surprisingly short. We determine that two

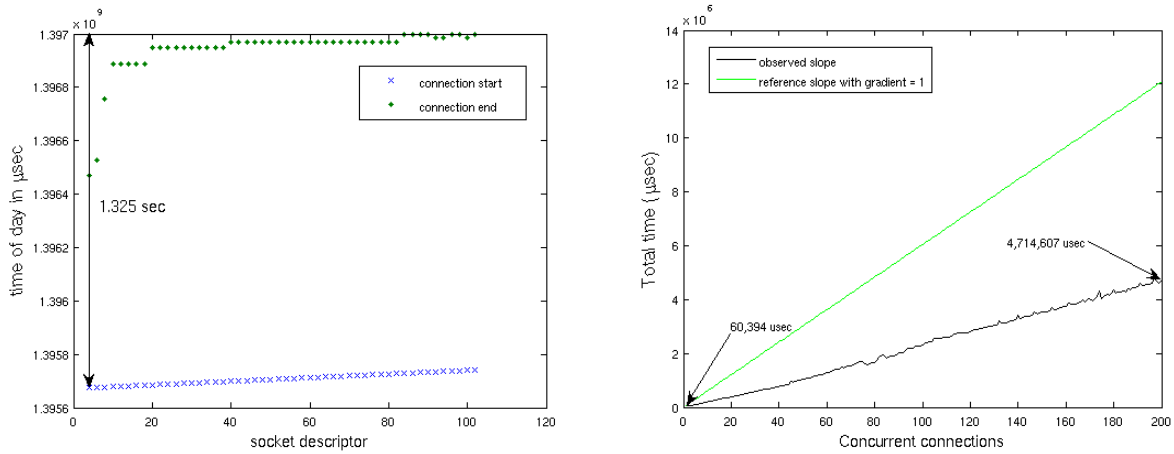

Figure 5: Concurrent connection setup. (a) *The X-axis represents socket descriptors that non-blocking connections were issued with. The Y-axis shows the times of each connection issued ('x' mark) and finished (dot). The time difference between the first connection issued and the last connection finished is 1.325 seconds.* **(b)** *The X-axis shows the number of concurrent connections issued. The Y-axis shows the total time to set up multiple connections.*

factors help XRAY's connection performance. First, the parallel connection setup of end-to-end channel (i.e. three connections in this case) reduces the overall connection time. Second, XRAY uses the session resumption [8] to avoid the expensive public key mechanism. XRAY entities cache and reuse security sessions to communicate with others they have recently talked with. Since XRAY agents are commonly contacted entities, session reuse is often possible. Also, note that all connections for end-to-end channel can be established without using a public key mechanism because all entities must have talked with each other for exchanging XRAY commands. Figure 4 (b) shows the data transfer results. The figure shows that XRAY data transfer is 1.81 times slower than the direct OpenSSL communication. We believe that this overhead is reasonable considering the relaying and encryption/decryption operations by two agents.

To see how XRAY scales, we also tested concurrent connection setup. Figure 5 (a) shows that multiple connections can be established concurrently rather than in a serial manner. In this test, a client in a private network issued 50 concurrent connections (i.e. non-blocking connections) to a server in another private network and recorded the times of connection issued and finished for each socket. The figure shows that multiple connections are established at the same time rather than one-by-one. Some connections (e.g. 45th, 46th, and 49th) finished earlier than those that started earlier. Figure 5 (a) also shows that all 50 connections were finished within 1.325 seconds, which is only 29 times as slow as a single connection setup using the same test program, instead of an expected 50 times for serial establishment. Figure 5 (b) shows how the total time varies as the number of concurrent connections increase. In this test, a client simultaneously issued up to 200 non-blocking connections to a single server in a different private network. 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. The result shows that it took about 4.7 seconds to establish 200 connections, which is only 78 times as slow as a single connection setup. Those results show that both the XRAY library and agent can handle many concurrent connections very well and that XRAY mitigates performance overheads by interleaving operations.

9. Related work

Many middlebox traversal systems have been proposed or developed. Unlike XRAY and our previous work CODO (Cooperative On-Demand Opening) [16], previous research mainly focuses on how to enable application traversal of middleboxes, with little attention to the security of the network. CODO dynamically adds and removes owner test rules for authorized applications. CODO has many characteristics in common with XRAY. CODO helps applications communicate with the world as well as

helps middleboxes to perform quality owner test; it controls both inbound and outbound traffic; it uses strong security mechanisms to protect unauthorized applications from having owner test rules created for them. CODO is more efficient than XRAY because with CODO applications communicate directly through holes made at middleboxes. However, the owner test CODO constructs is less secure than the owner test XRAY constructs. With CODO, attackers can cause owner test false positives using address spoofing, which is impossible with XRAY. CODO also has an additional requirement that middleboxes provide an API for dynamic control. Therefore, XRAY supports more organizations and provides more secure traversal while CODO provides secure and efficient traversal.

SOCKS [14] is also similar to XRAY. It enables communications through a middlebox by a proxy that relays connections. The proxy is application-independent and can be configured to use strong security mechanisms to authenticate applications. However, SOCKS does not have the concept of address leasing to server sockets (§5.1) and generally does not support private networks. SOCKS is also designed only for client-server applications and cannot support applications such as P2P and the Grid. In SOCKS, each application must act as a client or a server, but not both. A client application may accept connections over middleboxes. However, these passive connections must be secondary connections and a part of an active session that is initiated by an active primary connection. FTP is a good example. A SOCKS-enabled FPT client can establish an active connection (the control channel) to an FTP server behind a middlebox and then accept a passive connection (the data channel) from the same server. However, with SOCKS, a client application cannot have an independent passive socket to accept connections from arbitrary endpoints.

Other middlebox traversal mechanisms both middleware and fundamental approaches are reviewed in [10] and [16].

10. Conclusion

In this paper, we discussed middlebox traversal problem in a broader and formalized context of network security and presented a relay-based middlebox traversal system, called XRAY. XRAY is a middlebox-friendly system, which helps not only applications to communicate with the world but also middleboxes to better filter traffic. XRAY controls both inbound and outbound traffic in a secure manner. Middleboxes can achieve a practically error-free owner binding and owner test. Additionally, the conflict of interests between applications and middleboxes can be appropriately addressed with XRAY. Our experiments also showed that XRAY provides such benefits at reasonable performance overheads.

References

[1] K. Egevang, P. Francis, "The IP Network Address Translator (NAT)," *IETF RFC1631* May 1994.

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

[3] M. S. Blumenthal and D. 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.

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

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

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

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

[8] T. Dierks, C. Allen, "The TLS Protocol," *IETF RFC 2246*, Jan. 1999.

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

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

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

[12] J. Rosenberg, R. Mahy, C. Huitema, "Traversal Using Relay NAT (TURN)," *Internet-Draft*, July 2004.

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

[14] M. Leech, M.Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones, "SOCKS Protocol Version 5," *IETF RFC 1928*, March 1996.

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

[16] S. Son, B. Allcock, M. Livny, "CODO: Firewall Traversal by Cooperative On-Demand Opening," to appear at 14th HPDC, Research Triangle Park, NC, July, 2005.

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