Protocol Scrubbing: Network Security Through Transparent Flow Modification

David Watson, Matthew Smart, G. Robert Malan, *Member, IEEE*, and Farnam Jahanian, *Member, IEEE*

***Abstract*—This paper describes the design and implementation of *protocol scrubbers*. Protocol scrubbers are transparent, interposed mechanisms for explicitly removing network scans and attacks at various protocol layers. The *transport scrubber* supports downstream passive network-based intrusion detection systems by converting *ambiguous* network flows into *well-behaved* flows that are unequivocally interpreted by all downstream endpoints. The *fingerprint scrubber* restricts an attacker's ability to determine the operating system of a protected host. As an example, this paper presents the implementation of a TCP scrubber that eliminates *insertion* and *evasion* attacks—attacks that use ambiguities to subvert detection—on passive network-based intrusion detection systems, while preserving high performance. The TCP scrubber is based on a novel, simplified state machine that performs in a fast and scalable manner. The fingerprint scrubber is built upon the TCP scrubber and removes additional ambiguities from flows that can reveal implementation-specific details about a host's operating system.**

***Index Terms*—Intrusion detection, network security, protocol scrubber, stack fingerprinting.**

I. Introduction

As SOCIETY grows increasingly dependent on the Internet for commerce, banking, and mission-critical applications, the ability to detect and neutralize network attacks is becoming increasingly significant. Attackers can use ambiguities in network protocol specifications to deceive network security systems. Passive entities can only notify administrators or active mechanisms after attacks are detected. However, the response to this notification may not be timely enough to withstand some types of attacks—such as attacks on infrastructure control protocols. Active modification of flows is the only way to immediately detect or prevent these attacks. This paper presents the design and implementation of *protocol scrubbers*—transparent, interposed mechanisms for actively removing network attacks at various protocol layers. We describe two instances of protocol scrubbers in this paper. The *transport scrubber* addresses the problem of insertion and evasion attacks by removing protocol related ambiguities from network flows, enabling downstream passive network-based intrusion detection systems to operate with high assurance [1]. The *fingerprint scrubber* prevents a remote user from identifying the operating system of another host at the TCP/IP layers by actively homogenizing flows [2].

The transport scrubber's role is to convert *ambiguous* network flows—flows that may be interpreted differently at different endpoints—into *well-behaved* flows that are interpreted identically by all downstream endpoints. As an example, this paper presents the implementation of a TCP scrubber that eliminates *insertion* and *evasion* attacks against passive network-based intrusion detection systems. Insertion and evasion attacks use ambiguities in protocol specifications to subvert detection. This paper argues that passive network intrusion detection systems (NID systems) can only effectively identify malicious flows when used in conjunction with an interposed active mechanism. Through interposition, the transport scrubber can guarantee consistency, enabling downstream intrusion detection systems to work with confidence. The specifications for Internet protocols allow *well-behaved* implementations to exchange packets with deterministic results. However, sophisticated attackers can leverage subtle differences in protocol implementations to wedge attacks past the NID system's detection mechanism by purposefully creating ambiguous flows. In these attacks, the destination endpoint reconstructs a malicious interpretation, whereas the passive NID system's protocol stack interprets the flow of packets as a benign exchange. Examples of these sources of ambiguity are IP fragment reconstruction and the reassembly of overlapping out-of-order TCP byte sequences. The role of the transport scrubber is to pick one interpretation of the protocols and to convert incoming flows into a single representation that all endpoints will interpret identically. The transport scrubber's conversion of ambiguous network flows into consistent flows is analogous to that of network traffic shaping. Shapers modify traffic around the edges of a network to generate predictable utilization patterns within the network. Similarly, the transport scrubber intercepts packet flows at the edges of an interior network and modifies them in such a way that their security attributes are predictable.

Ambiguities in protocol specifications also allow attackers to determine a remote host's operating system. The process of determining the identity of a host's operating system by analyzing packets from that host is called TCP/IP stack fingerprinting. Fingerprinting scans are often preludes to further attacks, and therefore we built the fingerprint scrubber to block the majority of stack fingerprinting techniques in a general, fast, and transparent manner. Freely available tools (such as

Manuscript received February 8, 2001; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Calvert. This work was supported in part by a gift and equipment donation from Intel Corporation, and in part by a research grant from the Defense Advanced Research Projects Agency, monitored by the U.S. Air Force Research Laboratory under Grant F30602-99-1-0527.

D. Watson and F. Jahanian are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122 USA (e-mail: dwatson@umich.edu; farnam@umich.edu).

M. Smart and G. R. Malan are with Arbor Networks, Ann Arbor, MI 48104 USA (e-mail: smart@arbor.net; rmalan@arbor.net).

Digital Object Identifier 10.1109/TNET.2003.822645

nmap [3]) exist to scan TCP/IP stacks efficiently by quickly matching query results against a database of known operating systems. The reason this is called "fingerprinting" is therefore obvious; this process is similar to identifying an unknown person by taking his or her unique fingerprints and finding a match in a database of known fingerprints. A malicious use of fingerprinting techniques is to construct a database of IP addresses and corresponding operating systems for an entire network. When someone discovers a new exploit, the attacker can now target only those machines running the vulnerable operating system. This facilitates the systematic installation of malicious code, such as distributed denial of service tools, on many machines without detection. Current fingerprinting techniques provide fine-grained determination of an operating system. For example, nmap has knowledge of 15 different versions of Linux. Almost every system connected to the Internet is vulnerable to fingerprinting, including standard computers running the major operating systems, routers, switches, hubs, bridges, embedded systems, printers, firewalls, Web cameras, and even some game consoles. Many of these systems, such as routers, are important parts of the Internet infrastructure, and compromised infrastructure is a more serious problem than compromised end hosts. Therefore a general mechanism to protect any system is needed.

The main contributions of this work are:

- *Introduction of transport scrubbing*: The paper introduces the use of an active, interposed transport scrubber for the conversion of ambiguous network flows into well-behaved, unequivocally interpreted flows. We argue that the use of a transport scrubber is essential for correct operation of passive NID systems. The paper describes the use of transport scrubbers to eliminate insertion and evasion attacks on NID systems [4]. The concept of transport scrubbing can easily be merged with existing firewall technologies to provide the significant security benefits outlined in this paper.
- *Design and implementation of TCP scrubber*: The novel design and efficient implementation of the *half-duplex* TCP scrubber is presented. The current implementation of the TCP scrubber exists as a modified FreeBSD kernel [5]. This implementation is shown to scale with raw Unix-based Ethernet bridging. By keeping the design of the scrubber general, we plan to migrate the implementation to programmable networking hardware such as the Intel IXA architecture [6], [7].
- *Design and implementation of fingerprint scrubber*: Building upon the TCP scrubber, we present a tool to defeat TCP/IP stack fingerprinting. The fingerprint scrubber is transparently interposed between the Internet and the network under protection. We show that the tool blocks the majority of known stack fingerprinting techniques in a general, fast, and transparent manner.

The remainder of this paper is organized as follows. Section II places our work within the broader context of related work. Section III describes the design, implementation, and performance characteristics of our TCP transport scrubber. Section IV presents our tool for defeating TCP/IP stack fingerprinting. Finally, Section V presents our conclusions and plans for future work.

II. Related Work

Current network security depends on three main components: firewalls, intrusion detection systems, and encryption. Firewalls are designed to restrict the information flowing into and out of a network. Intrusion detection systems attempt to monitor either the network or a single host and detect attacks. Encryption of data over a network attempts to transport data securely through an untrusted network. While these technologies attempt to provide a comprehensive security solution, performance considerations limit their effectiveness. Protocol scrubbers complement these technologies by filling in the gaps without adversely affecting performance.

Firewalls [8] and protocol scrubbers are both active, interposed mechanisms—packets must physically travel through them in order to continue toward their destinations—and both operate at the ingress points of a network. Modern firewalls primarily act as gate-keepers, utilizing filtering techniques that range from simple header-based examination to sophisticated authentication schemes. In contrast, the protocol scrubber's primary function is to homogenize network flows, identifying and removing attacks in real time.

Older firewalls that utilize application-level proxies, such as the TIS Firewall Toolkit [9], provide similar functionality to protocol scrubbers. These types of firewalls provide the most security, but their performance characteristics are not acceptable for deployment in high-speed environments. Their utility has decreased as the Internet has evolved. In contrast, the protocol scrubbers have been designed to achieve maximum throughput as well as a high level of security.

Firewall technologies changed with the advent of so-called *stateful inspection* of networking flows, exemplified by Checkpoint's Firewall-1 [10]. These types of firewalls examine portions of the packet header and data payloads to determine whether or not entry should be granted. After the initial check, a flow record is stored in a table so that fast routing of the subsequent network packets can occur. These later packets are not checked for malicious content. The protocol scrubbers differ in that they continue to remove malicious content for the lifetime of the flow.

In an attempt to combine the best of stateful inspection and application level proxies, Network Associates introduced a new version of their Gauntlet firewall [11]. The approach taken in this firewall is a combination of application-level proxy and fast-path flow caching. At the beginning of a flow's lifetime, the flow is intercepted by an application-level proxy. Once this proxy authenticates the flow, it is cached in a lookup table for fast-path routing. Again, the protocol scrubber differs by allowing detection of malicious content, not only at the beginning, but throughout the flow's lifetime.

Protocol scrubbers can also complement the operation of intrusion detection systems (ID systems) [12], [13]. There are two broad categories of intrusion detection systems: network-

based and host-based. Network-based intrusion detection systems (NID systems) are implemented as passive network monitors that reconstruct networking flows and monitor protocol events through eavesdropping techniques [14]–[17]. As passive observers, NID systems have a vantage point problem [18] when reconstructing the semantics of passing network flows. This vulnerability can be exploited by sophisticated network attacks that leverage differences between the processing of packets by the destination host and by an intermediary observer [4]. As an active participant in a flow's behavior, the protocol scrubber removes these attacks and can function as a fail-closed, real-time NID system that can sever or modify malicious flows.

Protocol scrubbers deal with virtual private networks (VPN) and header and payload encryption [19] in the same manner as NID systems. There are two approaches to filtering encrypted flows: the first assumes that the flow is sanctioned if it is end-to-end encrypted; an alternative approach is to filter any flows that do not match a preconfigured list. As an active mechanism, the protocol scrubber can remove unsanctioned flows in real time. When placed on the inside of a VPN, the protocol scrubber can be used to further clean packet streams. This would apply to scrubbing e-commerce transactions and sensitive database accesses.

While fingerprinting scans do not pose a direct threat to security, they are often preludes to further attacks. A NID system will be able to detect and log such scans, but the fingerprint scrubber actively removes them. Various tools are available to secure a single machine against operating system fingerprinting. The TCP/IP traffic logger `iplog` [20] detects fingerprint scans and sends out a packet designed to confuse the results. Other tools and operating system modifications simply use the kernel TCP state to drop certain scan types. None of these tools, however, can be used to protect an entire network of heterogeneous systems.

Developed simultaneously with our work, the concept of *traffic normalization* uses the same basic framework to solve attacks against NID systems [21], [22]. While we have focused on extending the idea of protocol scrubbers into other areas, such as preventing fingerprinting, Paxson, Handley, and Kreibich have attempted to develop a systematic method for identifying all possible sources of ambiguity in the TCP/IP suite of protocols. In addition, they have attempted to minimize the impact of attacks against the normalizer by minimizing the state kept for each connection, and actively adapting in real time. Their novel approach to dealing with packets without any associated state not only protects the normalizer from attacks, but also allows it to intelligently deal with preexisting connections.

Although protocol scrubbing has similarities with current network security mechanisms, we view the idea as a necessary next step in providing increased network security. Ambiguities in protocol specifications and differences in protocol implementations are likely to always exist. Protocol scrubbing provides a simple solution that could easily be incorporated into the next generation of firewall design. Not only would these scrubbing firewalls provide increased protection to end hosts, they would also enable NID systems to accurately detect attacks that make it past the firewall.

III. Transport Scrubber

Network-based intrusion detection systems are based on the idea that packets observed on a network can be used to predict the behavior of the intended end host. While this idea holds for well-behaved network flows, it fails to account for easily created ambiguities that can render the NID system useless. Attackers can use the disparity between reconstruction at the end host and the passive NID system to attack the end host without detection. The TCP scrubber is an active mechanism that explicitly removes ambiguities from external network flows, enabling downstream NID systems to correctly predict the end host response to these flows. Utilizing a novel protocol-based approach in conjunction with an in-kernel implementation, the TCP scrubber provides high performance as well as enforcement of flow consistency. By keeping a significantly smaller amount of state than existing solutions, the scrubber is also able to scale to tens of thousands of concurrent connections with throughput performance that is comparable to commercial stateful inspection firewalls and raw Unix-based IP forwarding routers. This section describes the overall design and implementation of the TCP scrubber and provides a comprehensive performance profile using both macro and microbenchmarks.

A. TCP Ambiguities and ID Evasion

Sophisticated attacks can utilize differences in the processing of packets between a network intrusion detection system and an end host to slip past the watching NID system completely undetected. NID systems rely on their ability to correctly predict the effect of observed packets on an end host system in order to be useful. In their paper, Ptacek and Newsham describe a class of attacks that leave NID systems open to subversion [4]. We borrow their description of the two main categories of these attacks: *insertion attacks*, where the NID system accepts a packet that the end host rejects, and *evasion attacks*, where the NID system rejects a packet that the end host accepts.

Fig. 1 provides a simple example of how differences in the reconstruction of a TCP stream can result in two different interpretations, one benign and the other malicious. In this simple example an attacker is trying to log into an end host as `root`, while fooling the NID system into thinking that it is connecting as a regular user. The attacker takes advantage of the fact that the end host and the NID system reconstruct overlapping TCP sequences differently. In Fig. 1(a), the attacker sends a data sequence to the end host with a hole at the beginning (represented by the question mark). Since TCP is a reliable bytestream service that delivers data to the application layer in order, both the end host and NID system must wait until that hole is filled before proceeding [23]. However, unbeknownst to the NID system—but not the wily attacker—the end host deals with overlapping sequences of bytes differently than the NID system. In Fig. 1(b), when the attacker resends the data with the hole filled, but with a different username of the same length, the difference in implementation choice between the two systems allows the attacker to dupe the NID system. Since a correct TCP implementation would always send the same data upon retransmission, it is not mandated in the specification as to which set of bytes the endpoint should keep. In this example, the end host

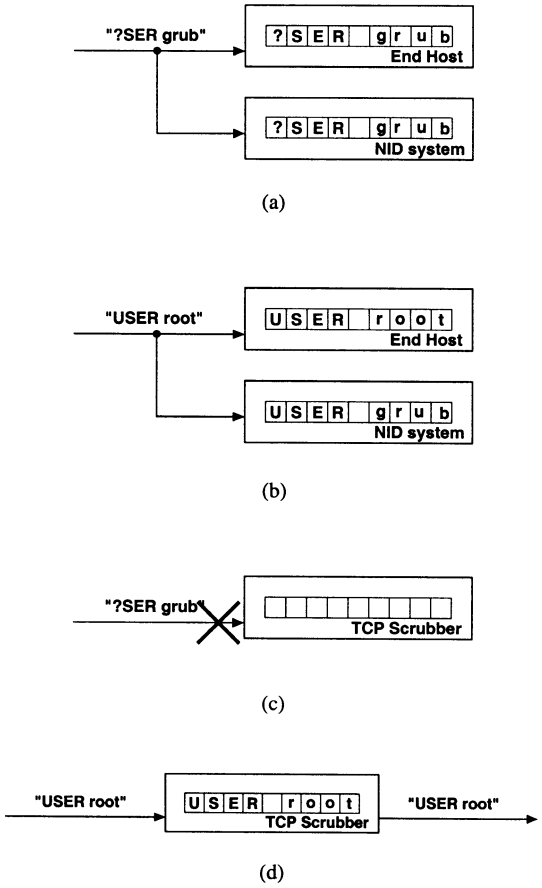

Fig. 1. Example of ambiguity of transport layer protocol implementation differences between an interposed agent (NID system) and an end host. (a) Host and NID system after attacker sends a hole. (b) Attacker filling in the hole and confusing the NID system. (c) Scrubber enforces single interpretation. (d) Attacker filling in the hole and sending new data.

chose to keep the new sequence of bytes that came in the second packet, whereas the NID system kept the first sequence of bytes. Neither is more correct than the other; just the fact that there is ambiguity in the protocol specification allows sophisticated attacks to succeed.

While these attacks do not increase the vulnerability of the end host, they significantly deteriorate the NID system's ability to detect attacks. This can lead to significant, undetected security breaches. While there is no evidence that these techniques are being used in the wild, simple toolkits such as `fragroute` [24] make the task easy. In addition, there is ample evidence that NID systems see ambiguous network flows from benign TCP/IP stacks [16]. This makes it increasingly difficult for the NID system to detect real attacks.

To address this problem, we have created the TCP scrubber. Specifically, the scrubber provides the uniformity that NID systems need for confident flow reconstruction and end host behavior prediction. For example, the scrubber stores in-sequence, unacknowledged data from the TCP sequence space. When any unacknowledged data is retransmitted, the original data is copied over the data in the packet to prevent possible ambiguity. When acknowledged, this stored data is thrown away and is removed from any subsequent packets. To prevent attacks against the scrubber, out-of-sequence data is thrown away and not retransmitted to the end host. Specifically, Fig. 1(c) and (d) demonstrates how the active protocol scrubber interposed between the attacker and the downstream systems eliminates the ambiguity. By picking a single way to resolve the TCP reconstruction both the downstream NID system and end host both see the attacker logging in as `root`. In this case the scrubber simply throws away the data after a hole.

In addition to the handling of overlapping TCP segments, there are many other ambiguities in the TCP/IP specification that produce different implementations [4]. To begin with, the handling of IP fragments and their reconstruction varies by implementation. Similar variations are seen with the reconstruction of TCP streams. End hosts deal differently with respect to IP options and malformed headers. They vary in their response to relatively new TCP header options such as PAWS [23]. Moreover, there are vantage point problems that passive NID systems encounter such as TTL-based routing attacks and TCP creation and tear-down issues. The large number of ambiguities with their exponential permutations of possible end host reconstructions make it impractical for NID systems to model all possible interpretations at the end host. They must pick some subset, generally a single interpretation, to evaluate in real time. For this reason it is impractical to adequately address the problem within the context of a passive NID system.

B. TCP Scrubber Design and Implementation

The TCP scrubber converts external network flows—sequences of network packets that may be interpreted differently by different end host networking stacks—into homogenized flows that have unequivocal interpretations, thereby removing TCP insertion and evasion attacks. While TCP/IP implementations vary significantly in many respects, correct implementations interpret well-behaved flows in the same manner. However, flows that are not well-behaved are often interpreted differently even by correct implementations. The protocol scrubber's job is to codify what constitutes well-behaved protocol behavior and to convert external network flows to this standard. To describe all aspects of a well-behaved TCP/IP protocol stack is impractical. However we will illustrate this approach by detailing its application to the TCP byte stream reassembly process. TCP reassembly is the most difficult aspect of the TCP/IP stack and is crucial to the correct operation of NID systems. Note, however, that we address more ambiguities of the TCP/IP specification when we discuss the fingerprint scrubber in Section IV.

The TCP scrubber's approach to converting ambiguous TCP streams into unequivocal, well-behaved flows lies in the middle of a wide spectrum of solutions. This spectrum contains stateless filters at one end and full transport-level proxies—with a considerable amount of state—at the other. Stateless filters can handle simple ambiguities such as nonstandard usage of TCP/IP header fields with little overhead. However, they are incapable of converting a stateful protocol, such as TCP, into a nonambiguous stream. Full transport-layer proxies lie at the other end of the spectrum, and can convert all ambiguities into a single well-behaved flow. However, the cost of constructing and maintaining two full TCP state machines—scheduling timer events,

estimating round-trip time, calculating window sizes, etc.—for each network flow restricts performance and scalability. The TCP scrubber's approach to converting ambiguous TCP streams into well-behaved flows attempts to balance the performance of stateless solutions with the security of a full transport-layer proxy. Specifically, the TCP scrubber maintains a small amount of state for each connection but leaves the bulk of the TCP processing and state maintenance to the end hosts. Moreover, the TCP scrubber only maintains state for the half of the TCP connection originating at the external source. Even for flows originating within a protected network there is generally a clear notion of which endpoints are more sensitive and need protection; if bidirectional scrubbing is required, the scrubber can be configured to provide it. With this compromise between a stateless and stateful design, the TCP scrubber removes ambiguities in TCP stream reassembly with performance comparable to stateless approaches.

To illustrate the design of the TCP scrubber, we compare it to a full transport-layer proxy. TIS Firewall Toolkit's `plug-gw` proxy is one example of a transport proxy [9]. It is a user-level application that listens to a service port waiting for connections. When a new connection from a client is established, a second connection is created from the proxy to the server. The transport proxy's only role is to blindly read and copy data from one connection to the other. In this manner, the transport proxy has fully obscured any ambiguities an attacker may have inserted into their data stream by forcing a single interpretation of the byte stream. This unequivocal interpretation of the byte stream is sent downstream to the server and accompanying network ID systems for reconstruction. However, this approach has serious costs associated with providing TCP processing for both sets of connections.

Unlike a transport layer proxy, the TCP scrubber leaves the bulk of the TCP processing to the end points. For example, it does not generate retransmissions, perform round-trip time estimation, or perform any timer-based processing; everything is driven by events generated by the end hosts. The TCP scrubber performs two main tasks: it maintains the current state of the connection and keeps a copy of the byte stream that has been sent by the external host but not acknowledged by the internal receiver. In this way it can make sure that the byte stream seen downstream is always consistent—it modifies or drops any packets that could lead to inconsistencies. Fig. 2 graphically represents the reduced TCP state processing that occurs at the TCP scrubber. This simple combined bidirectional state machine allows for high scalability by leaving the complex protocol processing to the end points. The scrubber has only three general states: connection establishment (*INIT* and *INIT*2 states), established operation (*EST AB*), and connection termination (*CLOSE* and *CLOSED*), whereas the endpoint TCP stack has much more complex state machines that include states such as fast retransmit and slow start.

The TCP scrubber scales significantly better than a full transport proxy because the amount of state kept by the scrubber is much less than that kept by a transport proxy. TCP is a reliable byte stream service, therefore a sender must keep a copy of any data it has sent until it receives a message from the receiver acknowledging its receipt. Fig. 3 illustrates a data

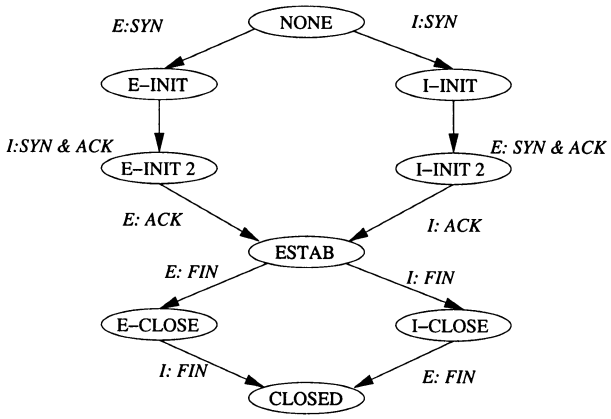

Fig. 2. TCP scrubber's state transition diagram for a single connection.

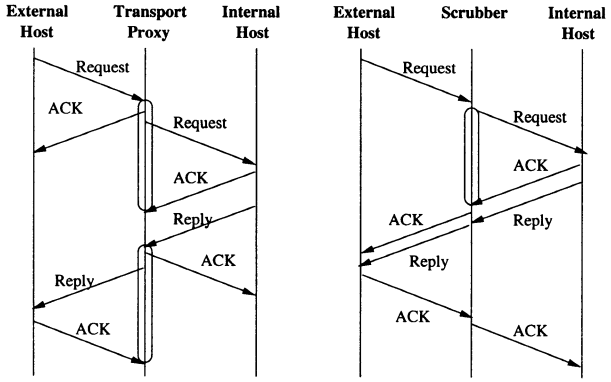

Fig. 3. Example of TCP data transfer messages between endpoints and interposed mechanism.

transfer operation from an external client to an internal service using TCP. The circled portions at the center timeline represent the amount of time that data from either the client or server is buffered at the transport proxy or scrubber. Notice that both the scrubber and the transport proxy must buffer the incoming external request until its receipt is acknowledged by the internal server. However, the server's reply is not modified or buffered by the TCP scrubber, whereas the transport proxy must buffer the outbound reply until it is acknowledged. This is a somewhat subtle point; the outbound reply will generally be held for much longer than the incoming request by an interposed mechanism. This is because the *distance*—measured as round-trip time and packet losses—from the scrubber to the server will be short relative to the long distance to an external client. It is fair to assume that the scrubber and services it protects are collocated on a fast enterprise network while the scrubber and external client are separated by a wide area network with widely varying loss and latency characteristics. The TCP scrubber's approach to homogenization of TCP flows improves scalability in the number of simultaneous connections it can service.

In addition to a novel protocol processing design, the TCP scrubber's in-kernel implementation provides for even greater performance advantages over a user-space transport proxy. Currently, the TCP scrubber is implemented within the FreeBSD 4.5 kernel's networking stack, which is a derivative of the BSD 4.4 code [25].

C. TCP Scrubber Security

The goal of the TCP scrubber is to improve the overall security of the protected network without significantly impacting the function or performance of the network. For this reason, it is important to consider the security of the scrubber itself.

First, the TCP scrubber is designed to be fail safe; any attacks that disable the scrubber will make the protected network unreachable. This prevents attackers from disabling the scrubber and then attempting to evade detection. Unfortunately, this feature creates a denial of service vulnerability—any successful attack against the scrubber will shut down the protected network. Standard techniques such as watchdogs would limit the severity of such attacks.

Another denial of service consideration of the TCP scrubber is the problem of preexisting connections. When the scrubber is first started it knows nothing about the state of existing connections. The current design forces all connections to go through the three-way handshake before data packets are allowed through. This means that any established connections will be shut down when the scrubber is initially added to the network or restarted. While this conservative approach prevents an attacker from gaining access to a protected host, it creates a severe denial of service vulnerability. One simple approach to this problem, as described in [21], would be to use the asymmetric design of the scrubber to construct state for existing connections. Any packets originating from a trusted host imply that a connection has already been established. By using modified packets from untrusted hosts, the scrubber can determine if they are part of an already established connection without allowing potential attacks through. If the end host responds, the scrubber can set up state for the connection and proceed. If the end host sends a reset or fails to respond, the scrubber will continue to block packets from the untrusted host. This solution would not only prevent the scrubber from being used by attackers, but would also prevent benign restarts of the scrubber from disrupting the network.

By forcing the TCP scrubber to keep large amounts of unnecessary state, an attacker can exhaust the memory of the scrubber, forcing it to block new connections. The scrubber's asymmetric design reduces the amount of state by keeping track of TCP data for only one half of the connection. The amount of state kept by the scrubber can be further reduced by actively limiting the amount of TCP data kept for each connection. If the buffer for a particular connection is full, new packets will be dropped. This might slow down legitimate connections, but it will not cause any of them to be disconnected. Another solution involves monitoring the amount of memory available to the scrubber and intelligently dropping connections. Since most attempts to establish a large number of connections are easily distinguished from legitimate traffic, it is simple to drop these connections first.

Other possible attacks against the scrubber include resource-based attacks such as CPU overloading. We expect that attempts to improve the performance of the TCP scrubber are the best defense against these attacks. Current tests with the scrubber show that with relatively inexpensive hardware, the current scrubber design is not the bottleneck on even gigabit-speed networks. While this hardware will not scale to high-speed links used in large backbone networks, we expect it to be usable on any network supporting end hosts.

TABLE I
LATENCY OF TCP/IP FORWARDING AND TCP SCRUBBING (IN MICROSECONDS)

Forwarding Type	Mean	Std Dev
IP Forwarding	8.00	2.91
TCP Scrub (1 byte payload)	13.19	3.38
TCP Scrub (> 1000 byte payload)	31.85	5.72

D. TCP Scrubber Performance

This section presents the results from a series of experiments that profile the TCP scrubber's performance characteristics. They show that, in general, the current implementation of the TCP scrubber can match the performance of both commercial stateful inspection firewalls and raw Unix-based Ethernet bridging routers.

The first set of experiments attempted to compare the raw overhead of scrubbing packets to simple IP forwarding. These microbenchmarks measured the amount of time it took for a packet to complete the kernel's `ip_input` routine. For an IP forwarding kernel, the time spent in `ip_input` corresponds to the amount of time needed to do IP processing and forwarding, including queuing at the outbound link-level device (Ethernet). For the TCP scrubber it represents the time to scrub the packet and queue it on the outbound link-level device. Due to the difficulty of tracing packets through user space, we were not able to measure any proxying solutions. Table I shows the results from this experiment. These numbers were measured on older hardware (a 300-MHz Pentium II CPU), but we expect that these numbers accurately reflect the relative performance. IP forwarding requires a minimal amount of work for each packet independent of the packet size. As expected, it imposes a minimal amount of overhead. The scrubber, on the other hand, copies the data from the packet into an internal buffer. As we would expect, with small payloads the scrubber adds a small amount of overhead beyond forwarding. For larger payload packets, copying the TCP data into memory adds a significant amount of overhead. The results from these experiments would seem to indicate that with large payload packets, TCP scrubbing will perform poorly compared to IP forwarding. However, as further experiments will show, this portion of processing packets in the kernel is not the dominant factor determining actual throughput.

Initial attempts to measure the performance of the TCP scrubber found that on a traditional 100-Mb/s Fast Ethernet network, the bottleneck for all forwarding options was the network itself [1]. In order to get a better picture of the real performance of the scrubber, we performed a new series of experiments on a gigabit-speed network. These experiments attempted to determine the performance of the TCP scrubber using realistic tests. Fig. 4 shows the experimental configuration used in these experiments. The clients and servers were each connected to their own Intel Express Gigabit Switch. The clients and servers were all running FreeBSD 4.5, had 500-MHz Pentium III processors, 128-MB main memory, and used Intel PRO/1000

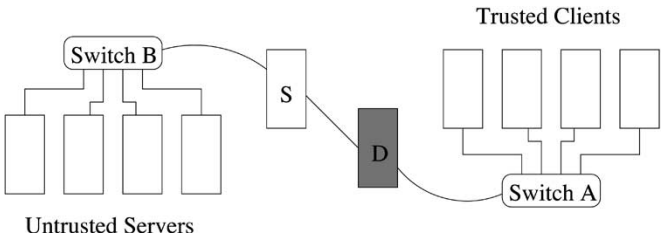

Fig. 4. Experimental apparatus for measuring the protocol scrubber's performance.

TABLE II
THROUGHPUT FOR A SINGLE EXTERNAL CONNECTION TO AN INTERNAL HOST (Mb/s, ±2.5% AT 99% CI)

Crossover	Bridge	TCP Scrubber
359.13	333.81	322.30

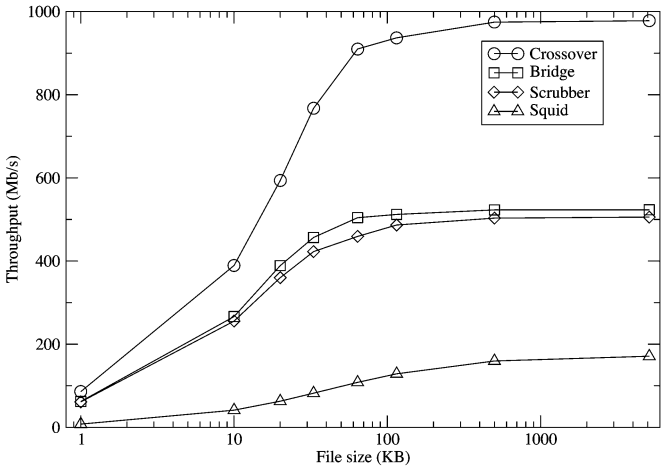

Fig. 5. Gigabit TCP scrubber performance results.

Gigabit Server Adapters (`em` driver). The two switches were then connected by different mechanisms: a patch cable, and a computer acting as an Ethernet bridge, Squid proxy, and TCP scrubber. The computer used to connect the two switches, S, was a 733-MHz computer, with 1.6 Gb of memory and two gigabit adapters. For one set of experiments, another computer, D, was added to simulate loss. It is important to note that these are all fiber connections, and that the maximum throughput of the system is 2 Gb/s, since everything is running in full-duplex mode. In addition, the maximum number of sockets and the maximum segment length on the servers and Squid proxy were modified to deal with the large number of connections left in the TIME_WAIT state. Specifically, the `kern.ipc.maxsockets` variable was set to 65 536 and the `net.inet.tcp.msl` variable was set to 1000, or 1 s.

As a baseline measurement, we used the Netperf benchmark [26] to measure the maximum TCP throughput for a single connection. Tests were run for 60 s using the TCP Stream test, between two and ten iterations, a confidence interval of 99% ±2.5%, a local send size of 32 768 bytes, and remote and local socket buffers of 57 344 bytes. Table II summarizes these results: a crossover cable provides the highest throughput, and the scrubber the worst. These numbers reflect the amount of processing power required by each of these techniques. However, the differences between the different methods is relatively small.

To better test the performance of the TCP scrubber, we ran a set of experiments designed to simulate fetching pages from a Web server. Each client machine was running a custom client designed to sequentially request the same Web page. Similarly, each server machine ran a custom single-threaded single-process Web server. In order to remove the performance impact of a traditional Web server the servers were designed to return a static Web page for every connection without parsing the HTTP request. While these servers are not realistic, they simulate the performance of a highly optimized Web server cluster. This best case server performance represents a worst case scenario for the TCP Scrubber. Another interesting point is that we made the servers untrusted, the reverse of what we described in previous sections. This was done to further stress the performance of the TCP scrubber. Since much more data is sent from the servers to the clients than *vice versa*, this scenario forces the scrubber to perform much more work sanitizing the data in the packets. Since the primary bottleneck in this system is the bandwidth of the half-duplex link from the servers to the clients, we only report this measurement. Another computer was connected to an out-of-band management port to measure the throughput of the inter-switch connection using the statistic counters maintained by the switch.

Fig. 5 shows the results of these experiments. The *x*-axis measures the size of the simulated Web pages. The *y*-axis measures the throughput of the data entering the client switch from the server switch. This is the half of the full-duplex connection that is carrying all the data from the servers. The top line in the graph represents the performance when the two switches are connected using a (crossover) patch cable. For large file sizes, we approach the absolute maximum of 1 Gb/s. At lower file sizes, the interrupt overhead of processing a large number of small packets limits the possible throughput. The next line down measures the impact of adding a computer into the link. The computer was connected to both switches and configured to act as an Ethernet bridge. This involves copying an incoming packet into the kernel, looking up the correct destination, and copying the packet to the correct outgoing interface. With the minimal processing involved we would expect this to represent the maximum throughput of any computer-based solution. The third line measures the performance of the scrubber. The scrubber appears to mirror the performance of the Ethernet bridge, with a relatively small performance hit. The fourth line represents the performance of a Squid proxy. Since we were attempting to compare the scrubber to a general purpose proxying solution, we configured Squid not to cache any pages. In addition, we turned off as much logging as possible, and redirected the rest of the logging to `/dev/null`.

The results of these experiments show that under the conditions that we tested, the TCP scrubber does not add a significant amount of overhead compared to other computer-based solutions. While we did not test any commercial firewalls, we expect that they would be unable to exceed the performance of the Ethernet bridge. For that reason, we feel that the TCP scrubber is very comparable to existing in-line security mechanisms. In addition, the common techniques that allow commercial firewalls to scale to high-speed networks, such as checksum

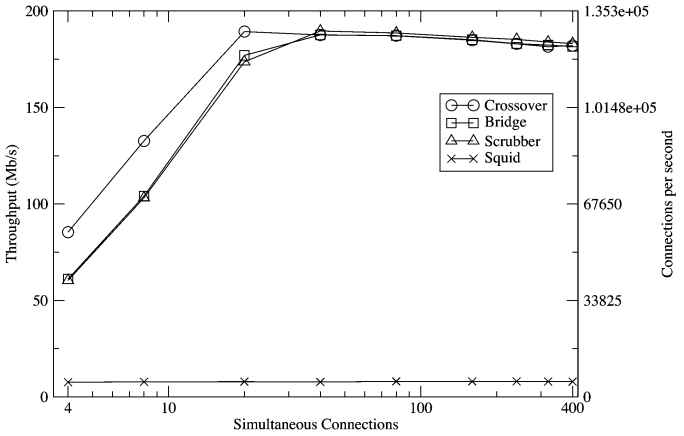

Fig. 6. TCP scrubber connections-per-second results.

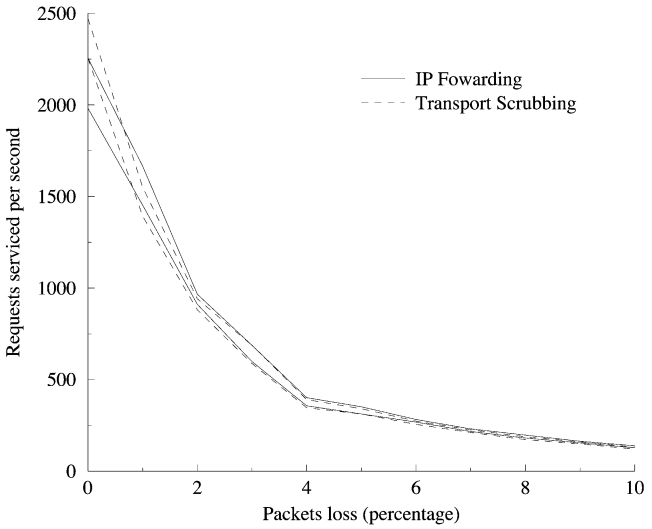

Fig. 7. TCP scrubber lossy results.

offloading, pooling, and zero-copy techniques, would equally benefit scrubbing. For environments where we needed to scrub links and achieve performance beyond the capabilities of PC hardware, we could implement the scrubber on specialized hardware. Intel's IXA architecture is one solution that provides easy maintainability while permitting processing on high-speed links [6], [7].

In addition to showing that the TCP scrubber does not add significant overhead to the throughput of TCP connections, we also wanted to show that the scrubber can handle a large number of small connections. Since setting up the state for each TCP connection is resource intensive, a large number of small connections tests this portion of the TCP scrubber code. The setup for this set of experiments was identical to the throughput tests, except we fix the file size to 1 K and vary the number of simultaneous connections. This was accomplished by running multiple clients on each client machine and multiple servers on each server machine. Fig. 6 shows the results of these experiments. The x-axis represents the number of simultaneous connections, which is the number of client processes on each client machine times four. The left y-axis shows the throughput of the half-duplex connections from the servers to the clients, as before. Since we were using all ten of our gigabit Ethernet cards, we were unable to directly measure the number of successful connections per second. Instead, the right y-axis uses the results of measuring the bandwidth consumed by a single connection to estimate the number of successful connections. As before, we measured the performance of a crossover cable, an Ethernet bridge, the TCP scrubber, and a Squid proxy. The most noticeable result is that the top three methods are essentially identical at 40 simultaneous connections and above. We were able to verify that the bottleneck in this situation was the processing capacity of the servers, something we were unable to remedy with our limited hardware. Beyond the server bottleneck, the results are very similar to the throughput tests. The Ethernet bridge and the TCP scrubber have similar performance, and the Squid proxy performs significantly worse. The latter result is not a surprise—we expect any user-level proxy to be slow. The multiple data copies and context switching will always doom any user-space implementation to significantly worse performance than the two in-kernel approaches [27], [28]. While we were unable to remove the server bottleneck from this experiment, the results show that the TCP scrubber can handle large numbers of very short connections.

Finally, we conducted a set of experiments to determine the effects of a lossy link between the external clients and the interposed machine. Because these experiments are not bandwidth limited, we performed them on a 100-Mb/s link. In these experiments, the number of Web clients was fixed at 480, while artificial packet loss was forced on each network flow by a `dummynet` router [29], labeled D in Fig. 4. The results of this experiment are shown in Fig. 7. The vertical axis represents the number of HTTP requests serviced per second; the horizontal axis represents the proportion of bidirectional packet loss randomly imposed by the dummynet router. The pairs of lines represent the 99% confidence intervals for the mean sustained connections per second. The most significant result from this experiment is that the TCP scrubbed flows behave comparably to the raw IP forwarded flows.

IV. Fingerprint Scrubber

We created a second protocol scrubber to remove TCP/IP fingerprinting scans. While fingerprinting does not pose an immediate threat, it is a precursor to further attacks. We based the fingerprint scrubber on the TCP scrubber to leverage the performance of the in-kernel implementation and the architecture of the TCP state reassembly. The scrubber removes further ambiguities from the TCP and IP protocol layers to defeat attempts at scanning a host for operating system fingerprints.

The most complete and widely used TCP/IP fingerprinting tool today is `nmap`. It uses a database of over 500 fingerprints to match TCP/IP stacks to a specific operating system or hardware platform. This database includes commercial operating systems, routers, switches, and many other systems. Any system that uses TCP/IP is potentially in the database, which is updated frequently. `Nmap` is free to download and is easy to use. For these reasons, we are going to restrict our talk of existing fingerprinting tools to `nmap`.

`Nmap` fingerprints a system in three steps. First, it performs a port scan to find a set of open and closed TCP and UDP ports.

Second, it generates specially formed packets, sends them to the remote host, and listens for responses. Third, it uses the results from the tests to find a matching entry in its database of fingerprints.

Nmap uses a set of nine tests to make its choice of operating system. A test consists of one or more packets and the responses received. Eight of nmap's tests are targeted at the TCP layer and one is targeted at the UDP layer. The TCP tests are the most important because TCP implementations vary significantly. Nmap looks at the ordering of TCP options, the pattern of initial sequence numbers, IP-level flags such as the *do not fragment* bit, the TCP flags such as RST, the advertised window size, as well as other elements that may differ depending on the implementation of the sender. For more details, including the specific options set in the test packets, refer to [3].

Fig. 8 is an example of the output of nmap when scanning our EECS department's Web server, http://www.eecs.umich.edu, and one of our department's printers. The TCP sequence prediction result comes from nmap's determination of how a host increments its initial sequence number for each TCP connection. Many commercial operating systems use a random, positive increment, but simpler systems tend to use fixed increments or increments based on the time between connection attempts.

While nmap does a good job of performing fine-grained fingerprinting, there are other methods for fingerprinting remote machines. For example, various timing-related scans could determine whether a host implements TCP Tahoe or TCP Reno by imitating packet loss and watching recovery behavior. We discuss this threat and potential solutions in Section IV-B4. Also, a persistent person could also use methods such as social engineering or application-level techniques to determine a host's operating system. Such techniques are outside the scope of this study.

In this section, we discuss the goals and intended use of the scrubber as well as its design and implementation. We demonstrate that the scrubber blocks known fingerprinting scans in a general, transparent manner. By transparent we mean that the active modification of flows is accomplished without requiring the fingerprint scrubber to have explicit knowledge about fingerprinting scans or end hosts' TCP/IP stack implementations. We also show that the performance is comparable to that of a standard IP forwarding gateway.

A. Goals and Intended Use of the Fingerprint Scrubber

The goal of the fingerprint scrubber is to block known stack fingerprinting techniques in a general, fast, and transparent manner. The tool should be general enough to block classes of scans, not just specific scans by known fingerprinting tools. The scrubber must not introduce much latency and must be able to handle many concurrent TCP connections. Also, the fingerprint scrubber must not cause any noticeable performance or behavioral differences in end hosts. For example, it is desirable to have a minimal effect on TCP's congestion control mechanisms by not delaying or dropping packets unnecessarily.

We intend for the fingerprint scrubber to be placed in front of a set of systems with only one connection to a larger network. We expect that a fingerprint scrubber would be most appropriately implemented in a gateway machine from a LAN of heterogenous systems (i.e., computers running a variety of operating systems, along with printers, switches) to a larger corporate or campus network. A logical place for such a system would be as part of an existing firewall. Another use would be to place a scrubber in front of the control connections of routers. Because packets traveling to and from a host must travel through the scrubber, the network under protection must be restricted to having one connection to the outside world.

```
TCP Sequence Prediction:
            Class=truly random
            Difficulty=9999999 (Good luck!)
Remote operating system guess:
            Linux 2.0.35-37
```

(a)

```
TCP Sequence Prediction:
            Class=trivial time dependency
            Difficulty=1 (Trivial joke)
Remote operating system guess:
            Xerox DocuPrint N40
```

(b)

Fig. 8. Output of an nmap scan against a Web server running Linux and a shared printer. (a) Web server running Linux. (b) Shared printer.

B. Fingerprint Scrubber Design and Implementation

To meet our goals, the fingerprint scrubber is based on the TCP scrubber described earlier and operates at the IP and TCP levels to cover a wide range of known and potential fingerprinting scans. The TCP scrubber provides quick and scalable reassembly of TCP flows and enforcement of the standard TCP three-way handshake (3WHS). Instead of using the TCP scrubber, we could have simply implemented a few techniques discussed in the following sections to defeat nmap. However, the goal of this work is to stay ahead of those developing fingerprinting tools. By making the scrubber operate generically for both IP and TCP, we feel we have raised the bar sufficiently high.

1) IP Scrubbing: In addition to the TCP ambiguities we discussed when talking about the TCP scrubber, there are IP-level ambiguities that facilitate fingerprinting. IP-level ambiguities arise mainly in IP header flags and fragment reassembly algorithms. We can easily modify the IP header flags to remove these ambiguities without restricting functionality. This involves little work in the scrubber, but does require adjustment of the header checksum. To defeat IP-level insertion and evasion attacks, we are forced to reassemble the fragments. This requires keeping state in the scrubber to store the waiting fragments. Once a completed IP datagram is formed, it may require additional processing to be re-fragmented when it leaves the scrubber.

The fingerprint scrubber normalizes IP type-of-service and fragment bits in all IP packets. Uncommon and generally unused combinations of TOS bits are removed. If these bits need to be used (e.g., an experimental modification to IP) an administrator could easily remove this functionality. All of the TCP/IP implementations we tested ignore the *reserved fragment* bit and reset it to zero if it is set, but we wanted to be safe so we mask it out explicitly. The *do not fragment* bit is reset if the MTU of the next link is large enough for the packet.

Modifying the *do not fragment* bit could break MTU discovery through the scrubber. One could argue that the reason one would put the fingerprint scrubber in place is to hide information about the systems behind it. This might include topology and bandwidth information. However, such a modification is controversial. We leave the decision on whether or not to clear the *do not fragment* bit up to an administrator by allowing the option to be turned off.

IP reassembly is desired for exactly the same reason TCP stream reassembly is done in the TCP scrubber—to prevent insertion and evasion attacks. The fragment reassembly code is a slightly modified version of the standard implementation in the FreeBSD 2.2.7 kernel. It keeps fragments on a set of doubly linked lists. It first calculates a hash to determine the corresponding list. A linear search is done over this list to find the correct IP datagram and the fragment's place within the datagram. Old data in the fragment queue is always chosen over new data, providing a consistent view of IP data at the downstream hosts.

2) ICMP Scrubbing: As with the IP layer, ICMP messages also contain distinctive characteristics that can be used for fingerprinting. In this section we describe the modifications the fingerprint scrubber makes to ICMP messages. We only modify ICMP messages returning from the trusted side back to the untrusted side because fingerprinting relies on ICMP responses and not requests. Specifically, we modify ICMP error messages and rate limit all outgoing ICMP messages.

ICMP error messages are specified to include at least the IP header plus 8 bytes of data from the packet that caused the error. According to [30], as many bytes as possible are allowed, up to a total length ICMP packet length of 576 bytes. However, `nmap` takes advantage of the fact that certain operating systems send different amounts of data. To counter this, we force all ICMP error messages coming from the trusted side to have data payloads of only 8 bytes by truncating larger data payloads. Alternatively, we could look inside of ICMP error messages to determine if IP tunneling is being used. If so, then we would allow more than 8 bytes. We will revisit ICMP when we discuss the larger problem of timing attacks in Section IV-B4.

3) TCP Scrubbing: Even though a significant number of fingerprinting attacks take place at the TCP level, the majority of them are removed by the TCP protocol scrubber. The TCP scrubber provides quick and scalable reassembly of flows and enforces the standard TCP 3WHS. This allows the fingerprint scrubber to block TCP scans that do not begin with a 3WHS. In fact, the first step in fingerprinting a system is typically to run a port scan to determine open and closed ports. Stealthy techniques for port scanning do not perform a 3WHS and are therefore blocked. While the TCP scrubber defeats a significant number of fingerprinting attempts, there are others that must be addressed by the fingerprint scrubber.

One such scan involves examining TCP options, a significant source of fingerprinting information. Different operating systems will return the same TCP options in different orders. Sometimes this order is enough to identify an operating system. We did not want to disallow certain options because some of them aid in the performance of TCP (i.e., SACK) yet are not widely deployed. Therefore, we restricted our modifications to reordering the options within the TCP header. We simply provide a canonical ordering of the TCP options known to us. Unknown options are included after all known options. The handling of unknown options and ordering can be configured by an administrator.

We also defeat attempts at predicting TCP sequence numbers by modifying the normal sequence number of new TCP connections. The fingerprint scrubber stores a random number when a new connection is initiated. Each TCP segment for the connection traveling from the trusted interface to the untrusted interface has its sequence number incremented by this value. Each segment for the connection traveling in the opposite direction has its acknowledgment number decremented by this value.

4) Timing Attacks: The scans we have discussed up to now have all been static query–response style probes. Another possible form of fingerprinting relies on timing responses. For example, the scanning could determine the difference between a TCP Tahoe and TCP Reno implementation by opening a TCP connection, simulating a packet loss, and watching the recovery behavior. The existence or lack of Reno's fast recovery mechanism would be apparent.

It would be very difficult to create a generic method for defeating timing-related scans, especially unknown scans. One approach would be to add a small, random delay to packets sent out the untrusted interface. The scrubber could even forward packets out of order. However, this approach would introduce an increased amount of queuing delay and probably degrade performance. While this might be fine when the scrubber is placed in front of a single host, the scrubber was designed to handle an entire network of hosts. In addition, these measures are not guaranteed to block scans. For example, even with small amounts of random delay, it would be relatively easy to determine if a TCP stack implements TCP Tahoe or TCP Reno based on simulated losses since a packet retransmitted after a retransmission timeout has a much larger delay than one retransmitted because of fast retransmit.

We implemented protection against one possible timing-related scan. Some operating systems implement ICMP rate limiting, but they do so at different rates, and some do not do any rate limiting. We added a parameter for ICMP rate limiting to the fingerprint scrubber to defeat such a scan. The scrubber records a timestamp when an ICMP message travels from the trusted interface to the untrusted interface. The timestamps are kept in a small hash table referenced by the combination of the source and destination IP addresses. Before an ICMP message is forwarded to the outgoing, untrusted interface, it is checked against the cached timestamp. The packet is dropped if a certain amount of time has not passed since the previous ICMP message was sent to that destination from the source specified in the cache.

Fig. 9 shows the fingerprint scrubber rate limiting ICMP echo requests and replies. In this instance, an untrusted host is sending ICMP echo requests once every 20 ms using the `-f` flag with `ping` (flooding). The scrubber allows the requests through unmodified since we are not trying to hide the identity of the untrusted host from the trusted host. As the ICMP echo replies come back, however, the fingerprint scrubber makes sure that only those replies that come at least 50 ms apart are forwarded. Since the requests are coming 20 ms apart, for every

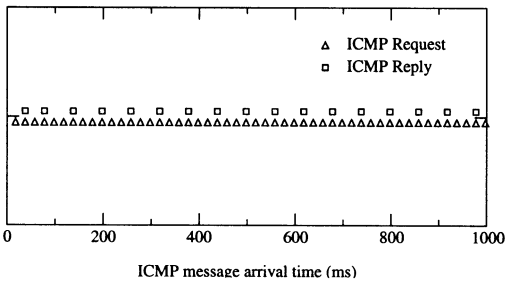

Fig. 9. ICMP rate limiting of returning ICMP echo replies captured using `tcpdump`.

three requests one reply will make it through the scrubber. Therefore, the untrusted host receives a reply once every 60 ms.

We chose 50 ms for convenience because `ping -f` generates a stream of ICMP echo requests 20 ms apart, and we wanted the rate limiting to be noticeable. The exact value for a production system would have to be determined by an administrator or based upon previous ICMP flood attack thresholds. The goal was to homogenize the rate of ICMP traffic traveling from the untrusted interface to the trusted interface because operating systems rate limit their ICMP messages at different rates. Another method for confusing a fingerprinter would be to add small, random delays to each ICMP message. Such an approach would require keeping less state. We can add delay to ICMP replies, as opposed to TCP segments, because they will not affect network performance.

C. Evaluation of Fingerprint Scrubber

This section presents results from a set of experiments to determine the validity and throughput of the fingerprint scrubber. They show that our current implementation blocks known fingerprint scan attempts and can match the performance of a plain IP forwarding gateway on the same hardware. The experiments were conducted using a set of kernels with different fingerprint scrubbing options enabled for comparison. The scrubber and end hosts each had 500-MHz Pentium III CPUs and 128 MB of main memory. The end hosts each had one 3Com 3c905B Fast Etherlink XL 10/100BaseTX Ethernet card (`xl` device driver). The gateway had two Intel EtherExpress Pro 10/100B Ethernet cards (`fxp` device driver). The network was configured as shown in Fig. 4.

1) Defeating Fingerprint Scans: To verify that our fingerprint scrubber did indeed defeat known scan attempts, we interposed our gateway in front of a set of machines running different operating systems. The operating systems we ran scans against under controlled conditions in our lab were FreeBSD 2.2.8, Solaris 2.7 x86, Windows NT 4.0 SP 3, and Linux 2.2.12. We also ran scans against a number of popular Web sites, and campus workstations, servers, and printers.

`Nmap` was consistently able to determine all of the host operating systems without the fingerprint scrubber interposed. However, it was completely unable to make even a close guess with the fingerprint scrubber interposed. In fact, it was not able to distinguish much about the hosts at all. For example, without the scrubber `nmap` was able to accurately identify a FreeBSD 2.2.8 system in our lab. With the scrubber `nmap` guessed 14 different

```
Remote operating system guess:
   FreeBSD 2.2.1 - 3.2
```

(a)

```
Remote OS guesses:
   AIX 4.0 - 4.1, AIX 4.02.0001.0000,
   AIX 4.1, AIX 4.1.5.0, AIX 4.2,
   AIX 4.3.2.0 on an IBM RS/*,
   Raptor Firewall 6 on Solaris 2.6,
   Solaris 2.5, 2.5.1, Solaris 2.6 - 2.7,
   Solaris 2.6 - 2.7 X86,
   Solaris 2.6 - 2.7 with tcp_strong_iss=0,
   Solaris 2.6 - 2.7 with tcp_strong_iss=2,
   Sun Solaris 8 early acces beta (5.8)
     Beta_Refresh February 2000
```

(b)

Fig. 10. Operating system guess before and after fingerprint scrubbing for an `nmap` scan against a machine running FreeBSD 2.2.8. (a) Before fingerprint scrubbing. (b) After fingerprint scrubbing.

TABLE III
THROUGHPUT FOR A SINGLE UNTRUSTED HOST TO A TRUSTED HOST USING TCP (Mb/s, ±2.5% AT 99% CI)

IP Forwarding	87.06
Fingerprint Scrubbing	86.86
Fingerprint Scrub. + Frag. Reas.	87.00

TABLE IV
THROUGHPUT FOR A SINGLE TRUSTED HOST TO AN UNTRUSTED HOST USING TCP (Mb/s, ±2.5% AT 99% CI)

IP Forwarding	87.06
Fingerprint Scrubbing	86.79
Fingerprint Scrub. + Frag. Reas.	86.84

operating systems from three vendors. Each guess was wrong. Fig. 10 shows a condensed result of the guesses `nmap` made against FreeBSD before and after interposing the scrubber.

The two main components that aid in blocking `nmap` are the enforcement of a three-way handshake for TCP and the reordering of TCP options. Many of `nmap`'s scans work by sending probes without the SYN flag set so they are discarded right away. Similarly, operating systems vary greatly in the order that they return TCP options. Therefore, `nmap` suffers from a large loss in available information. Even though `nmap` can be confused by these simple techniques, we intend this tool to be general enough to block new scans. We believe that the inclusion of IP header flag normalization and IP fragment reassembly aid in that goal even though we do not know of any existing tool that exploits such differences.

2) Throughput: We measured both the throughput from the trusted side out to the untrusted side and from the untrusted side into the trusted side using the Netperf benchmark [26]. This was to take into account our asymmetric filtering of the traffic. We ran experiments for TCP traffic to show the effect of a bulk TCP transfer and for UDP to exercise the fragment reassembly code. We used three kernels on the gateway machine to test different functionality of the fingerprint scrubber. The IP forwarding kernel is the unmodified FreeBSD kernel, which we use as our baseline for comparison. The fingerprint scrubbing kernel includes the TCP options reordering, IP header flag normalization, ICMP modifications, and TCP sequence number

TABLE V
THROUGHPUT FOR A SINGLE UNTRUSTED HOST TO A TRUSTED HOST USING UDP (Mb/s, ±2.5% AT 99% CI)

Forwarding Type	64 bytes	1472 bytes	2048 bytes	8192 bytes
IP Forwarding	14.39	89.39	92.76	90.11
Fingerprint Scrubbing + Frag. Reas.	14.48	89.35	92.76	90.11

TABLE VI
THROUGHPUT FOR A SINGLE TRUSTED HOST TO AN UNTRUSTED HOST USING UDP (Mb/s, ±2.5% AT 99% CI)

Forwarding Type	64 bytes	1472 bytes	2048 bytes	8192 bytes
IP Forwarding	14.39	89.39	92.76	90.11
Fingerprint Scrubbing + Frag. Reas.	14.40	89.37	92.76	90.12

TABLE VII
THROUGHPUT FOR A SINGLE UNTRUSTED HOST TO AN UNTRUSTED HOST USING TCP ON A GIGABIT SPEED NETWORK (Mb/s, ±2.5% AT 99% CI)

Crossover	359.13
Bridge	333.81
Fingerprint Scrubbing	333.45

modification, but not IP fragment reassembly. The last kernel is the full fingerprint scrubber with fragment reassembly code turned on.

Table III shows the TCP bulk transfer results for an untrusted host connecting to a trusted host. Table IV shows the results for a trusted host connecting to an untrusted host. The first result is that both directions show the same throughput. The second, and more important, result is that even when all of the fingerprint scrubber's functionality is enabled we are seeing a throughput almost exactly that of the plain IP forwarding. The bandwidth of the link is obviously the limiting factor.

We ran the UDP experiment with the IP forwarding kernel and the fingerprint scrubbing kernel with IP fragment reassembly. Again, we measured both the untrusted to trusted direction and *vice versa*. To measure the affects of fragmentation, we ran the test at varying sizes up to the MTU of the Ethernet link and above. Note that 1472 bytes is the maximum UDP data payload that can be transmitted since the UDP plus IP headers add an additional 28 bytes to get up to the 1500-byte MTU of the link. The 2048-byte test corresponds to two fragments and the 8192-byte test corresponds to five fragments. At a size of 64 bytes, the scrubber spends most of its time handling device interrupts.

Table V shows the UDP transfer results for an untrusted host connecting to a trusted host. Table VI shows the results for a trusted host connecting to an untrusted host. Once again, both directions show the same throughput. We also see that the throughput of the fingerprint scrubber with IP fragment reassembly is almost exactly that of the plain IP forwarding. This is even true in the case of the 8192-byte test where the fragments must be reassembled at the gateway and then refragmented before being sent out.

We also ran the fingerprint scrubber experiments on a gigabit-speed network. As with the TCP scrubber, we found that the fingerprint scrubber added a small performance penalty compared to simple computer-based forwarding techniques. Table VII shows the results from using Netperf to test the bulk transfer throughput. The results are very similar to the 100-Mb/s tests: the fingerprint scrubber adds very little overhead to simple forwarding tests.

V. CONCLUSION AND FUTURE WORK

This paper presented the design and implementation of protocol scrubbers, which are active interposed mechanisms for transparently removing attacks from protocol layers in real time. The key contributions of this work are the identification of transport scrubbing as a mechanism that enables passive NID systems to operate correctly, the design and implementation of the high performance half-duplex TCP/IP scrubber, and the creation of a TCP/IP stack fingerprint scrubber. The transport scrubber converts ambiguous network flows into well-behaved flows that are interpreted identically at all downstream endpoints. While the security community has examined application proxies, the concept of removing transport level attacks through a transport scrubber is new. The fingerprint scrubber removes clues about the identity of an end host's operating system to successfully and completely block known scans. Because of its general design, it should also be effective against any evolutionary enhancements to fingerprint scanners. By protecting networks against scans, we block the first step in an attacker's assault, increasing the security of a heterogeneous network.

Our future work will involve improving the protocol scrubbers' performance. Specifically, we plan to incorporate zero-copying techniques into the TCP scrubber's data handling routines, bringing the performance even closer to high-speed networking levels—1 Gb/s and beyond. To achieve such speeds, we would like to implement core components of the protocol scrubbers in hardware. One possible approach would be to implement the minimal TCP state machine on a platform such as Intel's Internet Exchange Architecture (IXA) [6], [7]. We believe that integrating these solutions within the context of existing security devices, specifically firewalls, will provide significant benefits to network security.

REFERENCES

[1] G. R. Malan, D. Watson, F. Jahanian, and P. Howell, "Transport and application protocol scrubbing," in *Proc. IEEE INFOCOM*, Tel Aviv, Israel, Mar. 2000, pp. 1381–1390.

[2] M. Smart, G. R. Malan, and F. Jahanian, "Defeating TCP/IP stack fingerprinting," in *Proc. 9th USENIX Security Symp.*, Denver, CO, Aug. 2000.

[3] Remote OS detection via TCP/IP stack fingerprinting. (1998). [Online]. Available: http://www.insecure.org/nmap/nmap-fingerprinting-article.html

[4] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," Secure Networks, Inc., Tech. Rep., Jan. 1998.
[5] FreeBSD. [Online]. Available: http://freebsd.org
[6] D. Putzolu, S. Bakshi, S. Yadav, and R. Yavatkar, "The Phoenix framework: A practical architecture for programmable networks," *IEEE Commun.*, vol. 38, pp. 160–165, Mar. 2000.
[7] Intel Internet Exchange Architecture. [Online]. Available: http://www.intel.com/design/network/ixa.htm
[8] D. B. Chapman and E. D. Zwicky, *Building Internet Firewalls*. Cambridge, MA: O'Reilly, 1995.
[9] TIS Firewall Toolkit, T. I. Systems. [Online]. Available: ftp://ftp.tis.com/firewalls/toolkit
[10] FireWall-1, C. P. S. Technologies. [Online]. Available: http://www.checkpoint.com
[11] Gauntlet Firewall, N. A., Inc. [Online]. Available: http://www.pgp.com/gauntlet/
[12] B. Mukherjee, L. T. Heberlein, and K. N. Levitt, "Network intrusion detection," *IEEE Network*, vol. 8, pp. 26–41, May/June 1994.
[13] M. J. Ranum. (1998) Intrusion Detection: Challenges and Myths. Network Flight Recorder, Inc. [Online]. Available: http://www.nfr.com/
[14] G. B. White, E. A. Fisch, and U. W. Pooch, "Cooperating security managers: A peer-based intrusion detection system," *IEEE Network*, vol. 10, pp. 20–23, Jan./Feb. 1996.
[15] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall, "Implementing a generalized tool for network monitoring," in *Proc. 11th Systems Administration Conf.*, San Diego, CA, Oct. 1997.
[16] V. Paxson, "Bro: A system for detecting network intruders in real time," *Comput. Netw.*, vol. 31, no. 23–24, pp. 2435–2463, Dec. 1999.
[17] RealSecure, I. S. Services. [Online]. Available: http://www.iss.net/products_services/enterprise_protection/rsnetwork/
[18] V. Paxson, "Automated packet trace analysis of TCP implementations," in *Proc. ACM SIGCOMM*, Cannes, France, Sept. 1997.
[19] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol,", RFC 2401, Nov. 1998.
[20] Iplog, R. McCabe. [Online]. Available: http://ojnk.sourceforge.net/
[21] M. Handley, C. Kreibich, and V. Paxson, "Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics," in *Proc. 10th USENIX Security Symp.*, Washington, DC, Aug. 2001.
[22] V. Paxson and M. Handley, "Defending against NIDS evasion using traffic normalizers," in *2nd Int. Workshop Recent Advances in Intrusion Detection*, Sept. 1999.
[23] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, MA: Addison-Wesley, 1994.
[24] Fragroute, D. Song. [Online]. Available: http://www.monkey.org/~dugsong/fragroute/
[25] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*. Reading, MA: Addison-Wesley, 1995.
[26] Netperf: A Network Performance Benchmark [Online]. Available: http://www.netperf.org/
[27] D. Maltz and P. Bhagwat, "TCP splicing for application layer proxy performance," IBM Res. Div., Tech. Rep. RC 21139, Mar. 1998.
[28] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson, "Optimizing TCP forwarder performance," Dept. Comput. Sci., Univ. Arizona, Tech. Rep. TR98-01, Feb. 1998.
[29] L. Rizzo, "Dummynet: A simple approach to the evaluation of network protocols," *ACM Comput. Commun. Rev.*, Jan. 1997.
[30] F. Baker, "Requirements for IP version 4 routers,", RFC 1812, 1995.

David Watson received the B.S. degree from Carnegie Mellon University, Pittsburgh, PA, and the M.S.E. degree from the University of Michigan, Ann Arbor. He is currently working toward the Ph.D. degree at the University of Michigan.

His research interests include network routing protocols as well as network infrastructure security.

Matthew Smart received the B.S.E. and M.S.E. degrees in computer science and engineering from the University of Michigan, Ann Arbor.

He is currently a Senior Software Engineer with Arbor Networks, Ann Arbor, a provider of Internet monitoring and security tools. His research interests include large-scale network monitoring and network security.

G. Robert Malan (M'00) received the B.S.E. degree from Carnegie Mellon University, Pittsburgh, PA, and the M.S.E. and Ph.D. degrees from the University of Michigan, Ann Arbor.

He is a Founder and Chief Technology Officer of Arbor Networks. The focus of his research interest is primarily on network security and measurement. He has been particularly active in the Internet service provider and enterprise security communities in the recent years.

Dr. Malan has been a member of the Association of Computing Machinery since 1994.

Farnam Jahanian (M'89) received the M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1987 and 1989, respectively.

He is currently a Professor of electrical engineering and computer science at the University of Michigan, Ann Arbor, and Chief Scientist with Arbor Networks, Ann Arbor. Prior to joining the faculty of the University of Michigan in 1993, he was a Research Staff Member at the IBM T. J. Watson Research Center. His research interests include the study of scalability, reliability, and security of computer networks and distributed systems.

Dr. Jahanian has been a member of the Association for Computing Machinery since 1989.