Protocol Scrubbing: Network Security through Transparent Flow Modification

David Watson, Matthew Smart, G. Robert Malan, and Farnam Jahanian
University of Michigan
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2122
{*dwatson,mcsmart,rmalan,farnam*}*@eecs.umich.edu*

Abstract

This paper describes the design and implementation of protocol scrubbers. Protocol scrubbers are transparent, active interposition 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, simplifed 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.

1. 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 vitally important. 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 network flows is the only way to immediately detect or prevent these attacks. This paper presents the design and implementation of *protocol scrubbers* – transparent, active interposition mechanisms for explicitly 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 ambiguities, enabling downstream passive network-based intrusion detection systems to operate with high assurance [5]. The *fingerprint scrubber* prevents a remote user from detecting the operating system of another host at the TCP/IP layers by actively homogenizing flows [19].

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 active interposition mechanism. Through interposition, the transport scrubber can guarantee protocol invariants that enable downstream intrusion detection systems to work with confidence. Because the Internet protocols are well described, *correct* implementations 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 protocol as a benign exchange. Examples of these ambiguities 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 universally interpret. The transport scrubber's conversion of ambiguous network flows into unequivocal interpretations 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 protocols at the edges of an interior network and modifies them in such a way that their security attributes are predictable.

Differences in protocol implementations 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 `nmap` [3] and `queso` [14]) 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 for a specific operating system, it is simple for the attacker to run the exploit against each corresponding host matching that operating system. This makes it very easy for an attacker to systematically install malicious code, such as distributed denial of service tools, on many machines. Current fingerprinting techniques provide fine-grained determination of an operating system. For example, the current version of `nmap` has knowledge of 15 different versions of Linux. Also, 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:

- *Identification 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 network-based intrusion detection systems. The paper describes the use of transport scrubbers to eliminate insertion and evasion attacks on NID systems [12]. 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 [2]. This implementation is shown to scale with commercial stateful inspection firewalls and raw Unix-based IP forwarding routers. By keeping the design of the scrubber general, we plan to migrate the implementation to programmable networking hardware such as the Intel IXA architecture [13; 4].

- *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 2 places our work within the broader context of related work. Section 3 describes the design, implementation and performance characteristics of our TCP transport scrubber. Section 4 presents our tool for defeating TCP/IP stack fingerprinting. Finally, Section 5 presents our conclusions and plans for future work.

2. Related Work

Firewall technologies [1] are closely related to protocol scrubbers. They are both active interposition mechanisms – packets must physically travel through them in order to continue towards 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. However, due to performance reasons, once a firewall has identified an authorized flow, packets are routed through a fast-path and are not scrutinized further for attacks. In contrast to firewalls, the protocol scrubber's primary function is to homogenize network flows, identifying and removing attacks in real-time. The scrubbers are utilized to remove attacks present within the protocols once a firewall has authorized a flow's access. As such, scrubbing technology can easily be added to existing firewall technologies to significantly enhance network security.

Intrusion Detection Systems (ID systems) [8; 15] are also closely related to protocol scrubbers. Network-based Intrusion Detection Systems (NID systems) are implemented as passive network monitors that reconstruct networking flows and monitor protocol events through eavesdropping techniques [24; 16; 11; 18]. As passive observers, NID systems have a vantage point problem [10] when reconstructing the semantics of passing network flows. This vulnerability can be exploited by sophisticated network attacks that understand the inherent schism between the protocol's destination and an intermediary observer [12]. As active participants 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.

Fingerprinting scans may be preludes to futher 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` [7] 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. In addition, these methods fail to protect networks that are not under single administrative control, unlike the fingerprint scrubber.

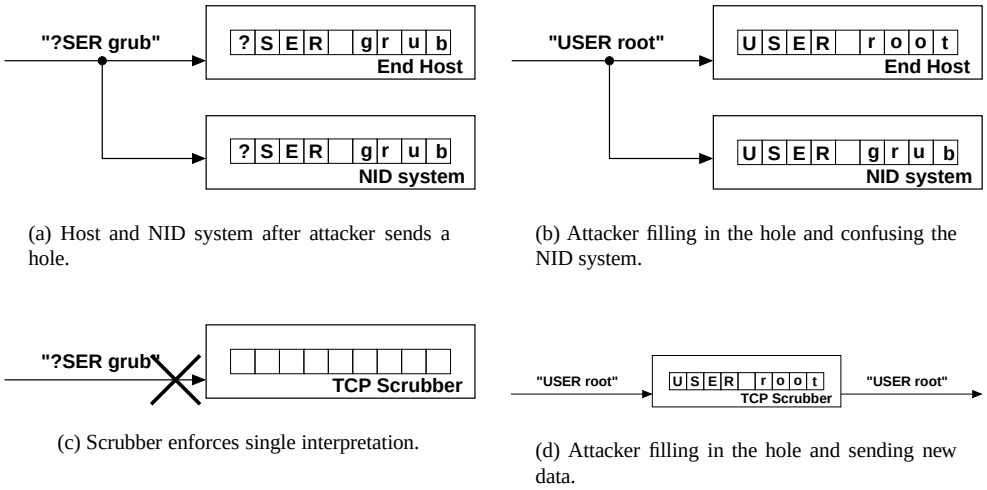
3. Transport Scrubber

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. By enforcing protocol invariants on the downstream flows, the TCP scrubber eliminates TCP insertion and evasion attacks against NID systems that can render them useless. 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 invariants (the TCP scrubber only reconstructs the incoming half of the connection). By keeping a significantly smaller amount of state, 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.

3.1. TCP Ambiguities and ID Evasion

Sophisticated attacks can utilize protocol ambiguities between a network intrusion detection system and an end-host to slip past the watching NID system completely undetected. Network ID systems rely on their ability to correctly predict the effect of observed packets on an end-host system in order to be useful. In [12], Ptacek and Newsham describe a class of attacks that leave NID systems wide open to subversion. 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.

Figure 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 Figure 1a 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 byte-stream 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 [21]. 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 Figure 1b when the attacker resends the data



(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.

Figure 1. Example of ambiguity of transport layer protocol implementation differences between an interposed agent (NID system) and an end host.

with the hole filled, but with a different username of the same length, the difference in implementation choice between the two systems allows the attack 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 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 implementation of the networking stacks allows sophisticated attacks to succeed.

To address this problem, we have created the TCP scrubber. Specifically, the scrubber provides the invariants that NID systems need for confident flow reconstruction and end-host behavior prediction. For example, the scrubber stores unacknowledged data from the TCP sequence space. When any unacknowledged data is retransmitted, the original data is copied to prevent possible ambiguity. When acknowledged, this data is thrown away and is removed from any subsequent packets. Specifically, Figures 1c and 1d demonstrate 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 – in this case the scrubber simply throws away the data after a hole – both the downstream NID system and end host both see the attacker logging in as `root`.

In addition to the handling of overlapping TCP segments, there are many other ambiguities in the implementation of the TCP/IP stack [12]. 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 [21]. 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.

3.2. TCP Scrubber Design and Implementation

The TCP scrubber converts external network flows – sequences of network packets that may be ambiguously interpreted 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. The protocol scrubber's job is to codify what consists of 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 in a paper of this length, 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 ambiguites of the TCP/IP stack when we discuss the fingerprint scrubber in Section 4.

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 non-standard usage of TCP/IP header fields with little overhead, however they are incapable of converting a stateful protocol, such as TCP, into a non-ambiguous 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, round-trip time estimation, window size calculations, 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 data 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 a situation arises that needs bidirectional scrubbing, it can be configured in the scrubber. 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 [22]. 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.

Table 1. Throughput for a single external connection to an internal host (Mbps, ± 2.5% at 99% CI).

IP Forwarding	Scrubbing	Plug Proxy
83.84	82.87	82.71

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 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 throws away any packets that could lead to inconsistencies.

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 2.2.7 kernel's networking stack – a derivative of the BSD 4.4 code [25].

3.3. 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 IP forwarding routers when used in networks of up to 500 Mbit per second. For all of the experiments, the interposed machine that ran the TCP scrubbing kernel, the IP forwarding kernel, and the TIS FWTK `plug-gw` proxy was the same: a 300 MHz Pentium II CPU, 128 megabytes main memory, and two Intel EtherExpress Pro 10/100B Ethernet (`fxp` device driver) cards. The TCP scrubbing kernel was used to generate the scrubber's statistics. An unmodified FreeBSD 2.2.7 kernel was used for the IP forwarding numbers. Finally, a modified 2.2.7 kernel was used as a substrate for the `plug-gw` experiments.

Several experiments were undertaken to determine the maximum sustainable bandwidth for the TCP scrubber. The results in Table 1 provide a baseline measure-

Table 2. 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

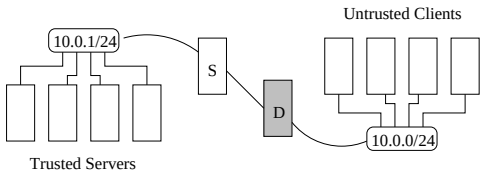
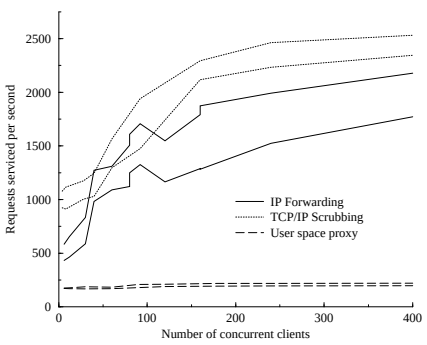


Figure 2. Experimental apparatus for measuring the protocol scrubber's implementation.

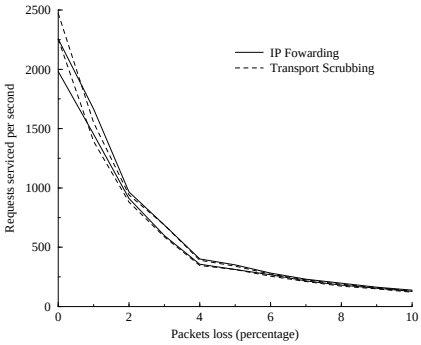
ment of the maximum TCP throughput for a single connection. This throughput was measured using the Netperf benchmark [9]. Three machines were used for the test; all were connected through a 100 Mbps Ethernet switch. The performance of all three forwarding mechanisms were comparable – the networking bandwidth was clearly the first-order bottleneck. In the absence of larger capacity networking resources, we undertook a series of microbenchmarks to pinpoint the TCP scrubber's maximum throughput. 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. Numbers were not gathered for the plug-proxy due to difficulty in matching incoming packets bound for one socket buffer to the outgoing packets from another. Table 2 shows the results from this experiment. From these numbers it is possible to calculate the optimal sustained throughput (excluding interrupt handling overhead) of both the IP forwarding and TCP scrubber. For scrubbing a stream of TCP packets with full-sized data payloads, the current implementation's ceiling on our test hardware is 366Mbps. We believe that with optimizations and fewer data copies we could increase this ceiling to 891Mbps (13.19 usec latency for scrubbing 1460 byte data payloads).

The next set of experiments show that the TCP scrubber does not have a negative impact on the performance characteristics of well-behaved TCP streams. We show this by measuring the sustainable client-server connections per second (similar to transactions) from a set of external client machines to a set of internal server machines. Specifically, an external set of custom web clients made identical fetches to an internal set of Apache web servers. The clients repeatedly fetched 1K byte pages from the servers, stressing the connection setup and teardown process. Figure 2 shows the experimental configuration used in these experiments. The server's $10.0.1/24$ network is comprised of an Intel Express 10/100 Ethernet switch, whereas the client's $10.0.0/24$ network is an Intel Express 10/100 Ethernet hub. The experiments were measured using a promiscuous mechanism attached to the client-side hub. The TCP scrubber, IP forwarding router, and `plug-gw` proxy all ran on the S machine. For a second set of experiments, a `dummynet` router was used as machine D [17]. All ten machines were equipped identically to the TCP scrubber described above. The clients and servers all ran a modified FreeBSD 2.2.7 kernel that was compiled with a large `maxusers` constant.

Figure 3a shows the number of sustained connections per second measured for the TCP scrubber, the IP forwarding router, and the user-space `plug-gw` proxy. The pairs of lines in the graph represent the 99% confidence intervals for the mean sustainable connections per second. The results are twofold: the TCP scrubber's performance is comparable, even better than the raw IP forwarding kernel, and the user-level proxy's performance is extremely low compared to the two in-kernel implementations. The first result is a somewhat surprising; however, when looking closely at the data it can be explained by buffering at the TCP scrubber. By buffering the incoming TCP connections, the TCP scrubber shapes the traffic that the servers see, effectively smoothing the request streams so that they are more easily handled at the receivers. These results only apply with very short-lived bursty traffic; the TCP scrubber's performance would decrease relative to IP forwarding when scrubbing long-lived flows. However, this decrease would be relatively small on low-bandwidth networks (100Mbps) as shown in Table 1. The second result is not a surprise – we expected the plug-proxy to be slow. The original `plug-gw` code was modified so that it did no logging and no DNS resolutions, which resulted in a large performance increase. The proxy's kernel was also modified so that a large number of processes could be accommodated. A custom user-space proxy optimized for speed would certainly do better (the `plug-gw` proxy forks a child for each incoming connection). However, the multiple data copies and context switching will always resign any user-space implemen-



(a) Connections per second with no artificial loss.



(b) Connections per second with 480 clients and varied artificial loss.

Figure 3. TCP scrubber scalability results.

tation to significantly worse performance than the two in-kernel approaches [6; 20].

Finally, we conducted a set of experiments to determine the effects of a lossy link between the external clients and the interposed machine. 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, labeled *D* in Figure 2. The results of this experiment are shown in Figure 3b. The vertical axis represents the number of 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 main result from this experiment is that the TCP scrubbed flows behave comparably to the raw IP forwarded flows.

To put these results in perspective it is useful to compare them with the performance of a fast commercial firewall. CheckPoint reports in a performance white paper that the peak throughput for their FireWall-1 product on a dual 167 MHz UltraSparc with four 100 Mbps Ethernet adapters (200 Mbps on each side) is 89.75 Mbps [23]. While it is difficult to accurately compare the results from separate performance experiments, the TCP scrubber's performance is clearly as good as current firewall technology.

4. Fingerprint Scrubber

We created a second protocol scrubber to remove TCP/IP fingerprinting scans as a transparent, active interposition mechanism. 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 380 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 speaks 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 protocol and one is targeted at the UDP protocol. The TCP tests are the most important because TCP implementations vary significantly. `Nmap` looks at the order of TCP options, the pattern of initial sequence numbers, IP-level flags such as the don't fragment bit, the TCP flags such as RST, the advertised window size, as well as other ambiguities. For more details, including the specific options set in the test packets, refer to [3].

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

(a) Web server running Linux.

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

(b) Shared printer.

Figure 4. Output of an `nmap` scan against a web server running Linux and a shared printer.

Figure 4 is an example of the output of `nmap` when scanning our EECS department's web server, `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` contains a lot of functionality and 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. 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 work.

In this section we discuss the goals and intended use of the scrubber as well as its design and implementation. We demonstrate 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.

4.1. 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. Windows, Solaris, MacOS, 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 put 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.

4.2. 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 layers 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 at a generic level for both IP and TCP, we feel we have raised the bar sufficiently high.

4.2.1. 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 and make it easier to fingerprint an operating

system. 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 ourself. 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 on the way out the interface.

4.2.2. ICMP Scrubbing

As with the IP layer, ICMP messages also contain ambiguities that can be used for fingerprinting. 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.

4.2.3. 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 don't 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, meaning difficult to detect, techniques for port scanning don't perform a 3WHS and are therefore blocked. While this 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 identifiy 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 orderding 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.

4.3. 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 256 megabytes 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 Figure 2.

4.3.1. 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 wasn't 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 operating systems from three vendors. Each guess was wrong.

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.

We intend this tool to be general enough to block new scans. We believe that the inclusion of IP header flag

Table 3. Throughput for a single untrusted host to a trusted host using TCP (Mbps, ±2.5% at 99% CI).

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

Table 4. Throughput for a single trusted host to an untrusted host using TCP (Mbps, ±2.5% at 99% CI).

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

normalization and IP fragment reassembly aid in that goal even though we do not know of any existing tool that exploits such differences.

4.3.2. 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 [9]. This was to take into account our asymmetric filtering of the traffic. We ran experiments for TCP traffic to show the affect 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 modification but not IP fragment reassembly. The last kernel is the full fingerprint scrubber with fragment reassembly code turned on.

Table 3 shows the TCP bulk transfer results for an untrusted host connecting to a trusted host. Table 4 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 critical factor, therefore we would like to run these experiments again on a faster network.

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 5 shows the UDP transfer results for an untrusted host connecting to a trusted host. Table 6 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 re-fragmented before being sent out.

5. Conclusion

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 has not been previously introduced. The fingerprint scrubber removes clues about the identity of an end host's operating system to successfully and completely blocks 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.

Acknowledgments

The Intel Corporation provided support for this work through a generous equipment donation and gift. This

Table 5. Throughput for a single untrusted host to a trusted host using UDP (Mbps, ±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 6. Throughput for a single trusted host to an untrusted host using UDP (Mbps, ±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

work was also supported 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.

References

[1] D. B. Chapman and E. D. Zwicky. *Building Internet Firewalls*. O'Reily and Associates, Inc., 1995.

[2] FreeBSD Homepage. `http://freebsd.org`.

[3] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`, October 1998.

[4] Intel Internet Exchange Architecture. `http://developer.intel.com/design/IXA/`.

[5] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and Application Protocol Scrubbing. In *Proceedings of the IEEE INFOCOMM 2000 Conference*, Tel Aviv, Israel, March 2000.

[6] D. Maltz and P. Bhagwat. TCP Splicing for Application Layer Proxy Performance. Technical Report RC 21139, IBM Research Division, March 1998.

[7] R. McCabe. Iplog. `http://ojnk.sourceforge.net/`.

[8] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network Intrusion Detection. *IEEE Network*, 8(3):26–41, May and June 1994.

[9] Netperf: A Network Performance Benchmark. `http://www.netperf.org/`.

[10] V. Paxson. Automated Packet Trace Analysis of TCP Implementations. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[11] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[12] T. H. Ptacek and T. N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Originally Secure Networks, Inc., now available as a white paper at the Network Associates Inc., `http://www.nai.com/`, January 1998.

[13] D. Putzolu, S. Bakshi, S. Yadav, and R. Yavatkar. The Phoenix Framework: A Practical Architecture for Programmable Networks. *IEEE Communications*, 38(3):160–165, March 2000.

[14] Queso Homepage. `http://www.apostols.org/projectz/queso/`.

[15] M. J. Ranum. Intrusion Detection: Challenges and Myths. Network Flight Recorder, Inc. whitepaper at `http://www.nfr.com/`, 1998.

[16] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a Generalized Tool for Network Monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA '97)*, San Diego, CA, October 1997.

[17] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[18] I. S. Services. RealSecure^{TM}. `http://www.iss.net/prod/rs.html`.

[19] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP Stack Fingerprinting. In *Proceedings of 9th USENIX Security Symposium*, Denver, Colorado, August 2000.

[20] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP Forwarder Performance. Technical Report TR98-01, Dept. of Computer Science, University of Arizona, February 1998.

[21] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[22] T. I. Systems. TIS Firewall Toolkit. `ftp://ftp.tis.com/pub/firewalls/toolkit`.

[23] C. P. S. Technologies. FireWall-1. `http://www.checkpoint.com`.

[24] G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating Security Managers: A Peer-Based Intrusion Detection System. *IEEE Network*, 10(1):20–23, January and February 1996.

[25] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.