Defeating TCP/IP Stack Fingerprinting

Matthew Smart G. Robert Malan Farnam Jahanian
Department of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Ave.
Ann Arbor, Mich. 48109-2122
{mcsmart,rmalan,farnam}@eecs.umich.edu

Abstract

This paper describes the design and implementation of a TCP/IP stack fingerprint scrubber. The fingerprint scrubber is a new tool to restrict a remote user's ability to determine the operating system of another host on the network. Allowing entire subnetworks to be remotely scanned and characterized opens up security vulnerabilities. Specifically, operating system exploits can be efficiently run against a pre-scanned network because exploits will usually only work against a specific operating system or software running on that platform. The fingerprint scrubber works at both the network and transport layers to convert ambiguous traffic from a heterogeneous group of hosts into sanitized packets that do not reveal clues about the hosts' operating systems. This paper evaluates the performance of a fingerprint scrubber implemented in the FreeBSD kernel and looks at the limitations of this approach.

1 Description

TCP/IP stack fingerprinting is the process of determining the identity of a remote host's operating system by analyzing packets from that host. Freely available tools (such as `nmap` [3] and `queso` [15]) 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. The difference is that in real fingerprinting, law enforcement agencies use fingerprinting to track down suspected criminals; in computer networking potential attackers can use fingerprinting to quickly create a list of targets.

We argue that fingerprinting tools can be used to aid unscrupulous users in their attempts to break into or disrupt computer systems. A user can build up a profile of IP addresses and corresponding operating systems for later attacks. `Nmap` can scan a subnetwork of 254 hosts in only a few seconds, or it can be set up to scan very slowly, i.e. over days. These reports can be compiled over weeks or months and cover large portions of a network. When someone discovers a new exploit for a specific operating system, it is simple for an attacker to generate a script to run the exploit against each corresponding host matching that operating system. An example might be an exploit that installs code on a machine to take part in a distributed denial of service attack. Fingerprinting scans can also potentially use non-trivial amounts of network resources including bandwidth and processing time by intrusion detection systems and routers.

Fingerprinting provides fine-grained determination of an operating system. For example, `nmap` has knowledge of 21 different versions of Linux. Other methods of determining an operating system are generally coarse-grained because they use application-level methods. An example is the banner message a user receives when he or she uses telnet to connect to a machine. Many systems freely advertise their operating system in this way. This paper does not deal with blocking application-level fingerprinting because it must be dealt with on an application by application basis.

Almost every system connected to the Internet is vulnerable to fingerprinting. The major operating systems are not the only TCP/IP stacks identified by fingerprinting tools. Routers, switches, hubs, bridges, embedded systems, printers, firewalls, web cameras, and even game consoles are identifiable. Many of these systems, like routers, are important parts of the Internet infrastructure, and compromising infrastructure is a more serious problem than compromising end hosts. Therefore a general mechanism to protect any system is needed.

Some people may consider stack fingerprinting a nuisance rather than a security attack. As with most tools, fingerprinting has both good and bad uses. Network administrators should be able to fingerprint machines under their control to find known vulnerabilities. Stack fingerprinting is not necessarily illegal or an indication of malicious behavior, but we believe the number of scans will grow in frequency as more people access the Internet and discover easy to use tools such as `nmap`. As such, network administrators may not be willing to spend time or money tracking down what they consider petty abuses each time they occur. Instead they may choose to reserve their resources for full-blown intrusions. Also, there may be networks that no single authority has administrative control over, such as a university residence hall. A tool that detects fingerprinting scans but turns them away would allow administrators to track attempts while keeping them from penetrating into local networks.

This paper presents the design and implementation of a tool to defeat TCP/IP stack fingerprinting. We call this new tool a *fingerprint scrubber*. The fingerprint scrubber is transparently interposed between the Internet and the network under protection. The intended use of the scrubber is for it to be placed in front of a set of end hosts or a set of network infrastructure components. The goal of the tool is to block the majority of stack fingerprinting techniques in a general, fast, scalable, and transparent manner.

We describe an experimental evaluation of the tool and show that our implementation blocks known fingerprint scan attempts and is prepared to block future scans. We also show that our fingerprint scrubber can match the performance of a plain IP forwarding gateway on the same hardware and is an order of magnitude more scalable than a transport-level firewall.

The remaining sections are organized as follows. We describe TCP/IP stack fingerprinting in more detail in Section 2. In Section 3 we describe the design and implementation of our fingerprint scrubber. In Section 4 we evaluate the validity and performance of the scrubber. In Section 5 we cover related work and in Section 6 we cover future directions. Finally, in Section 7 we summarize our work.

2 TCP/IP Stack Fingerprinting

The most complete and widely used TCP/IP fingerprinting tool today is `nmap`. It uses a database of over 450 fingerprints to match TCP/IP stacks to a specific operating system or hardware platform. This database includes commercial operating systems, routers, switches, firewalls, 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 layer and one is targeted at the UDP layer. The TCP tests are the most important because TCP has a lot of options and variability in implementations. `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, and a few more things. For more details, including the specific options set in the test packets, refer to the home page for `nmap` [3].

Figure 1 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.

(a)

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

(b)

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

Figure 1: Output of an `nmap` scan against (a) a web server running Linux and (b) a shared printer.

While `nmap` contains a lot of functionality and does a good job of performing fine-grained fingerprinting, it does not implement all of the techniques that could be used. Various timing-related scans could be performed. For example, determining 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 3.2.4. 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. However, there will still be a need to block TCP/IP fingerprinting scans even if an application-level fingerprinting tool is developed. Currently, TCP/IP fingerprinting is the fastest and easiest method for identifying remote hosts' operating systems, and introducing techniques that target applications will not make it obsolete.

3 Fingerprint Scrubber

We developed a tool called a fingerprint scrubber to remove ambiguities from TCP/IP traffic that give clues to a host's operating system. In this section we discuss the goals and intended use of the scrubber and its design and implementation. We demonstrate the validity of the scrubber in the face of known fingerprinting scans and give performance results in the next section.

3.1 Goals and Intended Use of Fingerprint Scrubber

The goal of the fingerprint scrubber is to block known stack fingerprinting techniques in a general, fast, scalable, 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 heterogeneous 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. The network under protection must be restricted to having one connection to the outside world because all packets traveling to and from a host must travel through the scrubber.

Because the scrubber affects only traffic moving through it, an administrator on the trusted side of the network will still be able to scan the network. Alternatively, an IP access list or some other authentication mechanism could be added to the fingerprint scrubber to allow authorized hosts to bypass scrubbing.

3.2 Fingerprint Scrubber Design and Implementation

We designed the fingerprint scrubber to be placed between a trusted network of heterogeneous systems and an untrusted connection (i.e. the Internet). The scrubber has two interfaces; one interface is designated as *trusted*, and the other is designated as *untrusted*. A packet coming from the untrusted interface is forwarded out the trusted interface and vice versa. The basic design principle is that data

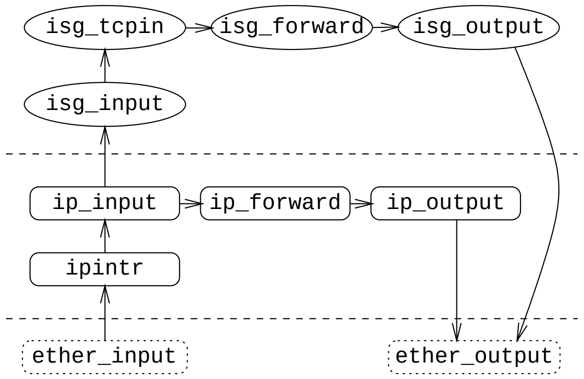

Figure 2: Data flow through modified FreeBSD kernel.

coming in from the untrusted interface is handled differently than data traveling out to the untrusted interface.

The fingerprint scrubber operates at the IP and TCP layers to cover a wide range of known and potential fingerprinting scans. We could have simply implemented a few of the 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.

The fingerprint scrubber is based off the protocol scrubber by Malan, et al. [7]. The protocol scrubber operates at the IP and TCP layers of the protocol stack. It is a set of kernel modifications to allow fast TCP flow reassembly to avoid TCP insertion and deletion attacks as described by Ptacek and Newsham [13]. The protocol scrubber follows TCP state transitions by maintaining a small amount of state for each connection, but it leaves the bulk of the TCP processing and state maintenance to the end hosts. This allows a tradeoff between the performance of a stateless solution with the control of a full transport-layer proxy. The protocol scrubber is implemented under FreeBSD, and we continued under FreeBSD 2.2.8 for our development.

Figure 2 shows the data flow through the kernel for the fingerprint scrubber. Packets come in from either the trusted or untrusted interface through an Ethernet driver. Incoming IP packets are handed to `ip_input` through a software interrupt, just as would be done normally. A filter in `ip_input` determines if the packet should be forwarded to the TCP scrubbing code. If not, then it follows the normal IP forwarding path to `ip_output`. If it is, then `isg_input` (ISG stands for Internet Scrubbing Gateway) performs IP fragment reassembly if necessary and passes the packet to `isg_tcpin`. Inside `isg_tcpin` the scrubber keeps track of the TCP connection's state. The packet is passed to `isg_forward` to perform TCP-level processing. Finally, `isg_output` modifies the next-hop link level address and `isg_output` or `ip_output` hands the packet straight to the correct device driver interface for the trusted or untrusted link.

We must also make sure that differences in the packets sent by the trusted hosts to the untrusted hosts don't reveal clues. These checks and modifications are done in `isg_forward` for TCP modifications, `isg_output` for IP modifications to TCP segments, and `ip_output` for IP modifications to non-TCP packets.

3.2.1 IP scrubbing

IP-level ambiguities arise mainly in IP header flags and fragment reassembly algorithms. Modifying flags requires no state but requires adjustment of the header checksum. Reassembly, however, requires fragments to be stored at the scrubber. Once a completed IP datagram is formed, it may need to be re-fragmented on the way out the interface.

The fingerprint scrubber uses the code in Figure 3 to normalize IP type-of-service and fragment bits in the header. This occurs for all ICMP, IGMP, UDP, TCP, and other packets for protocols built on top of IP. Uncommon and generally unused combinations of TOS bits are removed. In the case that these bits need to be used (i.e. an experimental modification to IP) this functionality could be removed. Most TCP/IP implementations we have tested ignore the reserved fragment bit and reset it to 0 if it is set, but we wanted to be safe so we mask it out explicitly. The don't fragment bit is reset if the MTU of the next link is large enough for the packet. This check is not shown in the figure.

Modifying the don't fragment bit could break MTU discovery through the scrubber. One could argue that the reason you 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

```
/*
 * Normalize IP type-of-service flags
 */
switch (ip->ip_tos)
{
    case IPTOS_LOWDELAY:
    case IPTOS_THROUGHPUT:
    case IPTOS_RELIABILITY:
    case IPTOS_MINCOST:
    case IPTOS_LOWDELAY|IPTOS_THROUGHPUT:
        break;
    default:
        ip->ip_tos = 0;
}

/*
 * Mask out reserved fragment flag.
 * The MTU of the next downstream link
 * is large enough for the packet so
 * clear the don't fragment flag.
 */
ip->ip_off &= ~(IP_RF|IP_DF);
```

Figure 3: Code fragment to normalize IP header flags.

not to clear the don't fragment bit up to the end user by allowing the option to be turned off.

The fragment reassembly code is a slightly modified version of the standard implementation in the FreeBSD 2.2.8 kernel. It keeps fragments on a set of doubly linked lists. It first calculates a hash to determine which list the fragment maps to. A linear search is done over this list to find the IP datagram the fragment goes with and its place within the datagram. Old data in the fragment queue is always chosen over new data.

3.2.2 ICMP scrubbing

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 meant to include at least the IP header plus 8 bytes of data from the packet that caused the error. According to RFC 1812 [1], as many bytes as possible, up to a total ICMP packet length of 576 bytes, are allowed. However, `nmap` takes advantage of the fact that certain operating systems quote 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.

3.2.3 TCP scrubbing

The TCP protocol scrubber we based the fingerprint scrubber on converts TCP streams into unambiguous flows by keeping a small amount of state per connection. The protocol scrubber keeps track of TCP connections using a simplified TCP state diagram. Basically, it keeps track of open connections by following the standard TCP three-way handshake (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. Only scans that commit to a 3WHS will get through.

A large amount of information can be gleaned from TCP options. 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 the end user.

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.

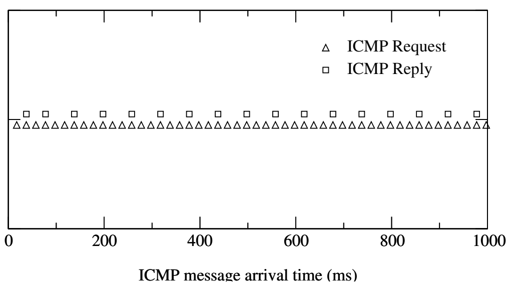

Figure 4: ICMP rate limiting of returning ICMP echo replies captured using `tcpdump`.

3.2.4 Timing attacks

The fingerprinting scans we have designed the fingerprint scrubber to block up to now have all been static, query-response style probes. A host carefully forms queries, sends them to a host, and analyzes the response or lack of response. Another possible form of scan is one that relies on timing responses. For example, the scanning host could open a TCP connection, simulate a packet loss, and watch the recovery behavior of the other host.

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 amount of delay to packets going 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. 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 because a packet retransmitted after an RTO 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 don't 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.

Figure 4 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 milliseconds 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 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 won't affect network performance.

4 Evaluation of Fingerprint Scrubber

This section presents results from a set of experiments we performed to determine the validity, throughput, and scalability 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 Pen-

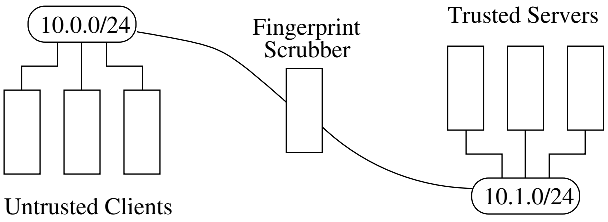

Figure 5: Experimental setup for measuring the performance of the fingerprint scrubber.

tium 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 to have all traffic from 10.0.0/24 go to 10.1.0/24 through the gateway machine. Figure 5 shows how the three machines were connected as well as the trusted and untrusted domains.

4.1 Defeating fingerprint scans

To verify that our fingerprint scrubber did indeed defeat known scan attempts, we interposed our gateway between 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. Figure 6 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

(a)

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

(b)

```
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
```

Figure 6: (a) Operating system guess before fingerprint scrubbing and (b) after fingerprint scrubbing for an `nmap` scan against a machine running FreeBSD 2.2.8.

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 potential or new scans also. 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.

4.2 Throughput

We conducted an experiment to test the raw throughput possible through the fingerprint scrubber. The throughput was measured using the `netperf` benchmark [11]. The three test machines were connected using a 100 Mbps switch.

We measured both the throughput from the trusted side out to the untrusted side and from the untrusted side into the trusted side. 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

IP Forwarding	87.06
Fingerprint Scrubbing	86.86
Fingerprint Scrub. + Frag. Reas.	87.00
Application-level Transport Proxy	86.53

Table 1: Throughput for a single untrusted host to a trusted host using TCP (Mbps, ±2.5% at 99% CI).

IP Forwarding	87.06
Fingerprint Scrubbing	86.79
Fingerprint Scrub. + Frag. Reas.	86.84
Application-level Transport Proxy	86.53

Table 2: Throughput for a single trusted host to an untrusted host using TCP (Mbps, ±2.5% at 99% CI).

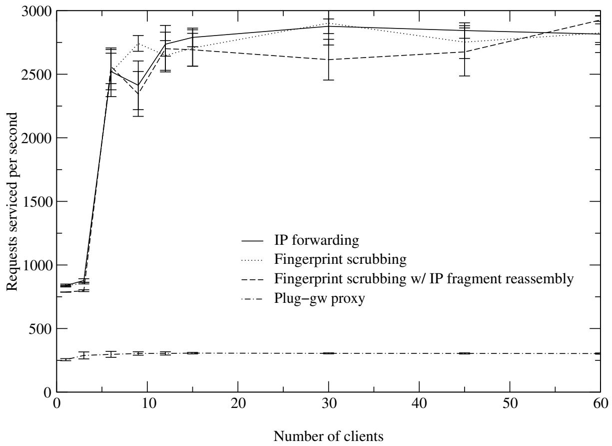

Figure 7: Connections per second through the gateway.

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.

We also compared the fingerprint scrubber to a full application-level proxy. The TIS Firewall Toolkit's `plug-gw` proxy is an example of a firewall component that operates at the user-level to do transport-layer proxying [18]. When a new TCP connection is made to the proxy, `plug-gw` creates a second connection from the proxy to the server. The proxy's only job is to read and copy data from one connection to the other. A more fully featured firewall will process the copied headers and data, which adds additional latency and requires more state. Therefore the performance of `plug-gw` represents a minimum amount of work a firewall built from application-level proxies must perform. We modified the original `plug-gw` code 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 implementation to significantly worse performance than in-kernel approaches [8; 17].

Table 1 shows the TCP bulk transfer results for an untrusted host connecting to a trusted host. Table 2 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 for all of the throughput experiments, therefore we would like to run these experiments again on a faster network in the future.

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.

Table 3 shows the UDP 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. 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

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

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

at the gateway and then re-fragmented before being sent out.

4.3 Scalability

We also ran an experiment to measure the scalability of the fingerprint scrubber. That is, how many concurrent TCP connections can our fingerprint scrubbing gateway support? We set up three machines as web servers to act as sinks for HTTP requests. On three other machines we ran increasing numbers of clients repeatedly requesting the same 1 KB file from the web servers. The choice of 1 KB allows us to keep the web servers' CPUs from being the limiting factor. Instead, the bandwidth of the link is again the bottleneck. The clients were connected with a 100 Mbps hub and the servers were connected with a 100 Mbps switch. The number of connections per second being made through the fingerprint scrubber was measured on the hub.

Figure 7 shows the number of sustained connections per second measured for plain IP forwarding, TCP/IP fingerprint scrubbing, fingerprint scrubbing with IP fragment reassembly, and the `plug-gw` application-level proxy. The error bars represent the standard deviation for each second. The results of the experiment are that the fingerprint scrubber scales comparably to the unmodified IP forwarder and performs much better than the transport proxy. The fingerprint scrubber achieves a rate of about 2,700 connections per second, which may be enough for most LANs. In comparison, the `plug-gw` proxy only achieves a rate of about 300 connections per second, which is an order of magnitude worse than the scrubber. The abysmal performance of the application-level proxy can be explained by the number of interrupts, data copies, and context switches incurred by such a user-level process. For each TCP connection, the proxy has to keep track of two complete TCP state machines and copy data up from the kernel then back down. The system running `plug-gw` was CPU bound for all but a few concurrent clients.

Achieving full line-speed in the fingerprint scrubber for higher bandwidth links would probably require dedicated hardware. A platform such as Intel's Internet Exchange Architecture (IXA) [14; 5] could help. The small size of the TCP state table we use in the fingerprint scrubber would be amenable to such a system.

5 Related Work

Current firewall technology is similar to the fingerprint scrubber [2]. Firewalls exist at the border of a network to provide access control. They both require packets to travel through them to get to their final destinations and can deny certain types of packets. Older firewalls, such as the TIS Firewall Toolkit [18], use application-level proxies and don't scale very well because they must keep two TCP connections open per session. Such a firewall will block TCP fingerprinting scans because it isolates the behavior of the TCP implementations on the side it is protecting by copying data. However, we showed that such a firewall's performance is an order of magnitude worse than the fingerprint scrubber. Also, the application-level proxy does not take care of IP-level ambiguities. Modern firewalls, such as Gauntlet [16], identify authorized flows by examining portions of packet headers and data payloads or using more sophisticated authentication methods. The firewall then routes packets through a fast path once the flow is set up to increase throughput and scalability. Such a firewall won't provide continued security against fingerprinting scans once a connection is set up. In contrast, the fingerprint scrubber removes scans throughout the lifetime of a flow while remaining more scalable by keeping a minimal amount of state per connection.

Various tools are available to secure a single machine against `nmap`'s operating system fingerprinting. The TCP/IP traffic logger `iplog` [10] can detect an `nmap` fingerprint scan and send out a packet designed to confuse the results. Other tools and operating system modifications simply use state inherently kept in the TCP implementation to drop certain scan types. However, none of these tools can be used to protect an entire network of heterogeneous systems. In addition, these methods will not work for networks that are not under single administrative control, unlike the fingerprint scrubber.

Vern Paxson presents a tool to analyze a TCP implementation's behavior called `tcpanaly` [12]. It works offline on `tcpdump` traces to try to distinguish if a certain traffic pattern is consistent with an implementation. In this way, it is doing a sort of TCP fingerprinting. However, `tcpanaly` suffers from a lot of uncertainty that makes it unfeasible as a fingerprinting tool. It also keeps explicit knowledge of several TCP/IP implementations. In contrast, our fingerprint scrubber has no knowledge of other implementations. The main contribution `tcpanaly` makes is not in fingerprinting but in analyzing the correctness of a TCP implementation and aiding in determining if an implementation has faults.

Malan, et al. [7] have presented the idea of not only transport-level scrubbing, but also application-level scrubbing. Obviously more specialization would need to be done. The main focus is on HTTP traffic to protect web servers. The idea could be extended to protect infrastructure components such as routers by scrubbing RIP, OSPF, and BGP.

6 Future Work

As mentioned in Sections 4.2 and 4.3, the first-order limiting factor in the performance of the fingerprint scrubber is the available link bandwidth. We are planning on testing the scrubber over gigabit Ethernet connections. To support a ten-fold increase in bandwidth we will be looking at reducing the amount of data copying, using incremental checksums when modifying header bits, and using a faster fragment reassembly algorithm.

Because of the close relationship between firewalls and fingerprint scrubbers, we would like to combine the two technologies. We would use the scrubber as a substrate and add features, such as authentication, required by a fully functional firewall. We believe such a system would combine the additional security benefits of a modern firewall with the performance characteristics and benefits of the fingerprint scrubber.

We would also like to examine how IP security [6] affects TCP/IP stack fingerprinting and operating system discovery. If a host implementing IPsec doesn't allow unknown hosts to establish connections, then those hosts will not be able to discern the host's operating system because all packets will be dropped. If a host does allow unknown hosts to connect in tunnel mode, however, then a fingerprint scrubber will be ineffective. The scrubber will be unable to examine and modify the encrypted and encapsulated IP and TCP headers. However, allowing any host to make a secure connection to an IPsec-enabled host is not the standard procedure unless it is a public server. Another portion of IPsec that could be exploited is the key exchange protocols, such as ISAKMP/IKE [9; 4]. If different systems have slight differences in their implementations, a scanner might be able to discern the host's operating system.

Another thing we would like to try is to have the fingerprint scrubber spoof an operating system's fingerprint instead of anonymizing it. For example, it might be interesting to have all of the computers on your network appear to be running the secure operating system OpenBSD. This is harder to do than simply removing ambiguities because you have to introduce artifacts in enough places to make the deception plausible.

As network infrastructure components increase in speed, tools such as the fingerprint scrubber must scale to meet the demand. To try to achieve line-speed, we would like to implement core components of the fingerprint scrubber in hardware. An example would be to build the minimal TCP state machine we use into a platform such as Intel's Internet Exchange Architecture (IXA) [14; 5].

7 Conclusions

We presented a new tool called a fingerprint scrubber to remove clues about the identity of an end host's operating system. We showed that the scrubber blocks known fingerprinting scans, is comparable in performance to a plain IP forwarding gateway, and is significantly more scalable than a full transport-layer firewall.

The fingerprint scrubber successfully and completely blocks known scans by removing many clues from the IP and TCP layers. Because of its general design, it should also be effective against any evolutionary enhancements to fingerprint scanners. It can protect an entire network against scans designed to profile vulnerable systems. Such scans are often the first step in an attack to gain control of exploitable computers. Once compromised these systems could be used as part of a distributed denial of service attack. By blocking the first step, the fingerprint scrubber increases the security of a heterogeneous network.

Acknowledgments

The Intel Corporation provided support for this work through a generous equipment donation and gift. This 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] F. Baker. Requirements for IP Version 4 Routers. RFC 1812, 1995.

[2] D. Brent Chapman and Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reily and Associates, Inc., 1995.

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

[4] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, November 1998.

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

[6] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[7] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and Application Protocol Scrubbing. In *Proceedings of the IEEE INFOCOM 2000 Conference*, Tel Aviv, Israel, March 2000.

[8] David Maltz and Pravin Bhagwat. TCP Splicing for Application Layer Proxy Performance. Technical Report RC 21139, IBM Research Division, March 1998.

[9] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Mangement Protocol (ISAKMP). RFC 2408, November 1998.

[10] Ryan McCabe. Iplog. `http://ojnk.sourceforge.net/`.

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

[12] Vern Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[13] Thomas H. Ptacek and Timothy 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. homepage at `http://www.nai.com/`, January 1998.

[14] David Putzolu, Sanjay Bakshi, Satyendra Yadav, and Raj Yavatkar. The Phoenix Framework: A Practical Architecture for Programmable Networks. *IEEE Communications*, 38(3):160–165, March 2000.

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

[16] PGP Security. Gauntlet Firewall. `http://www.pgp.com/asp_set/products/tns/gauntlet.asp`.

[17] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP Forwarder Performance. Technical Report TR98-01, Dept. of Computer Science, University of Arizona, February 1998.

[18] Trusted Information Systems. TIS Firewall Toolkit. `ftp://ftp.tislabs.com/pub/firewalls/toolkit`.