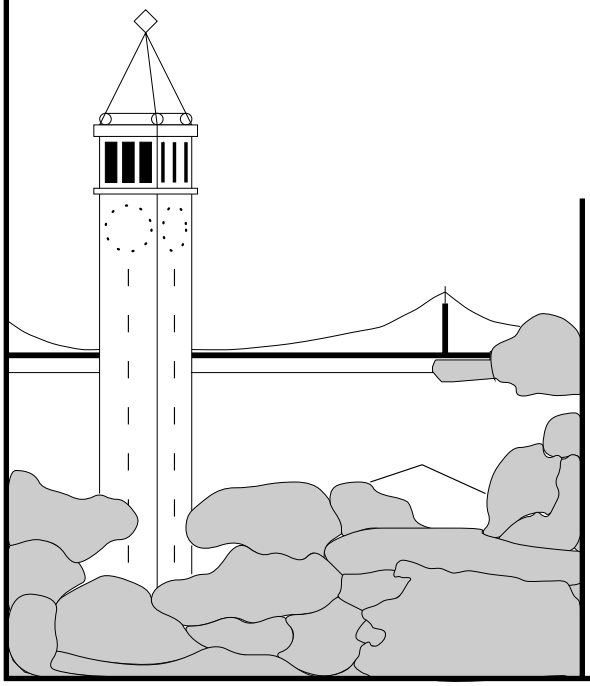
Active Mapping: Resisting NIDS Evasion Without Altering Traffic

Umesh Shankar



Report No. UCB//CSD-2-03-1246

December 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Active Mapping: Resisting NIDS Evasion Without Altering Traffic

by Umesh Shankar

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor David Wagner

Research Advisor

(Date)

* * * * * * *

Professor Scott Shenker

Second Reader

(Date)

Acknowledgments

Most of the work here was done jointly with Vern Paxson [SP02]. I have learned a tremendous amount from working him, and our discussions are always productive and often entertaining.

I would like to thank Mark Handley at ICIR and Partha Banerjee and Jay Krous in the SNS group at LBNL who helped us with testing Active Mapping on a large scale. I would also like to thank Mark Handley, David Wagner, Nikita Borisov, Rob Johnson, and Chris Karlof for their insightful and focusing comments on this paper.

Special thanks to David Wagner for being a model research advisor. He's always been available to guide me through the murky waters of doing research and his ability to cut to the heart of topics in our discussions has been invaluable.

Contents

Abstract

A critical problem faced by a Network Intrusion Detection System (NIDS) is that of *ambiguity*. The NIDS cannot always determine what traffic reaches a given host nor how that host will interpret the traffic, and attackers may exploit this ambiguity to avoid detection or cause misleading alarms. We present a novel, lightweight solution, *Active Mapping*, which eliminates TCP/IP-based ambiguity in a NIDS' analysis with minimal runtime cost. Active Mapping efficiently builds profiles of the network topology and the TCP/IP policies of hosts on the network; a NIDS may then use the host profiles to disambiguate the interpretation of the network traffic on a per-host basis. Active Mapping avoids the semantic and performance problems of *traffic normalization*, in which traffic streams are modified to remove ambiguities.

We have developed a prototype implementation of Active Mapping and modified a NIDS to use the Active Mapping-generated profile database in our tests. We found wide variation across operating systems' TCP/IP stack policies in real-world tests (about 6,700 hosts), underscoring the need for this sort of disambiguation.

We discuss the capabilities and limitations of Active Mapping in detail, including real-world challenges. We also present results on the performance impact of using Active Mapping in terms of time and memory.

Chapter 1

Introduction

1.1 The State of Network Intrusion Detection

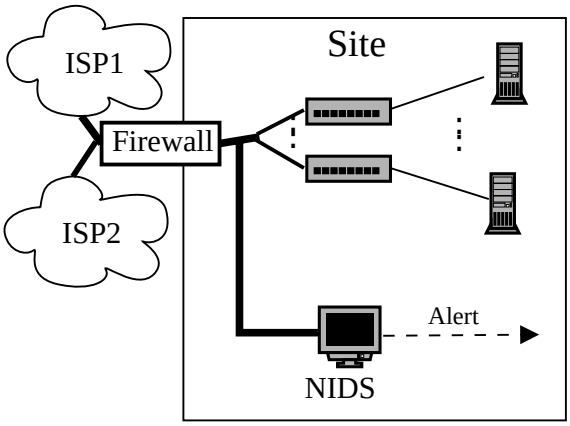


Figure 1.1: A diagram of a typical site's network with a NIDS

A Network Intrusion Detection System (NIDS) passively monitors network traffic on a link, looking for suspicious activity as defined by its protocol analyzers (see Figure 1.1). Typically this link is the external Internet link that carries packets in and out of the site being monitored[1]. Here we are only considering the (common) case where the NIDS only looks for intrusions where one end of the connection is inside the site and the other outside; inside-to-inside traffic is not examined.

The NIDS reconstructs IP packets and TCP streams flowing through the link[2]; the reconstructed data are then run though a number of *analyzers*, each of which looks for a particular pattern of undesirable activity. For example, one analyzer might look for HTTP requests corresponding to the Nimda worm, while another tries to detect port scans.

[1] Recent work has explored the possibility of distributing sensors throughout the internal network[HF99, VKB01]; while such a setup is by no means incompatible with Active Mapping, for simplicity we restrict ourselves to the former model in our discussion.

[2] Simple per-packet analysis is not uncommon, but is trivially defeated.

Current NIDS suffer from both false positives—generating spurious or irrelevant alerts—and false negatives—failing to report undesirable activity. There are many reasons for these, but a significant source of both is that analysis is usually *context-insensitive*. That is, the NIDS analyzes traffic without regard to the particular network and hosts that it is protecting. The problem is difficult to solve in general; in order to correctly analyze a stream of traffic destined for a particular host it is monitoring, the NIDS must first determine which packets actually reach the host and then, for those that do, interpret them exactly as the target host does. The problem is thus equivalent to NIDS being able to perform a complete and precise simulation of the network and the host machines. In this paper we restrict our discussion to the NIDS' ability to simulate the network and transport layers (TCP/UDP/ICMP/IP). The dominant obstacle to achieving precise simulation of even these two layers is *ambiguity*: the wide variety of network topologies and TCP/IP-stack policies makes it impossible for the NIDS to know the correct interpretation of traffic without additional context.

The result is a divergence between how a host interprets a sequence of packets and how the NIDS *believes* the sequence has been interpreted. The NIDS can be tricked by an attacker into believing that no attack occurred or may be confused by a multitude of possible interpretations, some of which are attacks and some of which are not. The evasions are not just theoretically possible: Ptacek and Newsham [PN98] describe a number of specific methods for exploiting this sort of ambiguity at the TCP/IP layer. Furthermore, toolkits have been developed which automate their use [So02, Mc98]. Thus it is of considerable practical concern that we find a way to resolve TCP/IP-based ambiguities.

1.2 Active Mapping

We have developed a novel approach to eliminating TCP/IP ambiguity, called *Active Mapping*. The key idea is to acquire sufficient knowledge about the intranet being monitored that, using it, a NIDS can tell which packets will arrive at their intended recipient and how they will be interpreted. Active Mapping does this by building up a profile database of the key properties of the hosts being monitored and the topology that connects them. Profiles are constructed by sending specially crafted packets to each host and interpreting the responses to determine path properties and TCP/IP policies (see Sections 3 and 3.4 for details).

Using Active Mapping profiles makes a NIDS *context-sensitive*. Some measure of context-sensitivity—awareness of the hosts the monitor is trying to protect—is necessary; writing more detailed analyzers is of no use when we don't know how to disambiguate the traffic we are analyzing. No amount of careful coding in the NIDS can remove context-related ambiguity. Thus, something like our approach—gathering host- and network-specific information and using it in the NIDS—is inevitable if we are to make inroads against the problem of ambiguity in a passive monitor. The information-gathering may be done in other ways, but the principle remains the same.

Previous work proposes to eliminate ambiguity in NIDS analysis by using a *traffic normalizer* [HKP01]. The normalizer, which sits in the forwarding path before the NIDS, rewrites incoming traffic into well-formed streams that presumably admit only one interpretation on all reasonable TCP/IP implementations. Thus the NIDS, with a single policy set, can unambiguously analyze the traffic for intrusion attempts on any of the hosts of the protected network.

Though it succeeds in reducing ambiguity, a normalizer, like any active (traffic-altering) element, has a

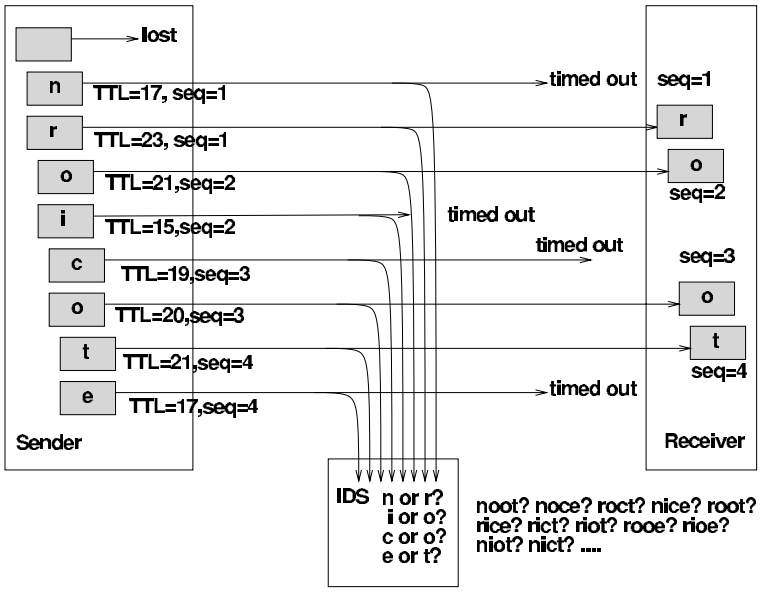


Figure 1.2: **Evading a NIDS by manipulating the TTL field [HKP01].** The NIDS is 15 hops from the sender, but the receiver is 20 hops from the sender. Packets with an initial TTL greater than 15 but less than 20 will be seen by the NIDS but will be dropped before reaching the receiver. Since the retransmitted segments are inconsistent, the NIDS does not know the correct interpretation.

number of drawbacks. One is performance: the normalizer must be able to reconstruct every TCP stream in real-time. Another is robustness: since the normalizer is in the forwarding path of every packet, it must be extremely reliable even in the face of resource exhaustion; it also must be resistant to stateholding and CPU attacks on itself. Normalization also potentially changes the semantics of a stream. As detailed in [HKP01], these changes can break some mechanisms, like *traceroute* and Path MTU discovery.

By contrast, a NIDS armed with a profile database can resolve ambiguities in a traffic stream it observes *without having to intercept or modify the stream*, which has major operational and semantic advantages. We stress that making contextual information available to the NIDS is the only way to do correct disambiguation of a stream without modifying it, so employing something like Active Mapping is essential.

Let us consider an example evasion. Figure 1.2 details an evasion based on uncertainty about the number of hops between the NIDS and a target host. If an attacker manipulates the TTL field of packets to confuse the NIDS, it can not know which of many possible packet sequences was actually received and accepted by the host. On the other hand, if the NIDS has information about the network path to the host, then it can eliminate the ambiguity. It is just this information that Active Mapping gathers and supplies to the NIDS. With it, the NIDS can ignore packets that will not reach the host, enabling correct analysis. It may be tempting, rather than gathering extra information, to try to simultaneously analyze all possible interpretations of the packet stream. However, the space of possible network topologies and TCP/IP policies is so large as to make the problem intractable (see Figure 1.2 and Section 3.4 for examples).

We have implemented a prototype of Active Mapping and run it on a network of about 6,700 hosts. Our tests showed that the increased precision in analysis does not come with any significant performance cost at runtime for the NIDS. The increased memory cost was minimal as well. We present results to this effect

in Section 5.

1.3 Roadmap

The organization of this paper is as follows. In Section 2, we discuss a model of operation of the mapper. In Section 3, we discuss the abilities and limitations of Active Mapping, examining selected tests in detail. The mapper's implementation is described in Section 4; the results of mapping real-world networks and NIDS integration tests are presented in Section 5 along with a discussion of performance and findings. We give an overview of related work in Section 6, including the potentially symbiotic relationship between Active Mapping and normalization, and conclude with a summary of our findings in Section 7. In Section 3.4, we make an effort to cover the complete spectrum of TCP/IP mappings.

Chapter 2

The Design of Active Mapping

2.1 Assumptions

In order to perform mapping efficiently, we make certain assumptions about the nature of the network being monitored:

- Network topology is relatively stable. We discuss how often mapping may be performed (based on the prototype mapper's performance) in Sections 5.2 and 5.5.
- The attacker is outside the network; if there is collusion with a user on the inside, there is little any system can do. Malicious insiders working alone are assumed to be unable to change or drop particular packets. This latter assumption is more likely to be true for switched networks.
- There is a firewall that can be used for simple packet-level filtering, especially address-based ingress and egress filtering to prevent spoofing. Also, we assume the NIDS' tap is inside the firewall.
- Hosts' TCP/IP stacks behave consistently within ordinary parameters: that is, if they exhibit unusual behavior, it will be at boundary cases. We do not, for example, run every TCP mapping test at every possible sequence number.

2.2 Design Goals

We have been guided by a number of design principles in constructing our system:

- **Comparable runtime performance.** The use of Active Mapping profiles should not appreciably slow down the NIDS nor significantly increase its memory requirements.
- **Mapping should be lightweight.** The bandwidth consumed by mapping packets should be small enough not to disrupt ordinary traffic on the network nor disrupt the operation of the host being mapped. The process of mapping should also be completed in a modest amount of wall-clock time.

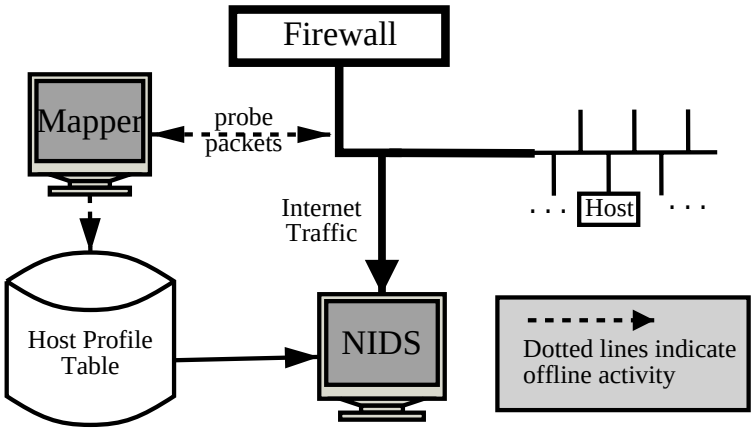


Figure 2.1: **Interaction between the NIDS and the Mapper.** The Active Mapping system sends specially crafted packets to each host to determine the hop count, Path MTU, and TCP/IP stack policies. The results are combined into a profile. The NIDS uses the profiles to correctly interpret each packet going to one of the hosts.

- **Avoid harming the hosts.** While no intentionally malicious packets are sent, bugs in a host's TCP/IP implementation might be triggered by the (unusual) mapping packets. Each test should be checked against known vulnerabilities before being deployed.

2.3 Architecture

Our overall strategy is as follows: independent of the NIDS, an Active Mapping machine scans each host on the internal network, building for each host a profile of the network path and TCP/IP stack properties. These profiles are stored in a database. At runtime, a NIDS can use the database to resolve potential ambiguities (see Figure 2.1).

For example, the NIDS can use the host profiles to decide whether to accept or discard SYN data in TCP packets, depending on the policy of the host for which the packet is destined. We note that although our method allows many ambiguities to be resolved, sometimes the very presence of ambiguity may be indicative of noteworthy behavior (an attack attempt, for example). Thus a NIDS may want to retain the ability to notify the administrator of suspicious behavior, even when the interpretation of that behavior is clear.

Our mapping tool is intended to be run on a machine whose network point is topologically equivalent to the link the NIDS is watching (see Figure 2.1). Typically this is between the firewall and any internal routers. It is important that the mapper be able to send packets on this link, since the mapping results should reflect the path of actual packets from the point of view of the NIDS. In order to keep the NIDS from interpreting mapping traffic as attacks on internal hosts, the NIDS should be configured to ignore traffic to and from the mapping machine.

The mapper begins mapping a host by performing service discovery and hop count and Path MTU (PMTU)

determination. It initiates TCP connections to a set of TCP services and sends ICMP echo packets to determine which services are available. Subsequent testing is abstracted within the mapper to use services that are available. For example, some tests require determining whether some particular TCP data are accepted by the receiver. To do so, we can use any service for which we can tell by its response whether it received the specific data. To date, we have developed concrete implementations of SSH and HTTP beneath this abstraction. Other tests require simpler abstract interfaces. PMTU determination, for example, is done with ICMP echo if available, or any TCP service failing that. The hop count and PMTU of the path to each host are determined next (Section 3.3). Once these basic properties have been established, we conduct a variety of IP and TCP tests (Section 3.3), generating the host profiles. Each test is repeated multiple times to confirm the result and account for timeouts and late or duplicated packets. To reduce correlated failures, we never run more than one instance of a (host, test) pair at the same time.

Many TCP and IP tests yield a binary answer (does the host accept a certain type of packet or not?); virtually all choose one of a small number of possibilities. Thus the search space is quite manageable and makes the resulting profiles small. Some tests with more than two possible policies require multiple rounds of probing or multiple connections to get the requisite number of bits of information. We give an example of such a test in 3.3.

Chapter 3

Active Mapping: Details and Limitations

3.1 Completeness

To thoroughly analyze the applicability of Active Mapping to resolving possible ambiguities, we follow the "header-walking" technique used in [HKP01]. We feel this is a good place to start, since the authors of that paper used a systematic approach to enumerate possible ambiguities in TCP/IP streams. We examine each ambiguity to see if it can be resolved via Active Mapping, or, if it is very simple, if it can be handled by stateless firewall rules (see Section 3.2). Thus, we provide a reasonably complete picture of the ability of Active Mapping to eliminate TCP/IP ambiguity.

A summary of the Active Mapping approach to every normalization in [HKP01] is in Section 3.4. To date, we have implemented a selected subset of these mappings, spanning a number of different types of ambiguities. We discuss them in detail in Section 3.3. Many of the mappings in the full list are straightforward additions given the types we have already implemented.

Active Mapping has some additional concerns beyond those for normalization (mostly regarding timing, since normalized traffic is generally not subject to timeouts). We discuss these and other cases that are not easily tackled using our approach in Section 3.5. A discussion of some practical concerns such as NATs and DHCP follows in Section 3.6.

3.2 Firewall Filters

Certain simple cases should be handled by stateless packet filtering at the firewall:

- Verifying IP header checksums
- Ingress and egress filtering, i.e., accepting external and internal source addresses only on those respective interfaces

- Blocking packets to broadcast addresses or to reserved private address spaces
- Rejecting packets whose IP header length field is too small or too large

In short, the firewall should reject packets that could not be part of legitimate traffic or that are so malformed as to be useless to an endhost.

3.3 Selected Mappings

Hop Count. Knowing the number of hops to an end host allows us to resist the evasion in Figure 1.2. The easiest way to determine hop count is to use the *traceroute* utility. However, its strategy of sending three packets per hop, increasing the search radius one hop at a time, is quite time-consuming. To speed this up, we instead use a known service on the system and send a packet to it that is expected to elicit a response. Most hosts will set the initial TTL value to 2^N or $2^N - 1$ for $5 \leq N \leq 8$. Thus from the response we can make a good first guess of the number of hops G by subtracting the TTL in the response packet from the next highest value of $2^N + 1$. This guess could be wrong if the routing is asymmetric or the host happens to use a different initial value. Our search strategy is therefore to test the range $G - 4$ to G, then if that fails to yield a result, perform a binary search.

PMTU. Knowing the Path MTU—the Maximum Transmission Unit over the path to the host—is important because packets greater than this size with the DF (Don't Fragment) bit set are discarded. To determine it, we send packets of various sizes to known services on the hosts with the DF bit set and watch for responses.[1] As with hop count determination, we optimize for the common case. Since many internal networks run on Ethernet, which has an MTU size of 1500 bytes, we test this first, then do a binary search on [0, 1500], since the mapper's MTU will generally be limited by its local Ethernet interface to 1500.

TCP RST Acceptance. RFC 793 [Po81c] specifies that a TCP RST packet is to be accepted if and only if it is within the receiver's window. A non-compliant TCP could create problems for a NIDS, which would not know if the connection had been terminated or not.

Repeat the following steps with the offset O set equal to 0 (in sequence), 1 (in the window), and W+small constant (outside the window):

- Send a TCP SYN packet at sequence number S.
- Receive a SYN/ACK packet, including a window size W.
- Send an ACK packet to establish the connection.
- Send RST packet at $S + O$.

[1] In principle, we could also watch for ICMP Needs Fragmentation responses, some of which indicate the limiting MTU size at the router generating the response. But it is simpler for us to directly assess PMTU end-to-end.

- Send FIN packet in sequence, i.e., at S.
- Receive one of: ACK of FIN packet $\rightarrow$ RST not accepted; RST or nothing[2] $\rightarrow$ RST accepted.

Overlapping and Inconsistent IP Fragments. RFC 791 [Po81a] states, "In the case that two or more fragments contain the same data either identically or through a partial overlap, this [suggested] procedure will use the more recently arrived copy in the data buffer and datagram delivered." It does not talk about inconsistent or overlapping fragments. Furthermore, employing the suggested policy has security implications: firewalls and other packet filters must reassemble packets before making any decisions about them, since at any point, a new fragment can overwrite data from an old one. It is therefore no surprise that there are many different implementations in use.

We perform fragment reassembly testing using ICMP echo packets; in principle the test could be performed with TCP packets as well. We send a sequence of fragments, each one containing a multiple-of-eight byte payload (since the IP offset field is in those units). The diagram below shows each of the six fragments, numbered by the order in which they were sent; their payloads consisted of that number replicated a multiple-of-eight number of times. For example, the third fragment was sent at an IP offset of 6 (corresponding to the 48th octet in the overall packet) and had a 24-byte payload of the repeating character '3'. Each fragment but the last had the MF (More Fragments) bit set. The fragments' offsets and the host's possible interpretations are given below, along with the names of the policies to which they correspond:[3]

```
012345678911 --> higher IP Offset

        Data Sent
111 22333    (Fragments 1,2,3)
 4444 555666 (Fragments 4,5,6)

      Data Received
111442333666 BSD policy
144422555666 BSD-right policy
111442555666 Linux policy
111422333666 First policy
144442555666 Last/RFC791 policy
```

The following is a description of the policies we have observed so far:

BSD. This policy left-trims an incoming fragment to existing fragments with a lower or equal offset, discarding it if it is overlapped entirely by existing fragments. All remaining octets are accepted; overlapping fragments with a greater offset are discarded or trimmed accordingly. This policy is documented more thoroughly in Wright and Stevens [WS95], pp. 293-296.

[2] Note that although a RST *should* be generated in response to the FIN if the initial RST was accepted, some hosts have firewalling software that will not respond to packets not sent to open connections (so as to leak as little information as possible). Thus we equate receiving no response (within 5 seconds) to an acceptance of the RST we sent.

[3] One unfortunate problem is that for the ICMP checksum to be correct, we must calculate it assuming a particular reassembly policy! Thus we must send all the fragments (with a different checksum) once for each policy.

BSD-right. This policy is similar to BSD, except fragments are right-trimmed (new fragments take precedence over those with a lower or equal offset).

Linux. The Linux policy is almost the same as the BSD policy, except that incoming fragments are trimmed only to existing fragments with a strictly lower offset; that is, existing fragments with the same offset will be overwritten, at least in part.

First. Always accept the first value received for each offset in the packet.

Last/RFC791. Always take the last value received for each offset in the packet.[4]

Other. Three other possible policies are tested for, but none have yet been observed in practice.

Overlapping and Inconsistent TCP segments. This problem is similar to that of IP fragment reassembly. RFC793 [Po81c] states that an implementation should trim segments "to contain only new data", which implies a "First" policy. The principle for testing is likewise similar to evaluating fragment reassembly ambiguities, and we could do the mapping using any TCP service for which we can conduct an application-level dialog. Ideally we would use the TCP Echo service, but this is rarely supported; we used SSH and HTTP in testing. We discuss it here as implemented for SSH.

Upon connecting, an SSH server sends a version string of the form
`SSH-<major>.<minor>-<comments>\r\n` [Yl02].
The client is expected to send a similar string of its own; if the string is well-formed, the server responds with additional parameters for negotiation. If not well-formed, the server closes the connection, optionally sending an error message.

Our test makes the well-formedness of the version string dependent on the reassembly policy. By sending different combinations of segments, we can deduce the policy from the varied responses. For each of the following two tests, some hosts will reassemble the following legal version string
`SSH-2.0-blah\r\n`[5]
and some will reassemble an illegal version string, upon which they will end the connection. Thus we can tell by the success or failure of the connection whether a legal string was reassembled or not.

The first test sends the following three segments. Only policies that do not left-trim (or indiscriminately trim) to earlier data will fail.

```
012346789012      TCP Seq. Offset
 SH-              (First segment)
   X2.0-blah\r\n  (Second segment)
S                 (Third segment)
```

Note that the initial 'S' is sent last to prevent reassembly until it is sent.

[4] In testing, some Cisco routers (which employed the Last policy) sent back a response with several additional trailing NUL characters.

[5] Actually, the version string the mapper sends to the server is the same as the one the server initially sends to the mapper. This prevents "protocol mismatch" errors.

The second sends four segments; this test tries to further discriminate among policies that succeeded on the first test. Policies which never discard already-received data will fail this test.

```
012346789012      TCP Seq. Offset
 SH               (First segment)
       +          (Second segment)
  X-2.0-blah\r\n  (Third segment)
S                 (Fourth segment)
```

Here there are three observed policies, characterized by the success (connection) of the (first, second) test. They are the same as for IP fragments: BSD (yes, yes), first (yes, no), and last (no, no). The fourth possibility (no, yes), has not yet been detected in our testing. Observed results by operating system may be found in Section 5.

3.4 Full List of Active Mappings

In [HKP01], the authors adopted a "header-walking" technique—inspection of each TCP and IP header field—in an attempt to enumerate all ambiguities (which would then be resolved using a normalizer). In our analysis of Active Mapping as an approach to the same problem, we borrow that work's list of normalizations, noting for each how it fits into the Active Mapping framework. The idea is to try to get a complete picture of how Active Mapping can (or can't) eliminate possible TCP/IP ambiguities by looking at each, then stating what sort of mapping technique would work. The reader is referred to [HKP01] for more thorough explanations of some of the normalizations. We note that we have not implemented all of the mappings suggested below; nonetheless, most are straightforward given the ones that we have implemented and tested.

The Disposition column in the tables below will usually contain one of three main approaches, sometimes coupled with a short explanation:

Drop. The stateless firewall should be configured to drop this packet.

Map. We can send chosen probe packets to the host to determine its policy. The most common case, "Map for drop," indicates that the packet should be sent to a host—usually as part of an open connection—to see whether it is dropped or acknowledged.

Ignore. We do not need to perform any mapping for this test.

There is a tradeoff between accepting malformed packets that might be useful and allowing in malicious traffic. For some normalizations, a choice should be made about whether the anomaly in question might (or empirically does) arise in normal traffic. If it is decided that the anomalous packet would not arise normally, it may be dropped by a firewall or a partial normalizer running in front of the NIDS.

IP Normalizations

#	IP Field	Normalization Performed	Disposition
1	Version	Non-IPv4 packets dropped.	Drop if NIDS is not IPv6-aware, else Ignore.
2	Header Len	Drop if hdr_len too small.	Drop.
3	Header Len	Drop if hdr_len too large.	Drop.
4	Diffserv	Clear field.	Ignore if internal routers don't support; add Diffserv policy to NIDS otherwise
5	ECT	Clear field.	Map for drop.
6	Total Len	Drop if tot_len $>$ link layer len.	Drop.
7	Total Len	Trim if tot_len $<$ link layer len.	Ignore.
8	IP Identifier	Encrypt ID.	Ignore.
9	Protocol	Enforce specific protocols.	Ignore unless the NIDS is aware of any other protocol.
–	Protocol	Pass packet to TCP,UDP,ICMP handlers.	N/A (done by NIDS)
10	Frag offset	Reassemble fragmented packets.	Map (see § 3.3).
11	Frag offset	Drop if offset + len $>$ 64KB.	Map to see if data $>$ 64k is accepted or trimmed off, but don't trigger known bugs.
12	DF	Clear DF.	Map PMTU (see § 3.3).
13	DF	Drop if DF set and offset $>$ 0.	Map for drop. One plausible interpretation is: do not *further* fragment this packet. Some Solaris machines generate these packets; it is not disallowed by RFC791 [Po81a].
14	Zero flag	Clear.	Firewall should clear if possible; otherwise Map to see if packets with zero flag set are dropped.
15	Src addr	Drop if class D or E.	Drop.
16	Src addr	Drop if MSByte=127 or 0.	Drop.
17	Src addr	Drop if 255.255.255.255.	Drop.
18	Dst addr	Drop if class E.	Drop.
19	Dst addr	Drop if MSByte=127 or 0.	Drop.
20	Dst addr	Drop if 255.255.255.255.	Drop.
21	TTL	Raise TTL to configured value.	Map (see § 3.3).
22	Checksum	Verify, drop if incorrect.	Drop or optionally Map for drop.
23	IP options	Remove IP options.	Map for drop (esp. source route/record route); add support for IP options to packet processing on NIDS. Optionally have router or partial normalizer clear unsupported options (packets already taking slow path on router).
24	IP options	Zero padding bytes.	Ignore. Optionally have router clear padding bytes.

UDP Normalizations

#	UDP Field	Normalization Performed	Disposition
1	Length	Drop if doesn't match length as indicated by IP total length.	Map: assume minimum of UDP or IP length taken. Also map for drop. Optionally drop if router supports it.
2	Checksum	Verify, drop if incorrect.	Map for drop. Optionally just Drop if router supports it.

TCP Normalizations

#	TCP Field	Normalization Performed	Disposition
1	Seq Num	Enforce data consistency in retransmitted segments.	Map (see § 3.3).
2	Seq Num	Trim data to window.	Map: send out-of-window segment, then segments in reverse to start of window to prevent stream reassembly until all segments have been received; check ACK sequence point.
3	Seq Num	Cold-start: trim to keep-alive.	If NIDS can send packets, send keep-alive (incorrect ACK designed to elicit the current sequence point in an ACK from the internal host). Otherwise Ignore: this is a cold-start problem.
4	Ack Num	Drop ACK above sequence hole.	Map to see if the ACK is accepted.
5	SYN	Remove data if SYN=1.	Map for drop; if not, see if data is ACKed.
6	SYN	If SYN=1 & RST=1, drop.	Map to see if RST accepted during open connection; Map to see if SYN accepted if no connection established.
7	SYN	If SYN=1 & FIN=1, clear FIN.	See if FIN is ACKed; the sender could plausibly say, "I want to initiate a connection, but have nothing to send," making the connection half-open right away.
8	SYN	If SYN=0 & ACK=0 & RST=0, drop.	Map for drop or optionally Drop.
9	RST	Remove data if RST=1.	Ignore: there are no known exploits. Optionally use normalizer to remove data.
10	RST	Make RST reliable.	If possible, have NIDS send-keep alive to ensure that RST was accepted (reliable RST).
11	RST	Drop if not in window.	Map (see Section 3.3)
12	FIN	If FIN=1 & ACK=0, drop.	Map for drop.
13	PUSH	If PUSH=1 & ACK=0, drop.	Map for drop.
14	Header Len	Drop if less than 5.	Map for drop.
15	Header Len	Drop if beyond end of packet.	Map for drop.
16	Reserved	Clear.	Ignore or optionally Map for drop.

#	TCP Field	Normalization Performed	Disposition
17	ECE, CWR	Optionally clear.	Ignore.
18	ECE, CWR	Clear if not negotiated.	Ignore.
19	Window	Remove window withdrawals.	Map for drop.
20	Checksum	Verify, drop if incorrect.	Map for drop.
21	URG,urgent	Zero urgent if URG not set.	Ignore. Optionally use app-level host information (e.g., particular HTTP server) to interpret urgent data.
22	URG,urgent	Zero if urgent > end of packet.	As above. Note that it is legal for the urgent pointer to point beyond of the packet containing it.
23	URG	If URG=1 & ACK=0, drop.	Map for drop.
24	MSS option	If SYN=0, remove option.	Map to see if the option actually changes the MSS in this case.
25	MSS option	Cache option, trim data to MSS.	The NIDS should do the caching.
26	WS option	If SYN=0, remove option.	Ignore: Window scaling presents a cold-start problem; if desired, partial normalization can remove the option or else the NIDS can try to infer its success from subsequent ACKs.
27	SACK	Normalizations 27-31 regarding SACK	Ignore: SACKs are advisory, so should not affect the semantics the NIDS uses.
32	T/TCP opts	Remove if NIDS doesn't support.	Map for drop.
33	T/TCP opts	Remove if under attack.	N/A
34	TS option	Remove from non-SYN if not negotiated in SYN.	Map for drop.
35	TS option	If packet fails PAWS test, drop.	Map for drop.
36	TS option	If echoed timestamp wasn't previously sent, drop.	Map for drop.
37	MD5 option	If MD5 used in SYN, drop non-SYN packets without it.	Map for drop when option not set in SYN. If not dropped, do same thing in NIDS as in normalizer, but this causes a cold-start problem.
38	*other opts*	Remove options	Ignore: optionally remove with a partial normalizer. (See Section 3.5).

ICMP Normalizations

#	ICMP Type	Normalization Performed	Disposition
1	Echo request	Drop if destination is a multicast or broadcast address.	Optionally Drop.
2	Echo request	Optionally drop if ping checksum incorrect.	Optionally Drop.
3	Echo request	Zero "code" field.	Map for drop.
4	Echo reply	Optionally drop if ping checksum incorrect.	Optionally drop.

#	ICMP Type	Normalization Performed	Disposition
5	Echo reply	Drop if no matching request.	Ignore.
6	Echo reply	Zero “code” field.	Map for drop.
7	Source quench	Optionally drop to prevent DoS.	Optionally Drop.
8	Destination Unreachable	Unscramble embedded scrambled IP identifier.	Ignore: IP identifiers not scrambled without normalizer.
9	*other*	Drop.	Optionally Drop depending on NIDS policy.

3.5 Difficult or Intractable Cases

The success of Active Mapping depends upon hosts’ behaving in a consistent and predictable way. This is generally a good assumption, since most protocol stacks are deterministic and obey relatively simple rules for ambiguity resolution. There are, however, at least three sources of nondeterminism that can make it difficult to perform precise simulation in the NIDS, even with Active Mapping: user-controlled parameters in the TCP stack, new semantics, and non-deterministic packet drops.

Application-level Parameters. Users can change certain parameters that affect the TCP/IP stack. One example, as noted in [HKP01], is the use of the TCP “urgent” pointer, which marks some part of the sequence space as containing important data that should be processed without delay. Depending on the implementation and user-set parameters, this data may be delivered via a signal or inline to the user process. There is no way for the NIDS to determine unambiguously the reconstructed byte stream as seen by the application without help from the host or hardcoding of the application’s interpretation of urgent data.

New semantics. A NIDS must understand the intended semantics of a stream if it is to interpret it correctly. Unknown TCP options, for example, can be ignored if the target host does not indicate support for them. The best the NIDS can do in general is to be updated regularly with support for new options as hosts on the internal network support them. If partial normalization (see Section 6.1) is available, unsupported options can be filtered out.

Nondeterministic Packet Drops. Perhaps the most common reason for packet drops is a full incoming packet buffer at an internal router or endhost. Thus if routers internal to a site become saturated, or if a particular host is facing very high traffic volumes, packets may be dropped. If an attacker can cause packets to be dropped in a very precise way during mapping, that could affect mapping results; less precise interference is likely to be caught as an inconsistency between multiple runs.

Dropping may also be done by routers to meet Quality of Service guarantees. Mechanisms like Diffserv [B+99] that implement QoS but whose exact workings are site-specific are hard to predict, since external and internal traffic may be mingled, each contributing to packet drops for the other. A mitigating factor is that such QoS policies tend to be implemented either at the boundary routers (which filter before the NIDS) or at an external aggregation point.

The NIDS must also know when a host will timeout an IP fragment or TCP segment. Without this knowledge, an attacker can later retransmit the fragment or segment with different data: the NIDS cannot know which was accepted, even with knowledge about which would be accepted if the first did not time out. Though Active Mapping can try to deduce the timeout value, the need for precision in the timeout determination makes this difficult.

3.5.1 Dealing with Timeouts and Packet Drops

The NIDS cannot be notified of every router or end host packet drop. The host being monitored does, however, give some implicit drop information, in the form of acknowledgments and responses to requests or lack thereof. When combined with temporal causality, this can allow partial reconstruction of the host's state.

If we see an acknowledgment of a TCP segment or a response to a UDP or ICMP request, we can infer that the request must have been accepted using only packets that preceded the response. Furthermore, if no response is sent when one is expected, we can infer that packets have been dropped. If the NIDS can send packets in real time, it can send a "keep-alive" TCP packet, one that is out of sequence. This should elicit an ACK that shows the current sequence number.

The NIDS can also watch for ICMP messages indicating timeouts ("Fragment Reassembly Time Exceeded," per [Po81b]). Not all hosts send these notifications, and they might leak information to an attacker. A compromise might be to configure hosts to generate informative ICMP messages that are filtered by the firewall (but are still seen by the NIDS).

3.6 Practical Considerations

There are additional concerns that arise in mapping real networks. Our initial prototype does not handle all these cases and there are likely to be others. We discuss possible approaches to common real-world scenarios below. We point out that Active Mapping does not require a complete profile for each host to be useful: at best, many ambiguities are eliminated; at worst, the default behavior is that of the original NIDS. Thus Active Mapping may be incrementally deployed even while some practical hurdles are being surmounted.

NAT Network Address Translation (NAT) [EF94] is a common scheme for conserving public IP addresses and for protecting clients against breakins. Briefly, the address translator assigns private IP addresses to a group of clients. When a client initiates a connection to an outside server, the translator maps the client's source port to a source port allocated on itself, rewriting packets so that servers see the translator as the client. Servers may be supported behind NAT by statically mapping certain ports on the translator (say port 80) to port 80 on a specific NAT client running an HTTP server.

So far our discussion of mapping has assumed that each IP address corresponded to exactly one machine (and a single set of policies). If a NAT [EF94] is running inside the monitored site (so that the NIDS does

not see the private addresses), however, we need additional strategies. To handle servers behind a NAT, we could map each port as though it belonged to a separate machine, checking for all relevant policies on each port. It is harder to deal with clients behind a NAT, though this is only relevant in the case of outside servers attacking internal clients in a client OS-specific way.

It can be difficult to detect when a NAT is being used, though recent work by Bellovin [Be02] suggests that it is possible in some cases. If not all NAT IPs are not known to system administrators, the mapper could map multiple ports independently or sample them for differences, which would indicate a NAT's presence.

DHCP The Dynamic Host Configuration Protocol (DHCP) [Dr97] dynamically assigns IP addresses to clients. A DHCP server leases out addresses when clients request them; leases expire periodically and must be renewed. A typical use of DHCP is to assign IP addresses to laptops that are periodically connected to the network. Dealing with DHCP requires some integration: the mapper could be triggered upon seeing DHCP requests (if the broadcast does not make it to the mapping machine, the DHCP server can be set up to notify it). The profile database could include MAC addresses, so the mapper would know when it already has a profile for a given machine (perhaps gathered previously under a different IP address). If integration with a DHCP server is not possible, determining MAC addresses might be nontrivial; it is an area for future work.

TCP Wrappers (Host-Based Access Control) Some hosts use *TCP Wrappers* to restrict access to services to a set of hosts determined by an Access Control List. If the Active Mapping machine is not granted access, some tests requiring meaningful interaction with a particular TCP service will fail. A simple solution is to allow a designated mapping machine access to relevant services.

Attacks on the Active Mapper A natural concern is whether or not an attacker could subvert the mapping process, causing false results, by attacking the mapping machine or trying to change mapping traffic. Preventing outsider attacks on the mapper directly is straightforward: simply have the firewall reject all traffic destined for the mapper. There is no legitimate need for direct access to a mapping machine from the outside. A greater concern would be direct attacks from inside machines that have been compromised; the threat could be mitigated by only allowing access to well-known ports from a restricted set of administrative machines at the mapper's local router. As for traffic modification, it is certainly the case that if an attacker can drop or modify packets used in Active Mapping, he can change the results. As an outsider, at best an attacker could hope to force packet drops by flooding. This would be hard to achieve in a switched network, and an attacker would have to cause consistent drops to prevent the mapper from detecting differences across multiple trials. An insider trying to drop packets would have more success, but again, the damage would likely be confined to the machines in the attacker's subnet since considerable precision is needed (and having a switched network would make it even more difficult).

Chapter 4

Prototype Implementation

We implemented Active Mapping in about 2,000 lines of Perl and have ported it to the Linux and FreeBSD operating systems. It requires a TCP/IP firewall capability, the libpcap packet capture library [MLJ94], and raw socket support. Using these features generally requires superuser access.

ICMP and TCP packets are sent directly using raw sockets. A Pcap filter is set up to capture responses. Our user-level TCP implementation follows a strategy similar to that of *Tbit* [PF01], a TCP behavior-inference tool. Like *Tbit*, we firewall off high-numbered TCP ports for use as ephemeral source ports (to prevent the kernel from responding to incoming traffic to those ports by sending RSTs). Unlike *Tbit*, which dynamically installs and removes firewall filters, we require the user to allocate and firewall off a range of ports in advance; this reduces the amount of system-dependent code in the mapper at the expense of transparency. Our TCP implementation is rudimentary; currently we perform neither reassembly nor implement congestion control, for example. Nonetheless, it has proved adequate thus far for the short-lived connections needed for mapping, especially since servers tend to send back well-formed replies to our often malformed queries.

The mapper conducts tests in parallel with respect to machines being mapped and with respect to each individual test. The degree of parallelism is determined by the number of available TCP source ports, the size of the packet buffers, and (due in particular to our unoptimized implementation) the CPU speed. In our initial tests, we suffered considerable packet loss. This was found to be due in part to small packet capture buffers, but mostly to the inability of the Perl packet handling library to keep up with the traffic. In response we implemented rate-limiting in the mapper, which gave us almost perfect consistency in results. We expect that in a faster implementation, the rate could be increased considerably.

Each test is repeated a configurable number of times (three, in testing) and all the results are recorded. This is important to account for dropped packets and timeouts.

Currently, we have implemented network topology and service discovery as well as the specific tests described in Section 3.3.

We modified the Bro NIDS to use Active Mapping profiles to properly interpret traffic. [Pa98].[1] The

[1]We note that this integration may be done with any NIDS which does TCP/IP stream reconstruction, since it will include all

integration was straightforward; a few hundred lines of C++ code were needed. In addition to the code needed to read and parse the profile database, we had to insert checks at each policy-dependent decision point (e.g., fragment reassembly) and, where necessary, implement additional policies. Thus, of the modest amount of time making these changes required, the majority was spent identifying the locations in the code requiring a policy-dependent check. The performance impact of the modifications is discussed in the following section.

the necessary decision points.

Chapter 5

Experiments and Results

5.1 Observed Active Mapping Profiles

We ran the prototype Active Mapper at Lawrence Berkeley National Lab. The exact number of active hosts during our scan is not known, but was estimated based on other scans to be around 6,700. We obtained nontrivial, consistent data (identical results over three trials for something other than the hostname) for just over 4,800 hosts. Many of the IPs for which we did not obtain results are in DHCP blocks (hosts may not always be present); in addition, many users employ host-based firewalls which would prevent our scanning. We are currently working on getting more precise data here (we note that firewalls are likely to prevent the attacks the NIDS is looking for in any case!). It is significant that we obtained results for virtually every machine for which OS data were known; presumably most other machines are more transient or are firewalled enough to stop OS detection. Some tests did not yield results due to services' being protected with TCP Wrappers. We expect this limitation can be overcome in practice by adding the mapping machine to the hosts' ACLs as needed.

We present Active Mapping profiles by operating system in Figure 5.1. The amount of observed diversity in policy is remarkable, given that we only ran five tests. While hosts with a given operating system version exhibited the same policies, it is interesting note how policies changed for different versions of the same operating system. Linux in particular seems to have undergone a number of policy changes, even during minor revisions of the kernel. We also note that individual users can alter policies by installing "hardening" or other patches. It is precisely this diversity (borne out by our experiments) that underscores the need to disambiguate traffic destined for each host based its particular observed policy.[1]

For 173 hosts, we were unable to get results that were consistent (defined as getting identical results for three trials). This is less surprising, perhaps, in light of the fact that all but 29 of them were found to be printers, routers, or scanners (many of the remaining had unknown operating systems). Of the 173, all but 36 gave consistent results when a result was obtained, but had a trial which did not complete. This could be due to congestion. In all, only 10 machines which were not known to be special-purpose devices yielded results with conflicting answers.

[1] We note that a first-order approximation might be obtained by using known OS version information with a lookup table; it may even make sense to run Active Mapping and then infer the OS from its results. We plan to investigate this relationship in the future.

OS	IP Frag	TCP Seg	RST in wnd	RST outside wnd
AIX 2	BSD	BSD	Yes	No
AIX 4.3 8.9.3	BSD	BSD	Yes	No
Cisco IOS	Last	BSD	Yes	No
FreeBSD	BSD	BSD	Yes	No
HP JetDirect (printer)	BSD-right	BSD	Yes	No
HP-UX B.10.20	BSD	BSD	Yes	No
HP-UX 11.00	First	BSD	Yes	Yes
IRIX 4.0.5F	BSD	No result	Yes	No
IRIX 6.2	BSD	No result	Yes	No
IRIX 6.3	BSD	BSD	Yes	No
IRIX64 6.4	BSD	BSD	Yes	No
Linux 2.2.10	linux	No result	No	No
Linux 2.2.14-5.0	linux	BSD	Yes	No
Linux 2.2.16-3	linux	BSD	No	No
Linux 2.2.19-6.2.10smp	linux	BSD	No	No
Linux 2.4.7-10	linux	BSD	Yes	No
Linux 2.4.9-31SGI_XFS_1.0.2smp	linux	BSD	Yes	No
Linux 2.4 (RedHat 7.1-7.3)	linux	BSD	Yes	No
MacOS (version unknown)	First	BSD	Yes	Yes
netapp unknown	No result	No result	No	No
netapp unknown	No result	No result	Yes	No
NCD Thin Clients (no services exported)	BSD	No result	No result	No result
OpenBSD (version unknown)	linux	BSD	Yes	No
OpenBSD (version unknown)	linux	BSD	No	No
OpenVMS 7.1	BSD	BSD	Yes	No
OS/2 (version unknown)	BSD	No result	Yes	Yes
OS/2 (version unknown)	No result	No result	No	No
OSF1 V3.0	BSD	BSD	Yes	No
OSF1 V3.2	BSD	No result	Yes	No
OSF1 V4.0,5.0,5.1	BSD	BSD	Yes	No
SunOS 4.1.4	BSD	BSD	Yes	No
SunOS 5.5.1,5.6,5.7,5.8	First	Last	Yes	No
Tektronix Phaser Printer (unknown model)	Last	No result	No	No
Tektronix Phaser Printer (unknown model)	First	BSD	Yes	Yes
Tru64 Unix V5.0A,V5.1	BSD	BSD	Yes	No
Vax/VMS	BSD	BSD	Yes	No
Windows (95/98/NT4/W2K/XP)	First	BSD	Yes	No

Figure 5.1: **Selected Observed Active Mapping Profiles.** Active Mapping profiles observed, by operating system of the host. Tests reported correspond to those described in section 3.3. Operating system data were not available for all mapping hosts, so the above table is not complete with respect to our test set; in some cases, version numbers were not known. Some entries with identical results across many versions of an OS have been summarized in one line; some very similar OS versions with identical results have been omitted for brevity. A value of "No Result" is due mostly to the use of TCP Wrappers; in some cases the mapped host did not support the service required to perform mapping. Since every machine accepted a RST in sequence, results for that test are not given.

5.2 Mapping Time

The amount of time taken to map an increasing number of hosts in given in Figure 5.2. Our implementation of Active Mapping is an untuned research prototype; as such the numbers below should serve only as a rough estimate to demonstrate the practicality of our approach.

The times measured are dependent on the policies found: since many tests' results are determined by the presence or absence of a response from the host within a certain time, some policies generate more timeouts than others. Most timeouts are on the order of 5–10 seconds; we found this interval to be sufficient to account for delays at the hosts and in the network.

Mapping a single host requires approximately 37 seconds. This minimum is due to the fact each of the mapping tests is repeated three times, and a single test requires two rounds of communication.

Wall-clock time rises sublinearly through 64 hosts, though for more than 64 hosts, times are likely to scale linearly since the mapper implements rate-limiting to avoid packet-buffer overflows (a problem we were able to alleviate in part by using larger-than-normal packet capture buffers). Mapping 101 hosts took 532 seconds, or 5.3 seconds per host; for 64 hosts, the time was 5.7 seconds per host and for 16 hosts, it took 10.1 seconds per host.

Our prototype implementation's inefficiency resulted in user time increases at the rate of somewhat less than two seconds per host being mapped. As a result, parallelism was limited, allowing steady-state rates of about 5 seconds per active host on the full-site mapping with thousands of hosts. We expect that this figure could be improved considerably with a better implementation. The packet-handling speeds of a decent NIDS are at least an order of magnitude better.

Hosts	User (s)	Sys (s)	Wall (s)	CPU (%)
1	2.3±0.0	0.1±0.0	37.1±0.5	6.7±0.2
2	4.3±0.0	0.3±0.0	41.1±0.6	11.0±0.2
4	4.4±0.0	0.3±0.0	40.0±1.2	11.6±0.3
8	11.9±0.1	0.7±0.1	85.3±0.6	14.8±0.2
16	28.7±0.2	1.7±0.2	161.7±0.6	18.7±0.1
32	60.0±0.0	3.4±0.2	232.3±0.8	27.2±0.1
64	105.1±0.1	6.1±0.1	365.0±3.5	28.1±0.2
Steady-state			5s / host	

Figure 5.2: **Time to map different numbers of machines.** Each run of the mapper performs network topology discovery and service discovery, and runs each mapping three times.

5.3 Mapping Traffic

We measured bidirectional network traffic generated during mapping. During a scan of a subnet with 101 live hosts, we recorded statistics (taken over three trials) relating to the number of bytes and packets generated by scanning, both to and from the mapper. The results are in Figure 5.3. ICMP packets were

due to ICMP service discovery, PMTU and hop count determination, and some IP mappings. TCP packets were due to TCP service discovery, PMTU and hop count determination (if ICMP was not supported), and TCP mappings.

	Total	**Per host**
Total bytes	1.9MB $\pm$ 49KB	19KB
Total packets	32,893 $\pm$ 345	326
ICMP packets	21,763 $\pm$ 2	215
TCP packets	10,588 $\pm$ 7	105
Packets/sec.	3.3 $\pm$ 0.0	
Bytes/sec.	191 $\pm$ 5	

Figure 5.3: **Traffic generated by mapping 101 hosts on a single subnet.** Three trials were conducted.

5.4 NIDS Integration Tests

We modified the Bro NIDS by adding support for disambiguation based on Active Mapping profiles; we stress that the choice of NIDS was for convenience since our techniques would apply equally to any TCP/IP-analyzing NIDS. Our goals in testing were twofold: first, to ensure that using Active Mapping would indeed result in correct interpretation of network traffic; second, to check that using Active Mapping would not incur any significant runtime cost. Accordingly, we ran two set of tests: first, a synthetic test with ambiguous traffic; second, a comparison of the original and Active Mapping-modified NIDS on real-world traces. (We expect that results would be substantially the same with any other NIDS integration.)

5.4.1 Synthetic Tests

In order to test the correctness of the modified NIDS (its ability to disambiguate traffic correctly, we generated HTTP attack traffic to 8 hosts with evasion measures added using `fragroute` [So02] to modified traffic to 2 hosts. `Fragroute` automatically transformed the request stream to include overlapping and inconsistent IP fragments and TCP segments. The inconsistency favored one of two policies (in our parlance, a "first" policy and a "BSD" policy); the data not expected to be accepted were chosen randomly. For the two machines receiving modified traffic, we used Active Mapping profiles which would allow the traffic to be properly interpreted.

We found that the unmodified NIDS believed the HTTP request to be:

```
GET /msadcTpo6EGKEY./../..%bTMmzy
QaL/system32/fipGNdDg++dir+c:\
```

rather than:

```
GET /msadc/../../../../../../winnt
/system32/cmd.exe?/c+dir+c:\
```

which was the actual request URL. It is clear that the unmodified NIDS, which had no way to properly resolve the ambiguous overlaps, chose the wrong data to use in reassembly. The modified NIDS performed reassembly correctly.

To measure the impact of Active Mapping on the NIDS' performance in the presence of a relatively high proportion of ambiguous traffic, we used two traces of 500 connections to the 8 hosts. In the first, where none of the connections were modified by `fragroute`, times were essentially identical over three trials. In the second, where connections to two of the machines were modified by `fragroute`, the Active Mapping-enabled NIDS was actually about 15% faster, since it was able to discard more data. In practice we expect this effect to be small, since it is only relevant when there are overlapping IP fragments or TCP segments (or the like); such occurrences are uncommon.

5.4.2 Real-world Tests

To get a picture of performance impact on a larger, more realistic dataset, we used two real-world traces. The first was of a wide variety of non-HTTP traffic (mostly just SYN/FIN/RST packets, the data filtered out) gathered by a one-hour capture at a busy site (100.2 MB data, 1.2 M packets, 273 K connections). The second was of two hours of HTTP traffic (with full data) at another site (137 MB, 197 K packets, 6,379 connections). In both cases, the results were the same: with Active Mapping on, execution time was essentially identical (with AM, it was less than 1% faster). Memory usage was approximately 200K higher with AM (specific profiles were used for about 4800 hosts; a default one for the rest), a small fraction of the 68MB used overall.

We are currently working on deploying an Active Mapping-enabled NIDS operationally to get more data on the impact of using AM profiles on performance and precision.

5.5 Conclusions and Recommendations

The test results suggest that mapping can be performed quite frequently. A full class C subnet can be scanned in about 20 minutes, so daily scans during off-peak times are certainly feasible. Importantly, with a steady-state rate of about 5 seconds per host (using our unoptimized prototype), it is feasible to completely remap even large sites—say, thousands of hosts—on a weekly basis during off-peak hours. Certain tests whose results we expect not to change often (e.g., those related to network topology) can be performed less frequently. The mapping-induced traffic of about 19 KB per host mapped is quite low and its impact during off-peak hours is likely to be negligible.

Remapping can also be triggered by any inconsistency between the stored policy and an observed one. For example, if a host sends an ICMP Needs Fragmentation message for a packet smaller than the stored PMTU, then the host should be remapped. External information, e.g., OS fingerprint results, can be used to detect changes in the status of a machine as well.

On-the-fly mapping—mapping when the first packet to a host is seen—is probably not possible, because many tests take several seconds. In any case, host policy changes are most likely to be triggered by infrequent operating system upgrades. More frequent changes to the policy database are those initiated by DHCP. As we have noted, we can store policies by MAC address and simply update a table when the NIDS sees a DHCP request (or is informed of a new lease by the DHCP server itself). For new hosts—say, a laptop attached for the first time to the network—mapping can be performed in under one minute (mapping a single host takes on the order of 30 seconds). This period of uncertainty is unlikely to be problematic,

since it is rare that DHCP clients export public services.

Runtime performance in the NIDS was not negatively affected by the addition of Active Mapping-based disambiguation. In fact, since using Active Mapping results allows the NIDS to discard additional packets, performance in some cases was actually improved. The additional memory footprint was approximately 100 bytes per host. We expect with all mappings implemented it would be on the order of a few hundred bytes.

The modified NIDS was also capable of correctly interpreting traffic in a way that the original one was not, detecting precise attacks that the original could only hint at through warnings about inconsistent retransmission. We stress that no amount of care in the design of the original could have changed its behavior in this respect: since hosts' behavior varies, any single policy employed by the NIDS will inevitably fail for hosts that employ a different one.

Chapter 6

Related Work

6.1 Normalization

As previously discussed, *traffic normalization* seeks to eliminate network traffic ambiguities by altering the traffic stream [HKP01]. The normalizer lies in the forwarding path of packets into a site. It reassembles IP fragments and TCP streams and statefully modifies or filters out nonconforming traffic before sending packets on to the internal network. Its efficacy in improving the precision of the NIDS relies on its output being interpreted in the same way by all the hosts on the network. It largely succeeds at achieving this goal; the paper also discusses some exceptions.

There are disadvantages to normalization, however. A normalizer performs the same sorts of tasks as a firewall, but is doing more work: it deals with TCP streams rather than just individual packets. Two main concerns arising from this complexity are performance and robustness. Since the normalizer is in the forwarding path, it must be able to process every packet as it arrives, even in the presence of stateholding attacks on itself. Further, it must be extremely reliable; if it is not, the entire site may lose Internet connectivity. An additional concern is that the normalizer changes the semantics of the streams it rewrites. This can block useful traffic, cause unintended problems with newer protocols, or decrease efficiency.

It appears that Active Mapping can replace many of the normalizations (see Section 3.4). Still, there are cases in which some amount of normalization can confer significant benefits: for example, its ability to remove flags and options can be used to eliminate any uncertainty as to their use.

Accordingly, it may sometimes work best to use *informed partial normalization*, that is, to perform a limited set of normalizations that eliminate ambiguities that Active Mapping cannot. If the host profiles indicate that certain kinds of noncompliant packets are never accepted by any host, or if administrators want an additional layer of safety, such packets may be filtered out at the firewall.

In this fashion, we can use Active Mapping to eliminate certain expensive normalizations (e.g., stream reassembly) that can be adequately mapped, while using traffic normalization for those that are relatively stateless and thus are less likely to cause performance or robustness problems. If the set of remaining normalizations is small enough, it may be possible to implement them using dedicated firewall/router hardware.

6.2 Mapping Tools

Active Mapping's tactic of sending specially crafted packets and interpreting responses to infer host properties has been employed in a variety of tools.

The most common purpose for such tools is to determine the operating system of a host. *Nmap* [Fyo01] uses port scans combined with IP and TCP options in responses to guess a host's operating system. *Queso* [Sa98] takes a similar tack, sending TCP packets with illegal flag combinations. By matching initial TTL values, advertised TCP windows, initial sequence numbers, nonconforming responses to packets sent to closed ports, and so forth, these tools can detect a large number of operating system versions.

Neither provides us with enough precise information on the long list of policy choices and parameters we need. Since doing OS detection takes approximately as long as Active Mapping, there seems little advantage to doing OS detection instead for this purpose; however, knowing the host OS can be very useful in eliminating false positives (i.e., could a particular attack actually succeed?). We note that, especially in light of the fact that operating systems may be user-modified, the *actually observed behavior* is the only relevant thing for correct interpretation: the OS version is at best a proxy for this information.

Nonetheless, there is a certain synergy between the two. If OS data are known, they can serve as a quick proxy for mapping characteristics when coupled to database containing canonical mappings by OS type and version. Conversely, known mappings can give at least a rough estimation of the OS a host is running. This can be useful for alert filtering: if a particular attack only works on Linux and the mapping data suggest a Windows machine, then we can filter out irrelevant alerts without knowing more precisely the OS versions.

The *Ntop* NIDS has been supplemented with network information inferred from passive monitoring [DS00]; this information appears to be limited to guessing the hop count and figuring out which IP addresses correspond to routers.

Tbit [PF01] tries to learn the TCP behavior of HTTP servers in regards to congestion control. It only sends legitimate TCP packets, relying on TCP options, advertised windows, and timing information to deduce the server's TCP configuration (or bugs therein). We use its scheme for implementing our user-level TCP.

Chapter 7

Summary and Future Work

7.1 Summary

Ambiguity in the interpretation of network traffic is a critical difficulty for Network Intrusion Detection. This ambiguity takes many forms. Some types may be resolved by careful construction of the NIDS. Other types are fundamentally more difficult to resolve, and require additional information about the network and the hosts being monitored. In this paper, we have presented *Active Mapping*, a method of eliminating network- and transport-layer ambiguities by informing the NIDS of relevant network and host TCP/IP stack policies. We stress that the ambiguities that Active Mapping seeks to address are readily exploitable; systems have been designed for doing just that [So02, Mc98].

Active Mapping runs separately from the NIDS (typically during off-peak hours) and works by sending specially crafted packets to each host and inferring policy from the responses it receives (or lack thereof). It does not require any real-time manipulation of the incoming traffic stream by the NIDS. In our tests with a NIDS modified to use Active Mapping-generated profiles, we found that there was essentially no cost in terms of speed or memory use to get the increased precision in analysis; we expect this will hold true for any NIDS. In addition, we have shown that Active Mapping itself is efficient in terms of time, network bandwidth consumed, and output size. Preliminary mapping results show considerable variation in policy among hosts' TCP/IP stacks, underscoring the need for the precise simulation that Active Mapping enables.

7.2 Future Work

We plan to deploy Active Mapping and an AM-enabled Bro NIDS at LBNL operationally. This will allow us to start to answer significant practical questions: How often is it necessary to remap hosts? How long does it take to do so? Is is feasible to do mapping by MAC address (and thus handle DHCP clients)?

We also plan to explore the integration of Active Mapping with related efforts. OS detection can be used as a quick proxy for mapping; likewise a host profile can be used to guess the host's OS. It is not clear how effective these tactics would be. We have also yet to try splitting disambiguation between a normalizer and

Active Mapping, i.e., informed partial normalization.

Lastly, a straightforward improvement would be to implement the full set of mappings in Section 3.4 to find if there are any unforseen obstacles (and make the implementation complete!). Performance improvements in the mapper are likely to become more important with more mappings implemented; a more efficient implementaion, perhaps in C, may be required.

7.3 A Concluding Note

The problem of ambiguous traffic is not confined to the network and transport layers. It also occurs at the application layer—for example, exactly how will a particular URL be interpreted?—and dealing with all possible ambiguities appears essentially intractable. Active Mapping profiles might be able to help lower false positives by allowing the NIDS to consider only platform-relevant attacks, but analysis of this potential benefit is beyond the scope of this paper. Thus we do not claim to have "solved" the NIDS evasion problem. However, we believe that the general problem of ambiguity resolution is best addressed in a systematic, layered fashion, and Active Mapping represents a step toward eliminating ambiguity at the bottom layers.

Bibliography

[SP02] Umesh Shankar and Vern Paxson, "Active Mapping: Resisting NIDS Evasion Without Altering Traffic". Submitted for publication.

[Be02] Steven M. Bellovin, "A Technique for Counting NATted Hosts." *Proceedings of the Second Internet Measurement Workshop*, November 2002.

[B+99] S. Blake et al, "An Architecture for Differentiated Services," RFC 2475, Dec. 1998.

[Dr97] R. Droms et al., "Dynamic Host Configuration Protocol," RFC 2131, Mar. 1997.

[DS00] L. Deri and S. Suin, "Improving Network Security Using Ntop," *Proc. Third International Workshop on the Recent Advances in Intrusion Detection (RAID 2000)*, 2000.

[EF94] K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," RFC 1631, 1994.

[Fyo01] Fyodor. *nmap*, 2001. http://www.insecure.org/nmap/ .

[HKP01] Mark Handley, Christian Kreibich and Vern Paxson, "Network Intrusion Detection: Evasion, Traffic Normalization," *Proc. 10th USENIX Security Symposium*, 2001.

[HF99] Steven A. Hofmeyr and Stephanie Forrest. "Immunizing Computer Networks: Getting All the Machines in Your Network to Fight the Hacker Disease" *Proc. 1999 IEEE Symposium on Security and Privacy*.

[Mc98] John McDonald. "Defeating Sniffers and Intrusion Detection Systems," Phrack Magazine, 8(54), Dec 25th, 1998.

[MLJ94] S. McCanne, C. Leres and V. Jacobson, `libpcap`, available at http://www.tcpdump.org, 1994.

[Pa98] Vern Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, 31(23-24), pp. 2435–2463, 14 Dec. 1999.

[PF01] Jitendra Padhye and Sally Floyd, "Identifying the TCP Behavior of Web Servers," *Proc. ACM SIGCOMM*, Aug. 2001.

[PN98] T. H. Ptacek and T. N. Newsham, "Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection", Secure Networks, Inc., Jan. 1998. http://www.icir.org/vern/Ptacek-Newsham-Evasion-98.ps

[Po80] J. Postel, "User Datagram Protocol," RFC 768, August 1980.

[Po81a] J. Postel, "Internet Protocol," RFC 791, September 1981.

[Po81b] J. Postel, “Internet Control Message Protocol,” RFC 792, September 1981.

[Po81c] J. Postel, “Transmission Control Protocol,” RFC 793, September 1981.

[Sa98] Savage, “QueSO,” *savage@apostols.org*, 1998. http://www.backupcentral.com/cgi-bin/redirect?url=ftp://contrib.redhat.com/pub/contrib/libc6/SRPMS/queso-980922b-1.src.rpm.

[So02] Dug Song, *fragroute*. Available at http://www.monkey.org/~dugsong/fragroute/.

[VKB01] Giovanni Vigna, Richard A. Kemmerer, and Per Blix. “Designing a Web of Highly-Configurable Intrusion Detection Sensors,” *Recent Advances in Intrusion Detection*, Proceedings of the 4th International Symposium, Lecture Notes in Computer Science vol. 2212, 2001.

[WS95] Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated: The Implementation*, Addison-Wesley, 1995.

[Yl02] T. Ylonen et al, “SSH Transport Layer Protocol,” Internet Draft, work in progress, 2002.