USENIX Association

Proceedings of the 10th USENIX Security Symposium

Washington, D.C., USA
August 13–17, 2001

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics

Mark Handley and Vern Paxson
AT&T Center for Internet Research at ICSI (ACIRI)
International Computer Science Institute
Berkeley, CA 94704 USA
{mjh,vern}@aciri.org

Christian Kreibich
Institut für Informatik
Technische Universität München
80290 München, Germany
kreibich@cs.tum.edu

Abstract

A fundamental problem for network intrusion detection systems is the ability of a skilled attacker to *evade* detection by exploiting ambiguities in the traffic stream as seen by the monitor. We discuss the viability of addressing this problem by introducing a new network forwarding element called a traffic *normalizer.* The normalizer sits directly in the path of traffic into a site and patches up the packet stream to eliminate potential ambiguities before the traffic is seen by the monitor, removing evasion opportunities. We examine a number of tradeoffs in designing a normalizer, emphasizing the important question of the degree to which normalizations undermine end-to-end protocol semantics. We discuss the key practical issues of "cold start" and attacks on the normalizer, and develop a methodology for systematically examining the ambiguities present in a protocol based on walking the protocol's header. We then present *norm,* a publicly available user-level implementation of a normalizer that can normalize a TCP traffic stream at 100,000 pkts/sec in memory-to-memory copies, suggesting that a kernel implementation using PC hardware could keep pace with a bidirectional 100 Mbps link with sufficient headroom to weather a high-speed flooding attack of small packets.

1 Introduction

A fundamental problem for network intrusion detection systems (NIDSs) that passively monitor a network link is the ability of a skilled attacker to *evade* detection by exploiting ambiguities in the traffic stream as seen by the NIDS [14]. Exploitable ambiguities can arise in three different ways:

(i) The NIDS may lack complete analysis for the full range of behavior allowed by a particular protocol. For example, an attacker can evade a NIDS that fails to reassemble IP fragments by intentionally transmitting their attack traffic in fragments rather than complete IP datagrams. Since IP end-systems are required to perform fragment reassembly, the attack traffic will still have the intended effect at the victim, but the NIDS will miss the attack because it never reconstructs the complete datagrams.

Of the four commercial systems tested by Ptacek and Newsham in 1998, none correctly reassembled fragments [14].

Also note that an attacker can evade the NIDS even if the NIDS does perform analysis for the protocol (e.g., it does reassemble fragments) if the NIDS's analysis is *incomplete* (e.g., it does not correctly reassemble out-of-order fragments).

(ii) Without detailed knowledge of the victim end-system's protocol implementation, the NIDS may be *unable* to determine how the victim will treat a given sequence of packets if different implementations interpret the same stream of packets in different ways. Unfortunately, Internet protocol specifications do not always accurately specify the complete behavior of protocols, especially for rare or exceptional conditions. In addition, different operating systems and applications implement different subsets of the protocols.

For example, when an end-system receives *overlapping* IP fragments that differ in the purported data for the overlapping region, some end-system's may favor the data first received, others the portion of the overlapping fragment present in the lower fragment, others the portion in the upper fragment.

(iii) Without detailed knowledge of the network topology between the NIDS and the victim end-system, the NIDS may be unable to determine whether a given packet will even be seen by the end-system. For example, a packet seen by the NIDS that has a low Time To Live (TTL) field may or may not have sufficient hop count remaining to make it all the way to the end-system [12]; see below for an example.

If the NIDS believes a packet was received when in fact it did not reach the end-system, then its model of the end-system's protocol state will be incorrect. If the attacker can find ways to systematically ensure that some packets will be received and some not, the attacker may be able to evade the NIDS.

The first of these shortcomings can in principle be addressed by a sufficiently diligent NIDS implementation, making sure that its analysis of each protocol is complete. However, the other two shortcomings are more fundamental: in the absence of external knowledge (end-system implementation details, topology details), *no amount* of analyzer completeness within the NIDS can help it correctly determine the end-system's ultimate processing of the packet stream. On the other hand, the attacker may be able to determine these end-system characteristics for a particular victim by actively probing the victim, perhaps in quite subtle (very hard to detect) ways. Thus, an attacker can craft their traffic so that, whatever algorithms the NIDS analyzer uses, it will err in determining how the end-system behaves.

Figure 1 shows an example of an evasion attack that can exploit either of the last two shortcomings above. The attacker fakes a missing packet, then sends a sequence of TCP packets above the sequence hole that contains the attack, and also sends a sequence of TCP packets containing innocuous data for the same TCP sequence space.

For the moment, ignore the "timed out" packets and assume all of the packets on the left arrive at the victim. Even in this case, the NIDS needs to know precisely how the end-system will interpret the inconsistent "retransmissions"—whether it will use "`n`" or "`r`" for sequence #1, "`o`" or "`i`" for sequence #2, etc.—when constructing the byte stream presented to the application. Unfortunately, different TCP stacks do different things in this error case; some accept the first packet, and some the second. There is no simple-and-correct rule the NIDS can use for its analysis.

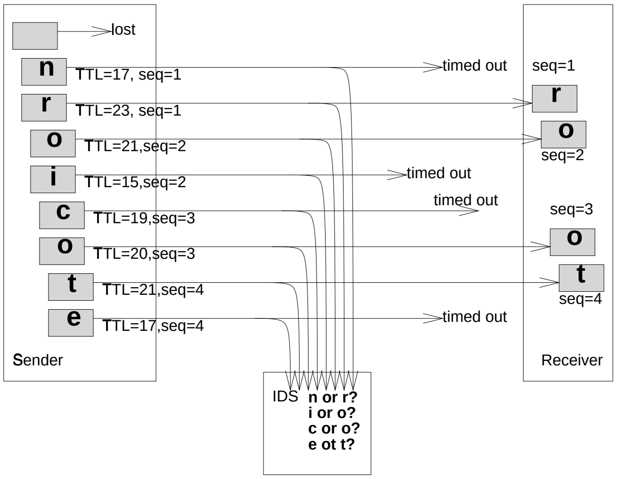


Figure 1: Inconsistent TCP "retransmissions"

In addition, the attacker may also be able to control which of the packets seen by the NIDS actually arrive at the end-system and which do not. In Figure 1, the attacker does so by manipulating the TTL field so that some of the packets lack sufficient hop count to travel all the way to the victim. In this case, to disambiguate the traffic the NIDS must know exactly how many forwarding hops lie between it and the victim.

One might argue that such evasive traffic or active probing will itself appear anomalous to the NIDS, and therefore the NIDS can detect that an attacker is attempting to evade it. However, doing so is greatly complicated by two factors. First, detection of an attempt at evasion degrades the precision of a NIDS's detection down from identifying the specifics of an attack to only being able to flag that an attack might possibly be in progress. Second, network traffic unfortunately often includes a non-negligible proportion of highly unusual, but benign, traffic, that will often result in *false positives* concerning possible evasion attempts. This is discussed in [12] as the problem of "crud"; examples include inconsistent TCP retransmissions and overlapping inconsistent fragments.

In the above argument we assume the attacker is aware of the existence of the NIDS, has access to its source code (or can deduce the operation of its algorithms) and attack profile database, and that the attacker is actively trying to evade the NIDS. All of these are prudent or plausible assumptions; for example, already the cracker community has discussed the issues [5] and some evasion toolkits (developed by “white hats” to aid in testing and hardening NIDSs) have been developed [2]. Thus, we again emphasize the serious and difficult nature of this problem: unless steps are taken to address all three of the evasion issues discussed above, network intrusion detection based on passive monitoring of traffic will eventually be completely circumventable, and provide no real protection to sites relying on it.

In this paper we consider the viability of addressing the evasion-by-ambiguity problem by introducing a new network forwarding element called a traffic *normalizer*. The normalizer’s job is to sit directly in the path of traffic into a site (a “bump in the wire”) and patch up or *normalize* the packet stream to remove potential ambiguities. The result is that a NIDS monitoring the normalized traffic stream no longer needs to consider potential ambiguities in interpreting the stream: the traffic as seen by the NIDS is guaranteed unambiguous, thanks to the normalizer. For example, a normalizer processing the traffic shown in Figure 1 might replace the data in any subsequent inconsistent retransmissions with the data from the original version of the same sequence space, so the only text the NIDS (*and the end-system*) would see would be noct.

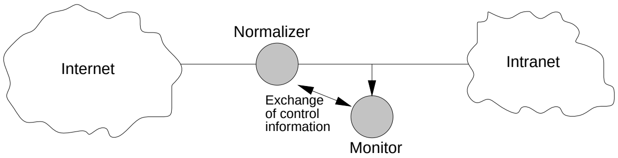


Figure 2: Typical locations of normalizer and NIDS

A normalizer differs from a firewall in that its purpose is not to prevent access to services on internal hosts, but to ensure that access to those hosts takes place in a manner that is unambiguous to the site’s NIDS. Figure 2 shows the typical locations of the normalizer relative to the NIDS and the end-systems being monitored. We will refer to traffic traveling from the “Internet” to the “Intranet” as *inbound*, and to traffic in the other direction as *outbound*.

The basic idea of traffic normalization was simultaneously invented in the form of a *protocol scrubber* [8, 13, 17]. The discussion of the TCP/IP scrubber in [8] focuses on ambiguous TCP retransmission attacks like the one described above. The key distinctions between our work and TCP/IP scrubbers is that we attempt to develop a systematic approach to identifying all potential normalizations (we find more than 70, per Appendix A), and we emphasize the implications of various normalizations with regard to maintaining or eroding the end-to-end transport semantics defined by the TCP/IP protocol suite. In addition, we attempt to defend against attacks on the normalizer itself, both through state exhaustion, and through state loss if the attacker can cause the normalizer or NIDS to restart (the “cold start” problem, per § 4.1).

In the next section we discuss other possible approaches for addressing the NIDS ambiguity problem. In § 3 we look at a number of tradeoffs in the design of a normalizer, and in § 4 two important practical considerations. § 5 first presents a systematic approach to discovering possible ambiguities in a protocol as seen by a network analyzer and then applies this approach to analyzing IP version 4. In § 6 we present examples of particularly illuminating normalizations for TCP, including an ambiguity problem that normalization cannot solve. We then discuss in § 7 a user-level normalizer called *norm*, which our performance measurements indicate should be able to process about 100,000 pkts/sec if implemented in the kernel.

2 Other approaches

In this section we briefly review other possible ways of addressing the problem of NIDS evasion, to provide general context for the normalizer approach.

Use a host-based IDS. We can eliminate ambiguities in the traffic stream by running the intrusion detection system (IDS) on all of the end-system hosts rather than by (or in addition to) passively monitoring network links. As the host IDS has access to the protocol state *above* the IP and transport stacks, it has unambiguous information as to how the host processes the packet stream. However, this approach is tantamount to giving up on network intrusion detection, as it loses the great advantage of being able to provide monitoring for an entire site cheaply, by deploying only a few monitors to watch key network links. Host-based systems also potentially face major deployment and management difficulties. In this work, we are concerned with the question of whether purely network-based IDS’s can remain viable, so we do not consider this solution further.

Understand the details of the intranet. In principle, a NIDS can eliminate much of the ambiguity if it has access to a sufficiently rich database cataloging the particulars of all of the end-system protocol implementations and the network topology. A major challenge with this approach is whether we can indeed construct such a database, particularly for a large site. Perhaps adding an active probing element to a NIDS can do so, and there has been some initial work in this regard [9]. However, another difficulty is that the NIDS would need to know how to make use of the database—it would require a model of every variant of every OS and application running within the site, potentially an immense task.

Bifurcating analysis. Finally, in some cases the NIDS can employ *bifurcating analysis* [12]: if the NIDS does not know which of two possible interpretations the end-system may apply to incoming packets, then it splits its analysis context for that connection into multiple threads, one for each possible interpretation, and analyzes each context separately from then onwards.

Bifurcating analysis works well when there are only a small number of possible interpretations no matter how many packets are sent. An example would be in the interpretation of the BACKSPACE vs. DELETE character during the authentication dialog at the beginning of a Telnet connection (before the user has an opportunity to remap the meaning of the characters): generally, either one or the other will delete the character of text most recently typed by the user. The NIDS can form two contexts, one interpreting DELETE as the deletion character, and the other interpreting BACKSPACE as the deletion character. Since the end-system will be in one state or the other, one of the analysis contexts will be correct at the NIDS no matter how many packets are sent.

However, bifurcating analysis will not be suitable if each arriving ambiguous packet requires an additional bifurcation, as in this case an attacker (or an inadvertent spate of "crud") can send a stream of packets such that the number of analysis contexts explodes exponentially, rapidly overwhelming the resources of the NIDS. Consider, for example, the attack shown in Figure 1. If the NIDS bifurcates its analysis on receipt of each potentially ambiguous packet, it will rapidly require a great deal of state and many analysis threads. Once it has seen the eight packets shown, it will need threads for the possible text root, nice, rice, noot, niot, roce, roct, *etc.* . . .

3 Normalization Tradeoffs

When designing a traffic normalizer, we are faced with a set of tradeoffs, which can be arranged along several axes:

- extent of normalization vs. protection
- impact on end-to-end semantics (service models)
- impact on end-to-end performance
- amount of state held
- work offloaded from the NIDS

Generally speaking, as we increase the degree of normalization and protection, we need to hold more state; performance decreases both for the normalizer and for end-to-end flows; and we impact end-to-end semantics more. Our goal is not to determine a single "sweet spot," but to understand the character of the tradeoffs, and, ideally, design a system that a site can tune to match their local requirements.

Normalization vs. protection. As a normalizer is a "bump in the wire," the same box performing normalization can also perform firewall functionality. For example, a normalizer can prevent known attacks, or shut down access to internal machines from an external host when the NIDS detects a probe or an attack. In this paper we concentrate mainly on normalization functionality, but will occasionally discuss protective functionality for which a normalizer is well suited.

End-to-end semantics. As much as possible, we would like a normalizer to preserve the end-to-end semantics of well-behaved network protocols, whilst cleaning up misbehaving traffic. Some packets arriving at the normalizer simply cannot be correct according to the protocol specification, and for these there often is a clear normalization to apply. For example, if two copies of an IP fragment arrive with the same fragment offset, but containing different data, then dropping either of the fragments or dropping the whole packet won't undermine the correct operation of the particular connection. Clearly the operation was already incorrect.

However, there are other packets that, while perfectly legal according to the protocol specifications, may still cause ambiguities for the NIDS. For example, it is perfectly legitimate for a packet to arrive at the normalizer with a low TTL. However, per the discussion in the Introduction, the NIDS cannot be sure whether the packet will reach the destination. A possible normalization for such packets is to increase its TTL to a large value.[1] For most traffic, this will have no adverse effect, but it will break diagnostics such as `traceroute`, which rely on the semantics of the TTL field for their correct operation.

Normalizations like these, which erode but do not brutally violate the end-to-end protocol semantics, present a basic tradeoff that each site must weigh as an individual policy decision, depending on its user community, performance needs, and threat model. In our analysis of different normalizations, we place particular emphasis on this tradeoff, because we believe the long-term utility of preserving end-to-end semantics is often underappreciated and at risk of being sacrificed for short-term

[1] Clearly, this is dangerous unless there is no possibility of the packet looping around to the normalizer again.

expediency.

Impact on end-to-end performance. Some normalizations are performed by modifying packets in a way that removes ambiguities, but also adversely affects the performance of the protocol being normalized. There is no clear answer as to how much impact on performance might be acceptable, as this clearly depends on the protocol, local network environment, and threat model.

Stateholding. A NIDS system must hold state in order to understand the context of incoming information. One form of attack on a NIDS is a *stateholding attack*, whereby the attacker creates traffic that will cause the NIDS to instantiate state (see § 4.2 below). If this state exceeds the NIDS's ability to cope, the attacker may well be able to launch an attack that passes undetected. This is possible in part because a NIDS generally operates passively, and so "fails open."

A normalizer also needs to hold state to correct ambiguities in the data flows. Such state might involve keeping track of unacknowledged TCP segments, or holding IP fragments for reassembly in the normalizer. However, unlike the NIDS, the normalizer is in the forwarding path, and so has the capability to "fail closed" in the presence of stateholding attacks. Similarly, the normalizer can perform "triage" amongst incoming flows: if the normalizer is near state exhaustion, it can shut down and discard state for flows that do not appear to be making progress, whilst passing and normalizing those that do make progress. The assumption here is that without complicity from internal hosts (see below), it is difficult for an attacker to fake a large number of *active* connections and stress the normalizer's stateholding.

But even given the ability to perform triage, if a normalizer operates fail-closed then we must take care to assess the degree to which an attacker can exploit stateholding to launch a denial-of-service attack against a site, by forcing the normalizer to terminate some of the site's legitimate connections.

Inbound vs. outbound traffic. The design of the Bro network intrusion detection system assumes that it is monitoring a bi-directional stream of traffic, and that either the source or the destination of the traffic can be trusted [12]. However it is equally effective at detecting inbound or outbound attacks.

The addition of a normalizer to the scenario potentially introduces an asymmetry due to its location—the normalizer protects the NIDS against ambiguities by processing the traffic before it reaches the NIDS (Figure 2). Thus, an internal host attempting to attack an external host might be able to exploit such ambiguities to evade the local NIDS. If the site's threat model includes such attacks, either two normalizers may be used, one on either side of the NIDS, or a NIDS integrated into a single normalizer. Finally, we note that if both internal and external hosts in a connection are compromised, there is little any NIDS or normalizer can do to prevent the use of encrypted or otherwise covert channels between the two hosts.

Whilst a normalizer will typically make most of its modifications to incoming packets, there may also be a number of normalizations it applies to outgoing packets. These normalizations are to ensure that the internal and external hosts' protocol state machines stay in step despite the normalization of the incoming traffic. It is also possible to normalize outgoing traffic to prevent unintended information about the internal hosts from escaping ([17], and see § 5.1 below).

Protection vs. offloading work. Although the primary purpose of a normalizer is to prevent ambiguous traffic from reaching the NIDS where it would either contribute to a state explosion or allow evasion, a normalizer can also serve to offload work from the NIDS. For example, if the normalizer discards packets with bad checksums, then the NIDS needn't spend cycles verifying checksums.

4 Real-world Considerations

Due to the adversarial nature of attacks, for security systems it is particularly important to consider not only the principles by which the system operates, but as much as possible also the "real world" details of operating the system. In this section, we discuss two such issues, the "cold start" problem, and attackers targeting the normalizer's operation.

4.1 Cold start

It is natural when designing a network traffic analyzer to structure its analysis in terms of tracking the progression of each connection from the negotiation to begin it, through the connection's establishment and data transfer operations, to its termination. Unless carefully done, however, such a design can prove vulnerable to incorrect analysis during a *cold start*. That is, when the analyzer first begins to run, it is confronted with traffic from already-established connections for which the analyzer lacks knowledge of the connection characteristics negotiated when the connections were established.

For example, the TCP scrubber [8] requires a connection to go through the normal start-up handshake. However, if a valid connection is in progress, and the scrubber restarts or otherwise loses state, then it will terminate any connections in progress at the time of the cold start, since to its analysis their traffic exchanges appear to violate the protocol semantics that require each newly seen connection to begin with a start-up handshake.

The cold-start problem also affects the NIDS itself. If the NIDS restarts, the loss of state can mean that previously monitored connections are no longer monitorable because the state negotiated at connection setup time is no longer available. As we will show, techniques required to allow clean normalizer restarts can also help a NIDS with cold start (§ 6.2).

Finally, we note that cold start is not an unlikely "corner case" to deal with, but instead an on-going issue for normalizers and NIDS alike. First, an attacker might be able to force a cold start by exploiting bugs in either system. Second, from operational experience we know that one cannot avoid occasionally restarting a monitor system, for example to reclaim leaked memory or update configuration files. Accordingly, a patient attacker who keeps a connection open for a long period of time can ensure a high probability that it will span a cold start.

4.2 Attacking the Normalizer

Inevitably we must expect the normalizer itself to be the target of attacks. Besides complete subversion, which can be prevented only though good design and coding practice, two other ways a normalizer can be attacked are stateholding attacks and CPU overload attacks.

Stateholding attacks. Some normalizations are stateless. For example, the TCP MSS option (Maximum Segment Size) is only allowed in TCP SYN packets. If a normalizer sees a TCP packet with an MSS Option but no SYN flag, then this is illegal; but even so, it may be unclear to the NIDS what the receiving host will do with the option, since its TCP implementation might incorrectly still honor it. Because the use of the option is illegal, the normalizer can safely remove it (and adjust the TCP checksum) without needing to instantiate any state for this purpose.

Other normalizations require the normalizer to hold state. For example, an attacker can create ambiguity by sending multiple copies of an IP fragment with different payloads. While a normalizer can remove fragment-based ambiguities by reassembling all fragmented IP packets itself before forwarding them (and if necessary re-fragmenting correctly), to do this, the normalizer must hold fragments until they can be reassembled into a complete packet. An attacker can thus cause the normalizer to use up memory by sending many fragments of packets without ever sending enough to complete a packet.

This particular attack is easily defended against by simply bounding the amount of memory that can be used for fragments, and culling the oldest fragments from the cache if the fragment cache fills up. Because fragments tend to arrive together, this simple strategy means an attacker has to flood with a very high rate of fragments to cause a problem. Also, as IP packets are unreliable, there's no guarantee they arrive anyway, so dropping the occasional packet doesn't break any end-to-end semantics.

More difficult to defend against is an attacker causing the normalizer to hold TCP state by flooding in, for example, the following ways:

1. Simple SYN flooding with SYNs for multiple connections to the same or to many hosts.
2. ACK flooding. A normalizer receiving a packet for which it has no state might be designed to then instantiate state (in order to address the "cold start" problem).
3. Initial window flooding. The attacker sends a SYN to a server that exists, receives a SYN-ACK, and then floods data without waiting for a response. A normalizer would normally temporarily store unacknowledged text to prevent inconsistent retransmissions.

Our strategy for defending against these is twofold. First, the normalizer knows whether or not it's under attack by monitoring the amount of memory it is consuming. If it's not under attack, it can instantiate whatever state it needs to normalize correctly. If it believes it's under attack, it takes a more conservative strategy that is designed to allow it to survive, although some legitimate traffic will see degraded performance.

In general our aim when under attack is to only instantiate TCP connection state when we see traffic from an internal (and hence trusted) host, as this restricts stateholding attacks on the normalizer to those actually involving real connections to internal hosts. Note here that the normalizer is explicitly *not* attempting to protect the internal hosts from denial-of-service attacks; only to protect itself and the NIDS.

CPU overload attacks. An attacker may also attempt to overload the CPU on the normalizer. However, unlike stateholding attacks, such an attack cannot cause the normalizer to allow an ambiguity to pass. Instead, CPU overload attacks can merely cause the normalizer to forward packets at a slower rate than it otherwise would.

In practice, we find that most normalizations are rather cheap to perform (§ 7.2), so such attacks need to concentrate on the normalizations where the attacker can utilize computational complexity to their advantage. Thus, CPU utilization attacks will normally need to be combined with stateholding attacks so that the normalizer performs an expensive search in a large state-space. Accordingly, we need to pay great attention to the implementation of such search algorithms, with extensive use of constant-complexity hash algorithms to locate matching state. An additional difficulty that arises is the need to be opportunistic about garbage collection, and to apply algorithms that are low cost at the possible expense of not being completely optimal in the choice of state that is reclaimed.

5 A Systematic Approach

For a normalizer to completely protect the NIDS, in principle we must be able to normalize every possible sequence of packets that the NIDS might treat differently from the end-system. Given that the NIDS cannot possibly know all the application state at the end-system for all applications, we focus in this work on the more tractable problem of normalizing the internetwork (IP, ICMP) and transport (TCP, UDP) layers.

Even with this somewhat more restricted scope, we find there are still a very large number of possible protocol ambiguities to address. Consequently, it behooves us to develop a systematic methodology for attempting to identify and analyze all of the possible normalizations. The methodology we adopt is to walk through the packet headers of each protocol we consider. This ensures that we have an opportunity to consider each facet of the semantics associated with the protocol.

For each header element, we consider its possible range of values, their semantics, and ways an attacker could exploit the different values; possible actions a normalizer might take to thwart the attacks; and the effects these actions might have on the protocol's end-to-end semantics. Whilst our primary intention is to explore the possible actions a normalizer can take, the exercise also raises interesting questions about the incompleteness of the specifications of error handling behavior in Internet protocols, and about the nature of the intentional and unintentional end-to-end semantics of Internet protocols.

For reasons of space, we confine our analysis here to a single protocol; we pick IP (version 4) because it is simple enough to cover fairly thoroughly in this paper, yet has rich enough semantics (especially fragmentation) to convey the flavor of more complicated normalizations. In § 6 we then present some particularly illuminating examples of TCP normalizations. We defer our methodical analysis of TCP (and UDP and ICMP) to [4].

Note that many of the normalizations we discuss below appear to address very unlikely evasion scenarios. However, we believe the right design approach is to normalize *everything* that we can see how to correctly normalize, because packet manipulation and semantic ambiguity is sufficiently subtle that we may miss an attack, but still thwart it because we normalized away the degrees of freedom to express the attack.

Figure 3 shows the fields of the IP packet header. For each field we identify possible issues that need normalization and discuss the effects of our solutions on end-to-end semantics. The reader preferring to delve into only more interesting normalizations may choose to jump ahead to § 5.1.

Version. A normalizer should only pass packets with IP version fields which the NIDS understands.

Header length. It may be possible to send a packet with an incorrect header length field that arrives at an end-systems and is accepted. However, other operating systems or internal routers may discard the packet. Thus the NIDS does not know if the packet will be processed or not.
Solution: If the header length field is less than 20 bytes, the header is incomplete, and the packet should be discarded. If the header length field exceeds the packet length, the packet should be discarded. (See *Total length* below for a discussion of exactly what constitutes the packet length.)
Effect on semantics: Packet is ill-formed—no adverse effect.
Note: If the header length is greater than 20 bytes, this indicates options are present. See IP option processing below.

Type Of Service/Diffserv/ECN. These bits have recently been reassigned to differentiated services [11] and explicit congestion notification [15].

Issue: The Diffserv bits might potentially be used to deterministically drop a subset of packets at an internal Diffserv-enabled router, for example by sending bursts of packets that violate the conditioning required by their Diffserv class.
Solution: If the site does not actually use Diffserv mechanisms for incoming traffic, clear the bits.
Effect on semantics: If Diffserv is not being used internally, the bits should be zero anyway, so zeroing them is safe. Otherwise, clearing them breaks use of Diffserv.

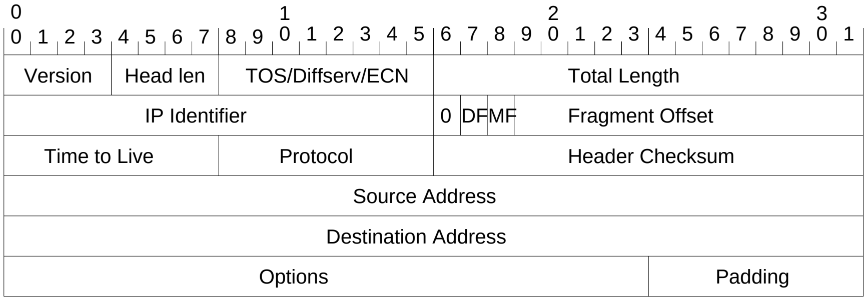


Figure 3: IP v4 Header

Issue: Some network elements (such as firewalls) may drop packets with the ECN bits set, because they are not yet cognizant of the new interpretation of these header bits.
Solution: Clear the bits for all connections unless the connection previously negotiated use of ECN. Optionally, remove attempts to negotiate use of ECN.
Effect on semantics: For connections that have not negotiated use of ECN, no end-to-end effect. Removing attempted negotiation of ECN will prevent connections from benefiting from avoiding packet drops in some circumstances.

Total length. If the total length field does not match the actual total length of the packet as indicated by the link layer, then some end-systems may treat the packet as being one length, some may treat it as being the other, and some may discard the packet.
Solution: Discard packets whose length field exceeds their link-layer length. Trim packets with longer link-layer lengths down to just those bytes indicated by the length field.
Effect on semantics: None, only ill-formed packets are dropped.

IP Identifier. See § 5.1.

Must Be Zero. The IP specification requires that the bit between "IP Identifier" and "DF" must be zero.
Issue: If the bit is set, then intermediary or end-systems processing the packet may or may not discard it.
Solution: Clear the bit to zero.
Effect on semantics: None, since the bit is already required to be zero.
Note: We might think that we could just as well discard the packet, since it violates the IP specification. The benefit of merely clearing the bit is that if in the future a new use for the bit is deployed, then clearing the bit will permit connections to continue, rather than abruptly terminating them, in keeping with the philosophy that Internet protocols should degrade gradually in the presence of difficulties.

Don't Fragment (DF) flag. If DF is set, and the Maximum Transmission Unit (MTU) anywhere in the internal network is smaller than the MTU on the access link to the site, then an attacker can deterministically cause some packets to fail to reach end-systems behind the smaller MTU link. This is done by setting DF on packets with a larger MTU than the link.
Note: The NIDS might be able to infer this attack from ICMP responses sent by the router that drops the packets, but the NIDS needs to hold state to do so, leading to state-holding attacks on the NIDS. Also, it is not certain that the NIDS will always see the ICMP responses, due to rate-limiting and multi-pathing.
Solution: Clear DF on incoming packets.
Effect on semantics: Breaks "Path MTU Discovery." If an incoming packet is too large for an internal link, it will now be fragmented, which could have an adverse effect on performance—in the router performing the fragmentation, in the end host performing reassembly, or due to increased effective packet loss rates on the network links after fragmentation occurs [6]. That said, for many network environments, these are unlikely to be serious problems.

Issue: Packets arriving with DF set and a non-zero fragmentation offset are illegal. However, it is not clear whether the end-system will discard these packets.
Solution: Discard such packets.
Effect on semantics: None, ill-formed packet.

More Fragments (MF) flag, **Fragment Offset.** We treat these two fields together because they are interpreted together. An ambiguity arises if the NIDS sees two fragments that overlap each other and differ in their contents. As noted in [14], different operating systems resolve the ambiguity differently.
Solution: Reassemble incoming fragments in the normalizer rather than forwarding them. If required, re-fragment the packet for transmission to the internal network if it is larger than the MTU.
Effect on semantics: Reassembly is a valid operation for a router to perform, although it is not normally done. Thus this does not affect end-to-end semantics.
Note: A normalizer that reassembles fragments is vulnerable to stateholding attacks, and requires an appropriate triage strategy to discard partially reassembled packets if the normalizer starts to run out of memory.

Issue: Packets where the length plus the fragmentation offset exceeds 65535 are illegal. They may or may not be accepted by the end host. They may also cause some hosts to crash.
Solution: Drop the packets.
Effect on semantics: Packet is ill-formed, so no effect.

TTL (Time-to-live). As with DF, an attacker can use TTL to manipulate which of the packets seen by the NIDS reaches the end-system, per the discussion for Figure 1.
Solution #1: In principle, a NIDS could measure the number of hops to every end host, and ignore packets that lack sufficient TTL. In practice, though, at many sites this requires holding a large amount of state, and it is possible that the internal routing may change (possibly triggered by the attacker in some way) leaving a window of time where the NIDS's measurement is incorrect.
Solution #2: The NIDS may also be able to see ICMP time-exceeded-in-transit packets elicited by the attack. However, ICMP responses are usually rate limited, so the NIDS may still not be able to tell exactly which packets were discarded.
Solution #3: Configure the normalizer with a TTL that is larger than the longest path across the internal site. If packets arrive that have a TTL lower than the configured minimum, then the normalizer restores the TTL to the minimum.
Effect on semantics: First, if a routing loop passes through the normalizer, then it may be possible for packets to loop forever, rapidly consuming the available bandwidth. Second, restoring TTL will break `traceroute` due to its use of limited-TTL packets to discover forwarding hops. Third, restoring TTL on multicast packets may impair the performance of applications that use expanding ring searches. The effect will be that all internal hosts in the group appear to be immediately inside the normalizer from the point of view of the search algorithm.

Protocol. The protocol field indicates the next-layer protocol, such as TCP or UDP. Blocking traffic based on it is a firewall function and not a normalizer function. However, an administrator may still configure a normalizer to discard packets that do not contain well-known protocols, such as those the NIDS understands.

IP header checksum. Packets with incorrect IP header checksums might possibly be accepted by end-hosts with dodgy IP implementations.
Solution: In practice this is not a likely scenario, but the normalizer can discard these packets anyway, which avoids the NIDS needing to verify checksums itself.
Effect on semantics: Normally, no effect. However, it might be possible to use corrupted packets to gather information on link errors or to signal to TCP not to back off because the loss is due to corruption and not congestion. But since routers will normally discard packets with incorrect IP checksums anyway, the issue is likely moot.

Source address. If the source address of an IP packet is invalid in some way, then the end-host may or may not discard the packet. Examples are `127.0.0.1` (localhost), `0.0.0.0` and `255.255.255.255` (broadcast), multicast (class D) and class E addresses.
Solution: Drop the packet.
Effect on semantics: None, packet is ill-formed.
Note: If the incoming packet has a source address belonging to a known internal network, the normalizer might be configured to drop the packet. This is more firewall-type functionality than normalization, but will generally be desirable. However it would break applications that rely on "source routing" packets via an external host and back into the site, such as using `traceroute` to trace a route from an external site back *into* the tracing site. Also, if an outgoing packet has a source address that does not belong to a known internal network, the normalizer might be configured to drop the packet.

Destination address. Like source addresses, invalid destination addresses might cause unexpected behavior at internal hosts. Examples are local broadcast addresses ("smurf" attacks), the localhost and broadcast addresses mentioned above, and class E addresses (which are currently unused).
Solution: Drop the packet. In addition, the normalizer should be capable of dropping incoming packets with destination addresses that would not normally be routed to the site; these might appear as a result of source-routing, and it is unclear what effect they might have on internal hosts or routers.
Effect on semantics: None, destination is illegal.

IP options. IP packets may contain IP options that modify the behavior of internal hosts, or cause packets to be interpreted differently.
Solution: Remove IP options from incoming packets.
Effect on semantics: For end-to-end connections, presumably none, as IP options should not have effects visible at higher layers; *except* the absence of an option may impair end-to-end connectivity, for example because the connectivity requires source routing. For diagnostics tools, potentially serious.
That said, the reality today is that options generally suffer from poor performance because routers defer their processing to the "slow path," and many sites disable their use to counter certain security risks. So in practice, removing IP options should have little ill effect, other than the loss of source routing for diagnosing connectivity problems. This last can be addressed (as can all semantic tradeoffs associated with normalization) through site-specific policies controlling the normalizer's operation.

Padding. The padding field at the end of a list of IP options is explicitly ignored by the receiver, so it is difficult to see that it can be manipulated in any useful way. While it does provide a possible covert channel, so do many other header fields, and thwarting these is not a normalizer task.
Solution: Zero the padding bytes, on the principle that we perform normalizations even when we do not know of a corresponding attack.
Effect on semantics: None, field is explicitly ignored.

5.1 The IP Identifier and Stealth Port Scans

The IP identifier (ID) of outgoing packets may give away information about services running on internal hosts. This issue is not strictly a normalizer problem, but the normalizer is in a location well suited to deal with the issue.

One particular problem is the exceedingly devious stealth port-scanning technique described in [16, 18], which enables an attacker to probe the services running on a remote host without giving away the IP address of the host being used to conduct the scan. Figure 4 illustrates the technique, which we review here to develop how a normalizer can thwart it. Host A is the attacker, V is the victim, and P is the *patsy*. The patsy must run an operating system that increments the IP ID by one[2] for every packet sent, no matter to what destination—many common operating systems use such a "global" IP ID.

Host A continually exchanges packets with host P, either through a TCP transfer or simply by pinging it. While doing this, the IP IDs of the responses from P to A normally increment by one from one packet to the next. Now A fakes a TCP SYN to the port on V they wish to probe, and they fake the source address of the packet as being from P.

If there is no service listening on the port, V sends a RST to P. As P has no associated connection state, P ignores the RST, and there is no effect on the IP IDs of the stream of packets from P to A.

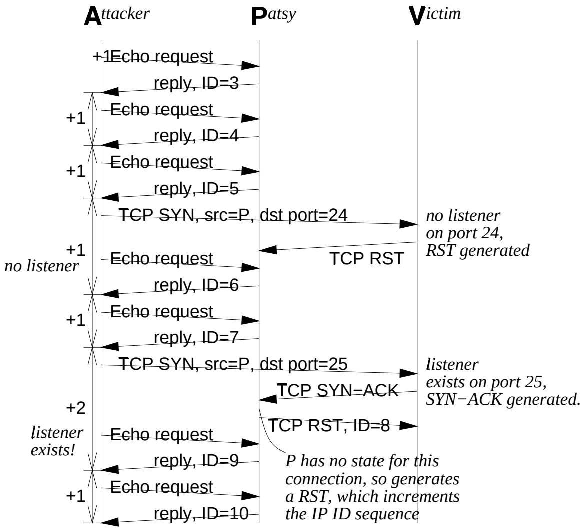


Figure 4: Stealth Port Scan

However, if there is a service listening on the port, V sends a SYN-ACK to P to complete the connection, rather than a RST. P has no state for this connection, and promptly sends a RST back to V. In so doing, the global ID sequence on P increases by one; consequently, in the stream of packets from P to A, the attacker observes a step of two (rather than one) in the ID sequence, since it missed one of the packets sent by P, namely the RST from P to V.

Thus P and not A appears to be the host conducting the port-scan, whereas in fact it is completely innocent. V never sees a packet with a source of A. If A chooses a different patsy for every port it wishes to check, then this port scan is very hard to detect.

The solution for patsies is for the normalizer to scramble (in a cryptographically secure, but reversible fashion) the IP IDs of incoming and outgoing packets. This prevents internal hosts from being used as patsies for such scans. The effect on semantics is that any diagnostic protocol that reports the IP IDs of incoming packets back to the sender may break. ICMP messages can still function if the normalizer applies the unscrambling to the embedded ID fields they carry.

[2] More generally, advances the ID field in a predictable fashion.

The solution for victims is to use the "reliable RST" technique (see § 6.1 below). The normalizer transmits a "keep-alive" acknowledgment (ACK) packet behind every RST packet it forwards out of the site. When the ACK arrives at the patsy, the patsy will reply with a RST, just as it does in the SYN-ACK case. Consequently, the IP ID sequence as seen by the attacker will jump by two in both cases, whether the victim is running the given service or not.

Sending keep-alives for reliable RSTs generates extra traffic, but has no effect on end-to-end semantics, since the keep-alive ACK following the RST is guaranteed to be either rejected by the victim (if it first received the RST) or ignored (if the RST was lost and the connection remains open).

6 Examples of TCP Normalizations

We applied the same "walk the header" methodology as in the previous section to TCP, UDP, and ICMP. However, due to space limitations we defer the detailed analysis to [4], and in this section focus on three examples for TCP that illuminate different normalization issues: reliable RSTs, cold start for TCP, and an example of a TCP ambiguity that a normalizer cannot remove.

6.1 Reliable RSTs

With TCP, the control signals for connection establishment and completion (SYN and FIN, respectively) are delivered reliably, but the "abrupt termination" (RST) signal is not. This leads to a significant problem: in general, both a normalizer and a NIDS needs to tear down state for an existing connection once that connection completes, in order to recover the associated memory. But it is not safe to do so upon seeing a RST, because the RST packet might be lost prior to arriving at the receiver, or might be rejected by the receiver.

Thus, a monitor cannot tell whether a given RST does in fact terminate its corresponding connection. If the monitor errs and assumes it does when in fact it did not, then an attacker can later continue sending traffic on the connection, and the monitor will lack the necessary state (namely, that the connection is still established, and with what sequence numbers, windows, etc.) to correctly interpret that traffic. On the other hand, if the monitor assumes the RST does *not* terminate the connection, then it is left holding the corresponding state potentially indefinitely. (Unfortunately, RST-termination is not uncommon in practice, so even for benign traffic, this state will grow significantly over time.)

The RST might fail to arrive at the receiver because of normal loss processes such as buffer overflows at congested routers, or because of manipulation by an attacker, such as the TTL games discussed in the context of Figure 1. In addition, the rules applied by receivers to determine whether a particular RST is valid vary across different operating systems, which the NIDS likely cannot track.

A general solution to this problem would be to ensure that RSTs are indeed delivered and accepted, i.e., we want "reliable RSTs." We can do so, as follows. Whenever the normalizer sees a RST packet sent from A to B, after normalizing it and sending it on, it synthesizes a second packet and sends that to B, too. This additional packet takes the form of a TCP "keep-alive," which is a dataless[3] ACK packet with a sequence number just below the point cumulatively acknowledged by B. The TCP specification requires that B must in turn reply to the keep-alive with an ACK packet of its own, one with the correct sequence number to be accepted by A, to ensure that the two TCP peers are synchronized. However, B only does this if the connection is still open; if it is closed, it sends a RST in response to the keep-alive.

Thus, using this approach, there are four possible outcomes whenever the normalizer forwards a RST packet (and the accompanying keep-alive):

(i) The RST was accepted by B, and so B will generate another RST back to A upon receipt of the keep-alive;

(ii) the RST either did not make it to B, or B ignored it, in which case B will generate an ACK in response to the keep-alive;

(iii) the keep-alive did not make it to B, or B ignored it (though this latter shouldn't happen);

(iv) or, the response B sent in reply to the keep-alive was lost before the normalizer could see it.

The normalizer then uses the following rule for managing its state upon seeing a RST: *upon seeing a RST from A to B, retain the connection state; but subsequently, upon seeing a RST from B to A, tear down the state.*[4] Thus, the normalizer only discards the connection state upon seeing proof that B has indeed terminated the connection. Note that if either A or B misbehaves, the scheme still works, because one of the RSTs will still

[3] In practice, one sends the last acknowledged byte if possible, for interoperability with older TCP implementations.

[4] Of course we do not send a keep-alive to make the second RST reliable or we'd initiate a RST war.

have been legitimate; only if A and B collude will the scheme fail, and, as noted earlier, in that case there is little a normalizer or a NIDS can do to thwart evasion.

The rule above addresses case *(i)*. For case *(ii)*, the normalizer needn't do anything special (it still retains the connection state, in accordance with the rule). For cases *(iii)* and *(iv)*, it will likewise retain the state, perhaps needlessly; but these cases should be rare, and are not subject to manipulation by A. They could be created by B if B is malicious; but not to much effect, as in that case the connection is already terminated as far as A is concerned.

6.2 Cold start for TCP

Recall that the "cold start" problem concerns how a normalizer should behave when it sees traffic for a connection that apparently existed before the normalizer began its current execution (§ 4.1). One particular goal is that the normalizer (and NIDS) *refrain from instantiating state* for apparently-active connections unless they can determine that the connection is indeed active; otherwise, a flood of bogus traffic for a variety of non-existent connections would result in the normalizer creating a great deal of state, resulting in a state-holding attack.

Accordingly, we need some way for the normalizer to distinguish between genuine, pre-existing connections, and bogus, non-existent connections, and to do so in a *stateless* fashion!

As with the need in the previous section to make RSTs trustworthy, we can again use the trick of encapsulating the uncertainty in a probe packet and using the state held (or not held) at the receiver to inform the normalizer's decision process.

Our approach is based on the assumption that the normalizer lies between a trusted network and an untrusted network, and works as follows. Upon seeing a packet from A to B for which the normalizer does not have knowledge of an associated connection, if A is from the trusted network, then the normalizer instantiates state for a corresponding connection and continues as usual. However, if A is from the untrusted network, the normalizer transforms the packet into a "keep-alive" by stripping off the payload and decrementing the sequence number in the header. It then forwards the modified packet to B and forgets about it. If there is indeed a connection between A and B, then B will respond to the keep-alive with an ACK, which will suffice to then instantiate state for the connection, since B is from the trusted network. If no connection does in fact exist, then B will either respond with a RST, or not at all (if B itself does not exist, for example). In both of these cases, the normalizer does not wind up instantiating any state.

The scheme works in part because TCP is reliable: removing the data from a packet does not break the connection, because A will work diligently to eventually deliver the data and continue the connection.

(Note that a similar scheme can also be applied when the normalizer sees an initial SYN for a new connection: by only instantiating state for the connection upon seeing a SYN-ACK from the trusted network, the load on a normalizer in the face of a SYN flooding attack is diminished to reflect the rate at which the target can absorb the flood, rather than the full incoming flooding rate.)

Even with this approach, though, cold start for TCP still includes some subtle, thorny issues. One in particular concerns the *window scaling* option that can be negotiated in TCP SYN packets when establishing a connection. It specifies a left-shift to be applied to the 16 bit window field in the TCP header, in order to permit receiver windows of greater than 64 KB. In general, a normalizer must be able to tell whether a packet will be accepted at the receiver. Because receivers can discard packets with data that lies beyond the bounds of the receiver window, the normalizer needs to know the window scaling factor in order to mirror this determination. However, upon cold start, the normalizer cannot determine the window scaling value, because the TCP endpoints no longer exchange it, they just use the value they agreed upon at connection establishment.

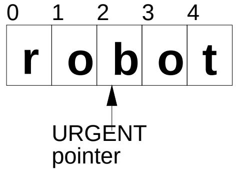
We know of no fully reliable way by which the normalizer might infer the window scaling factor in this case. Consequently, if the normalizer wishes to avoid this ambiguity, it must either ensure that window scaling is simply not used, i.e., *it must remove the window scale option from all TCP SYN packets* to prevent its negotiation (or it must have access to persistent state so it can recover the context for each active connection unambiguously).

Doing so is not without a potentially significant cost: window scaling is required for good performance for connections operating over long-haul, high-speed paths [1], and sites with such traffic might in particular want to disable this normalization.

More generally, this aspect of the cold start problem illustrates how normalizations can sometimes come quite expensively. The next section illustrates how they are sometimes simply not possible.

6.3 Incompleteness of Normalization

In the absence of detailed knowledge about the various applications, normalizations will tend to be restricted to the internetwork and transport layers. However, even at the transport level a normalizer cannot remove all possible ambiguities. For example, the semantics of the TCP urgent pointer cannot be understood without knowing the semantics of the application using TCP:



If the sender sends the text "`robot`" with the TCP urgent pointer set to point to the letter "`b`", then the application may receive *either* "`robot`" or "`root`," depending on the socket options enabled by the receiving application. Without knowledge of the socket options enabled, the normalizer cannot correctly normalize such a packet because either interpretation of it could be valid.

In this case, the problem is likely not significant in practice, because all protocols of which we are aware either enable or disable the relevant option for the entire connection—so the NIDS can use a bifurcating analysis without the attacker being able to create an exponential increase in analysis state. However, the example highlights that normalizers, while arguably very useful for reducing the evasion opportunities provided by ambiguities, are not an all-encompassing solution.

7 Implementation

We have implemented *norm*, a fairly complete, user-level normalizer for IP, TCP, UDP and ICMP. The code comprises about 4,800 lines of C and uses `libpcap` [10] to capture packets and a raw socket to forward them. We have currently tested *norm* under FreeBSD and Linux, and will release it (and NetDuDE, see below) publicly in Summer 2001 via *www.sourceforge.net*.

Naturally, for high performance a production normalizer would need to run in the kernel rather than at user level, but our current implementation makes testing, debugging and evaluation much simpler.

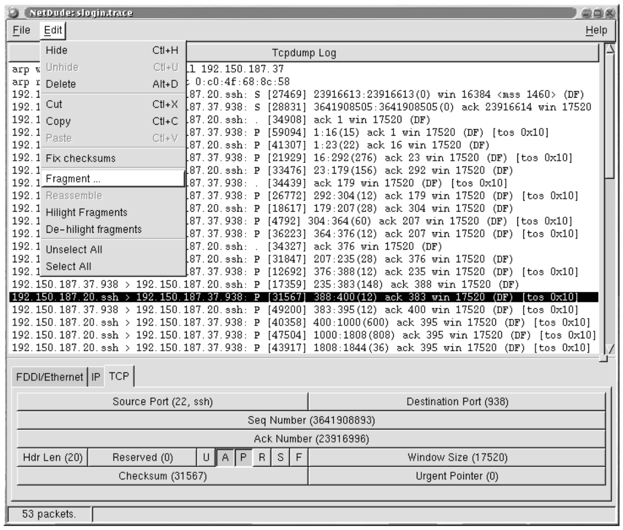
Appendix A summarizes the complete list of normalizations *norm* performs, and these are discussed in detail in [4]. Here we describe our process for testing and evaluating *norm*, and find that the performance on commodity PC hardware is adequate for deployment at a site like ours with a bidirectional 100Mb/s access link to the Internet.

7.1 Evaluation methodology

In evaluating a normalizer, we care about completeness, correctness, and performance. The evaluation presents a challenging problem because by definition most of the functionality of a normalizer applies only to unusual or "impossible" traffic, and the results of a normalizer in general are invisible to connection endpoints (depending on the degree to which the normalizations preserve end-to-end semantics). We primarily use a trace-driven approach, in which we present the normalizer with an input trace of packets to process as though it had received them from a network interface, and inspect an output trace of the transformed packets it in turn would have forwarded to the other interface.

Each individual normalization needs to be tested in isolation to ensure that it behaves as we intend. The first problem here is to obtain test traffic that exhibits the behavior we wish to normalize; once this is done, we need to ensure that *norm* correctly normalizes it.

With some anomalous behavior, we can capture packet traces of traffic that our NIDS identifies as being ambiguous. Primarily this is "crud" and not real attack traffic [12]. We can also use tools such as *nmap* [3] and *fragrouter* [2] to generate traffic similar to that an attacker might generate. However, for most of the normalizations we identified, no real trace traffic is available, and so we must generate our own.



```
arp w                           l1 192.150.187.37
arp r                           t 0:c0:4f:68:8c:58
192.1                           87.20.ssh: S [27469] 23916613:23916613(0) win 16384 <mss 1460> (DF)
192.1                           87.37.938: S [28831] 3641908505:3641908505(0) ack 23916614 win 17520
192.1                           87.20.ssh: . [34908] ack 1 win 17520 (DF)
192.1                           87.37.938: P [59094] 1:16(15) ack 1 win 17520 (DF) [tos 0x10]
192.1                           87.20.ssh: P [41307] 1:23(22) ack 16 win 17520 (DF)
192.1                           87.37.938: P [21929] 16:292(276) ack 23 win 17520 (DF) [tos 0x10]
192.1                           87.20.ssh: P [33476] 23:179(156) ack 292 win 17520 (DF)
192.1                           87.37.938: . [34439] ack 179 win 17520 (DF) [tos 0x10]
192.1                           87.37.938: P [26772] 292:304(12) ack 179 win 17520 (DF) [tos 0x10]
192.1                           87.20.ssh: P [18617] 179:207(28) ack 304 win 17520 (DF)
192.1                           87.37.938: P [4792] 304:364(60) ack 207 win 17520 (DF) [tos 0x10]
192.1                           87.37.938: P [36223] 364:376(12) ack 207 win 17520 (DF) [tos 0x10]
192.1                           87.20.ssh: . [34327] ack 376 win 17520 (DF)
192.1                           87.20.ssh: P [31847] 207:235(28) ack 376 win 17520 (DF)
192.1                           87.37.938: P [12692] 376:388(12) ack 235 win 17520 (DF) [tos 0x10]
192.150.187.37.938 > 192.150.187.20.ssh: P [17359] 235:383(148) ack 388 win 17520 (DF)
192.150.187.20.ssh > 192.150.187.37.938: P [31567] 388:400(12) ack 383 win 17520 (DF) [tos 0x10]
192.150.187.37.938 > 192.150.187.20.ssh: P [49200] 383:395(12) ack 400 win 17520 (DF) [tos 0x10]
192.150.187.20.ssh > 192.150.187.37.938: P [40358] 400:1000(600) ack 395 win 17520 (DF) [tos 0x10]
192.150.187.20.ssh > 192.150.187.37.938: P [47504] 1000:1808(808) ack 395 win 17520 (DF) [tos 0x10]
192.150.187.20.ssh > 192.150.187.37.938: P [43917] 1808:1844(36) ack 395 win 17520 (DF) [tos 0x10]
```

Source Port (22, ssh)
Destination Port (938)
Seq Number (3641908893)
Ack Number (23916996)
Hdr Len (20)
Reserved (0)
Window Size (17520)
Checksum (31567)
Urgent Pointer (0)
53 packets.

Figure 5: Using NetDuDE to create test traffic

To this end, we developed NetDuDE (Figure 5), the Network Dump Displayer and Editor. NetDuDE takes `libpcap` packet tracefile, displays the packets graphically, and allows us to examine IP, TCP, UDP, and ICMP header fields.[5] In addition, it allows us to *edit* the tracefile, setting the values of fields, adding and removing options, recalculating checksums, changing the packet ordering, and duplicating, fragmenting, reassembling or deleting packets.

To test a particular normalization, we edit an existing trace to create the appropriate anomalies. We then feed the tracefile through *norm* to create a new normalized trace. We then both reexamine this trace in NetDuDE to manually check that the normalization we intended actually occurred, and feed the trace back into *norm*, to ensure that on a second pass it does not modify the trace further. Finally, we store the input and output tracefiles in our library of anomalous traces so that we can perform automated validation tests whenever we make a change to *norm*, to ensure that changing one normalization does not adversely affect any others.

7.2 Performance

As mentioned above, our current implementation of *norm* runs at user level, but we are primarily interested in assessing how well it might run as a streamlined kernel implementation, since it is reasonable to expect that a production normalizer will merit a highly optimized implementation.

To address this, *norm* incorporates a test mode whereby it reads an entire `libpcap` trace file into memory and in addition allocates sufficient memory to store all the resulting normalized packets. It then times how long it takes to run, reading packets from one pool of memory, normalizing them, and storing the results in the second memory pool. After measuring the performance, *norm* writes the second memory pool out to a `libpcap` trace file, so we can ensure that the test did in fact measure the normalizations we intended.

These measurements thus factor out the cost of getting packets to the normalizer and sending them out once the normalizer is done with them. For a user-level implementation, this cost is high, as it involves copying the entire packet stream up from kernel space to user space and then back down again; for a kernel implementation, it should be low (and we give evidence below that it is).

For baseline testing, we use three tracefiles:

Trace T1: a 100,000 packet trace captured from the Internet access link at the Lawrence Berkeley National Laboratory, containing mostly TCP traffic (88%) with some UDP (10%), ICMP (1.5%), and miscellaneous (IGMP, ESP, tunneled IP, PIM; 0.2%). The mean packet size is 491 bytes.

Trace U1: a trace derived from T1, where each TCP header has been replaced with a UDP header. The IP parts of the packets are unchanged from T1.

Trace U2: a 100,000 packet trace consisting entirely of 92 byte UDP packets, generated using *netcat*.

T1 gives us results for a realistic mix of traffic; there's nothing particularly unusual about this trace compared to the other captured network traces we've tested. U1 is totally unrealistic, but as UDP normalization is completely stateless with very few checks, it gives us a baseline number for how expensive the more streamlined IP normalization is, as opposed to TCP normalization, which includes many more checks and involves maintaining a control block for each flow. Trace U2 is for comparison with U1, allowing us to test what fraction of the processing cost is per-packet as opposed to per-byte.

We performed all of our measurements on an x86 PC running FreeBSD 4.2, with a 1.1GHz AMD Athlon Thunderbird processor and 133MHz SDRAM. In a barebones configuration suitable for a normalizer box, such a machine costs under US$1,000.

For an initial baseline comparison, we examine how fast *norm* can take packets from one memory pool and copy them to the other, without examining the packets at all:

Memory-to-memory copy only		
Trace	pkts/sec	bit rate
T1,U1	727,270	2856 Mb/s
U2	1,015,600	747 Mb/s

Enabling all the checks that *norm* can perform for both inbound and outbound traffic[6] examines the cost of performing the tests for the checks, even though most of them entail no actual packet transformation, since (as in normal operation) most fields do not require normalization:

All checks enabled		
Trace	pkts/sec	bit rate
T1	101,000	397 Mb/s
U1	378,000	1484 Mb/s
U2	626,400	461 Mb/s

Number of Normalizations					
Trace	IP	TCP	UDP	ICMP	Total
T1	111,551	757	0	0	112,308

[5] At the time of writing, ICMP support is still incomplete.

[6] Normally fewer checks would be enabled for outbound traffic.

Comparing against the baseline tests, we see that IP normalization is about half the speed of simply copying the packets. The large number of IP normalizations consist mostly of simple actions such as TTL restoration, and clearing the DF and Diffserv fields. We also see that TCP normalization, despite holding state, is not vastly more expensive, such that TCP/IP normalization is roughly one quarter of the speed of UDP/IP normalization.

These results do not, of course, mean that a kernel implementation forwarding between interfaces will achieve these speeds. However, the Linux implementation of the *click* modular router [7] can forward 333,000 small packets/sec on a 700MHz Pentium-III. The results above indicate that normalization is cheap enough that a normalizer implemented as (say) a *click* module should be able to forward normal traffic at line-speed on a bi-directional 100Mb/s link.

Furthermore, if the normalizer's incoming link is attacked by flooding with small packets, we should still have enough performance to sustain the outgoing link at full capacity. Thus we conclude that deployment of the normalizer would not worsen any denial-of-service attack based on link flooding.

A more stressful attack would be to flood the normalizer with small fragmented packets, especially if the attacker generates out-of-order fragments and intersperses many fragmented packets. Whilst a normalizer under attack can perform triage, preferentially dropping fragmented packets, we prefer to only do this as a last resort.

To test this attack, we took the T1 trace and fragmented every packet with an IP payload larger than 16 bytes: trace T1-frag comprises some 3 million IP fragments with a mean size of 35.7 bytes. Randomizing the order of the fragment stream over increasing intervals demonstrates the additional work the normalizer must perform. For example, with minimal re-ordering the normalizer can reassemble fragments at a rate of about 90Mb/s. However, if we randomize the order of fragments by up to 2,000 packets, then the number of packets simultaneously in the fragmentation cache grows to 335 and the data rate we can handle halves.

rnd intv'l	input frags/s	frag'ed bit rate	output pkts/sec	output bit rate	pkts in cache
100	299,670	86Mb/s	9,989	39Mb/s	70
500	245,640	70Mb/s	8,188	32Mb/s	133
1,000	202,200	58Mb/s	6,740	26Mb/s	211
2,000	144,870	41Mb/s	4,829	19Mb/s	335

It is clear that in the worst case, *norm* does need to perform triage, but that it can delay doing so until a large fraction of the packets are very badly fragmented, which is unlikely except when under attack.

The other attack that slows the normalizer noticeably is when *norm* has to cope with inconsistent TCP retransmissions. If we duplicate every TCP packet in T1, then this stresses the consistency mechanism:

All checks enabled		
Trace	pkts/sec	bit rate
T1	101,000	397 Mb/s
T1-dup	60,220	236 Mb/s

Although the throughput decreases somewhat, the reduction in performance is not grave.

To conclude, a software implementation of a traffic normalizer appears to be capable of applying a large number of normalizations at line speed in a bi-directional 100Mb/s environment using commodity PC hardware. Such a normalizer is robust to denial-of-service attacks, although in the specific case of fragment reassembly, very severe attacks may require the normalizer to perform triage on the attack traffic.

Acknowledgments

We'd like to thank Bill Fenner, Brad Karp, Orion Hodson, Yin Zhang, Kevin Fall, Steve Bellovin and the End-to-end Research Group for their comments and suggestions. Thanks also go to Jupiter's in Berkeley and the fine Guinness and Lost Coast brewing companies for lubricating many hours of discussion on this topic.

References

[1] M. Allman, D. Glover and L. Sanchez, "Enhancing TCP Over Satellite Channels using Standard Mechanisms," RFC 2488, Jan. 1999.

[2] Anzen Computing, *fragrouter*, 1999.
http://www.anzen.com/research/nidsbench/

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

[4] M. Handley, C. Kreibich, and V. Paxson, draft technical report, to appear at http://www.aciri.org/vern/papers/norm-TR-2001.ps.gz, 2001.

[5] horizon `<jmcdonal@unf.edu>`, "Defeating Sniffers and Intrusion Detection Systems", Phrack Magazine Volume 8, Issue 54, Dec. 25th, 1998.

[6] C. Kent and J. Mogul, “Fragmentation Considered Harmful,” *Proc. ACM SIGCOMM*, 1987.

[7] E. Kohler, R. Morris, B. Chen, J. Jannotti and M.F. Kaashoek, “The Click modular router,” *ACM Transactions on Computer Systems*, 18(3), pp. 263–297, Aug. 2000.

[8] G. R. Malan, D. Watson, F. Jahanian and P. Howell, “Transport and Application Protocol Scrubbing”, Proceedings of the IEEE INFOCOM 2000 Conference, Tel Aviv, Israel, Mar. 2000.

[9] L. Deri and S. Suin, “Improving Network Security Using Ntop,” *Proc. Third International Workshop on the Recent Advances in Intrusion Detection (RAID 2000),* Toulouse, France, Oct. 2000.

[10] S. McCanne, C. Leres and V. Jacobson, `libpcap`, 1994. ftp://ftp.ee.lbl.gov/libpcap.tar.Z

[11] K. Nichols, S. Blake, F. Baker and D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers”, RFC 2474, Dec. 1998.

[12] V. Paxson, “Bro: A System for Detecting Network Intruders in Real-Time”, Computer Networks, 31(23-24), pp. 2435-2463, 14 Dec 1999.

[13] V. Paxson and M. Handley, “Defending Against NIDS Evasion using Traffic Normalizers,” presented at *Second International Workshop on the Recent Advances in Intrusion Detection*, Sept. 1999.

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

[15] K. Ramakrishnan and S. Floyd, “A Proposal to add Explicit Congestion Notification (ECN) to IP”, RFC 2481, Jan. 1999.

[16] S. Sanfilippo, “new tcp scan method,” *Bugtraq*, Dec. 18, 1998.

[17] M. Smart, G.R. Malan and F. Jahanian, “Defeating TCP/IP Stack Fingerprinting,” *Proc. USENIX Security Symposium*, Aug. 2000.

[18] M. de Vivo, E. Carrasco, G. Isern and G. de Vivo, “A Review of Port Scanning Techniques,” *Computer Communication Review*, 29(2), April 1999.

A Normalizations performed by *norm*

Our normalizer implementation *norm* currently performs 54 of the following 73 normalizations we identified:

IP Normalizations

#	IP Field	Normalization Performed
1	Version	Non-IPv4 packets dropped.
2	Header Len	Drop if hdr_len too small.
3	Header Len	Drop if hdr_len too large.
4	Diffserv	Clear field.
5	ECT	Clear field.
6	Total Len	Drop if tot_len $>$ link layer len.
7	Total Len	Trim if tot_len $<$ link layer len.
8	IP Identifier	Encrypt ID.†
9	Protocol	Enforce specific protocols.†
–	Protocol	Pass packet to TCP,UDP,ICMP handlers.
10	Frag offset	Reassemble fragmented packets.
11	Frag offset	Drop if offset + len $>$ 64KB.
12	DF	Clear DF.
13	DF	Drop if DF set and offset $>$ 0.
14	Zero flag	Clear.
15	Src addr	Drop if class D or E.
16	Src addr	Drop if MSByte=127 or 0.
17	Src addr	Drop if 255.255.255.255.
18	Dst addr	Drop if class E.
19	Dst addr	Drop if MSByte=127 or 0.
20	Dst addr	Drop if 255.255.255.255.
21	TTL	Raise TTL to configured value.
22	Checksum	Verify, drop if incorrect.
23	IP options	Remove IP options.†
24	IP options	Zero padding bytes.†

† Indicates normalizations planned, but either not yet implemented or not yet tested at the time of writing.

Note that most normalizations are optional, according to local site policy.

UDP Normalizations

#	UDP Field	Normalization Performed
1	Length	Drop if doesn’t match length as indicated by IP total length.
2	Checksum	Verify, drop if incorrect.

TCP Normalizations

#	TCP Field	Normalization Performed
1	Seq Num	Enforce data consistency in retransmitted segments.
2	Seq Num	Trim data to window.
3	Seq Num	Cold-start: trim to keep-alive.
4	Ack Num	Drop ACK above sequence hole.
5	SYN	Remove data if SYN=1.
6	SYN	If SYN=1 & RST=1, drop.
7	SYN	If SYN=1 & FIN=1, clear FIN.
8	SYN	If SYN=0 & ACK=0 & RST=0, drop.
9	RST	Remove data if RST=1.
10	RST	Make RST reliable.
11	RST	Drop if not in window.†
12	FIN	If FIN=1 & ACK=0, drop.
13	PUSH	If PUSH=1 & ACK=0, drop.
14	Header Len	Drop if less than 5.
15	Header Len	Drop if beyond end of packet.
16	Reserved	Clear.
17	ECE, CWR	Optionally clear.
18	ECE, CWR	Clear if not negotiated.†
19	Window	Remove window withdrawals.
20	Checksum	Verify, drop if incorrect.
21	URG,urgent	Zero urgent if URG not set.
22	URG,urgent	Zero if urgent > end of packet.
23	URG	If URG=1 & ACK=0, drop.
24	MSS option	If SYN=0, remove option.
25	MSS option	Cache option, trim data to MSS.
26	WS option	If SYN=0, remove option.
27	SACK pmt'd	If SYN=0, remove option.
28	SACK opt	Remove option if length invalid.†
29	SACK opt	Remove if left edge of SACK block > right edge.†
30	SACK opt	Remove if any block above highest seq. seen.†
31	SACK opt	Trim any block(s) overlapping or continguous to cumulative acknowledgement point.†
32	T/TCP opts	Remove if NIDS doesn't support.
33	T/TCP opts	Remove if under attack.†
34	TS option	Remove from non-SYN if not negotiated in SYN.†
35	TS option	If packet fails PAWS test, drop.†
36	TS option	If echoed timestamp wasn't previously sent, drop.†
37	MD5 option	If MD5 used in SYN, drop non-SYN packets without it.†
38	*other opts*	Remove options.

ICMP Normalizations

#	ICMP Type	Normalization Performed
1	Echo request	Drop if destination is a multicast or broadcast address.
2	Echo request	Optionally drop if ping checksum incorrect.
3	Echo request	Zero "code" field.
4	Echo reply	Optionally drop if ping checksum incorrect.
5	Echo reply	Drop if no matching request.†
6	Echo reply	Zero "code" field.
7	Source quench	Optionally drop to prevent DoS.†
8	Destination Unreachable	Unscramble embedded scrambled IP identifier.†
9	*other*	Drop.†

The following "transport" protocols are recognized, but currently passed through unnormalized: IGMP, IP-in-IP, RSVP, IGRP, PGM.